jetzt lerne ich

Start ohne Vorwissen

Visual C++ 2008

Der schnelle und einfache Einstieg in die Windows-Programmierung

DIRK LOUIS

Markt+Technik

Kapitel 3

jetzt lerne ich

C++: Datenverarbeitung

Im vorangehenden Kapitel haben wir bereits zwei erste Programme aufgesetzt: Windows-Anwendungen mit grafischer Benutzeroberfläche und Steuerelementen. Für all dies mussten wir nicht einmal richtig programmieren – dank der RAD-Umgebung von Visual C++ konnten wir uns alles mit der Maus zusammenklicken. Als wir aber daran gingen, eine Ereignisbehandlung für das Drücken des Schalters aufzusetzen, haben wir die Grenzen der RAD-Umgebung kennen gelernt. Sie ist wunderbar geeignet, um grafische Benutzeroberflächen zusammenzustellen oder Software-Bausteine, die als Komponenten zur Verfügung stehen, in ein Programm zu integrieren, doch wenn es darum geht, den funktionellen Code aufzusetzen, der hinter der schönen grafischen Oberfläche steht, dann sind wir wieder auf uns allein gestellt. Dann hilft uns keine RAD-Umgebung mehr, dann helfen nur noch gute C++-Kenntnisse. Die nächsten drei Kapitel sind daher der grundlegenden C++-Syntax gewidmet.

Sie lernen in diesem Kapitel,

- was Konsolenanwendungen sind und wie man sie erstellt,
- wie ein C++-Programm mit Daten umgeht,
- was Variablen, Konstanten und Anweisungen sind,
- was Datentypen und Operatoren sind,
- was Klassen sind,
- was Arrays sind,
- welche Bedeutung den Namespaces zukommt.

3.1 Konsolenanwendungen

C++-Programme gibt es in verschiedenen Ausprägungen. So bezeichnen wir zum Beispiel Anwendungen, die über eine grafische Benutzeroberfläche (also Fenster, Schalter, etc.) verfügen, als GUI- oder Windows-Anwendungen, wohingegen Programme, die ohne eine solche Benutzeroberfläche auskommen, unter dem Begriff *Konsolenanwendungen* zusammengefasst werden. Die Bezeichnung rührt daher, dass Ein- und Ausgaben nur über eine Konsole[1] erfolgen können.

Reine Konsolenanwendungen werden heutzutage eigentlich nur noch für Aufgaben verwendet, die unsichtbar im Hintergrund erledigt werden können – beispielsweise für die Implementierung von Webservern. In der bunten, Multimedia-verwöhnten Welt der Endanwender spielen Konsolenanwendungen keine große Rolle mehr, da heute jedermann nach Programmen verlangt, die grafische Benutzeroberflächen (GUIs = *Graphical User Interfaces*) haben. Trotzdem werden wir uns in diesem und dem nächsten Kapitel ausschließlich mit Konsolenanwendungen beschäftigen, und zwar aus didaktischen Gründen. Wenn wir uns in der Folge mit der Definition von Variablen, mit Operatoren, Bedingungen und Schleifen oder mit den Grundkonzepten der objektorientierten Programmierung beschäftigen, soll uns nichts von der eigentlichen Thematik ablenken. Darum verzichten wir auf die grafische Oberfläche und konzentrieren uns ganz auf den funktionellen Code.

3.1.1 Das Grundgerüst

Das minimale Grundgerüst einer C++-Anwendung ohne grafische Benutzeroberfläche ist erstaunlich kurz. Tatsächlich braucht man im Grunde ja nichts anderes als eine `main()`-Funktion. Da die Programmausführung, wie Sie aus Kapitel 2 bereits wissen, mit dem Aufruf dieser Funktion beginnt, müssen Sie nur den gesamten Programmcode in den Rumpf dieser Funktion schreiben.

Listing 3.1: Ein minimales Anwendungsgerüst (bei Verwendung der C++-Standardbibliothek)

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

int main()
{
    cout << "Hallo Welt!" << endl;

    return 0;
}
```

1. Unter Windows XP/Vista ist dies die Eingabeaufforderung, die Sie über Start/Alle Programme/Zubehör aufrufen können.

Listing 3.1 zeigt uns, wie ein minimales ausführbares C++-Programm aussehen könnte. Es besteht aus nicht mehr als

- einer `#include`- und einer `using`-Anweisung, die uns den Zugriff auf die C++-Bibliothek erschließen, und
- der `main()`-Funktion, in der die auszuführenden Anweisungen stehen.

Das Programm verwendet aber noch die traditionelle C++-Standardbibliothek (siehe Kapitel 1.4.3). So verbirgt sich beispielsweise hinter dem Bezeichner `cout` ein Objekt einer Bibliotheksklasse, das die Konsole repräsentiert. Gleiches gilt für den Bezeichner `endl`, der einen Zeilenumbruch repräsentiert. Und der <<-Operator, der den auszugebenden Text und den Zeilenumbruch zur Konsole schickt, ist ebenfalls in der Standardbibliothek definiert! (Die zugehörigen Deklarationen stehen allesamt in `iostream`.)

Wir wollen uns in diesem Buch aber vorrangig auf die C++-Programmierung für das .NET Framework konzentrieren. Wie also sieht der obige Quelltext aus, wenn wir statt der C++-Standardbibliothek die .NET-Bibliothek verwenden? Wird er umfangreicher, wird er komplizierter? Nun, umfangreicher wird er nicht, aber vielleicht sogar ein bisschen besser verständlich.

```
using namespace System;

int main()
{
    Console::WriteLine("Hallo Welt!");

    return 0;
}
```

Listing 3.2: Ein minimales Anwendungsgerüst (bei Verwendung der .NET-Bibliothek)

Statt des Namespace `std` (für die Elemente der C++-Standardbibliothek) verwenden wir den Namespace `System`, in dem die grundlegenden Klassen der .NET-Bibliothek definiert sind. Eine dieser Klasse ist `Console`, welche die Funktionalität zur Arbeit mit dem Konsolenfenster bereitstellt.

Die `#include`-Direktive entfällt, da die Elemente der .NET-Bibliothek ohne vorherige Deklaration verwendet werden können (siehe Kapitel 1.4)

Statt mit Hilfe von `cout` und dem <<-Operator geben wir den Text `"Hallo Welt"` mit Hilfe der `Console`-Methode `WriteLine()` aus – die dankenswerter Weise die Ausgabe automatisch mit einem Zeilenumbruch abschließt.

Wenn das Programm gestartet wird, beginnt der JIT-Compiler die Ausführung, indem er die `main()`-Funktion sucht und dort zur ersten Anweisung (in unserem Beispiel also zum `Console::WriteLine()`-Aufruf) springt und diese ausführt. Danach werden schrittweise alle weiteren Anweisungen ausgeführt, bis die schließende Klammer der `main()`-Funktion erreicht wird. Im obigen Miniprogramm gibt es nur eine Anweisung, die den Text »Hallo Welt!« ausgibt.

In einem C++-Programm darf es nur eine einzige Quelltextdatei mit einer `main()`-Funktion geben.

3.1.2 Konsolenanwendungen in Visual C++

Nun wird es Zeit, dieses Programm mit Visual C++ zu erstellen. Zum Glück bietet Visual C++ direkte Unterstützung für Konsolenanwendungen, d.h., es gibt unter den von Visual C++ vorinstallierten Vorlagen (Aufruf über DATEI/NEU) ein Symbol namens CLR-KONSOLENANWENDUNG, das Ihnen die lästige Arbeit, ein Programmgerüst zu erstellen, abnimmt.

1. Rufen Sie den Menübefehl DATEI/NEU/PROJEKT auf, wählen Sie in dem Dialogfeld zuerst den Projekttyp VISUAL C++/CLR und dann als Vorlage CLR-KONSOLENANWENDUNG. Geben Sie als Programmnamen *HalloWelt* ein, wählen Sie den Speicherort und klicken Sie auf OK.

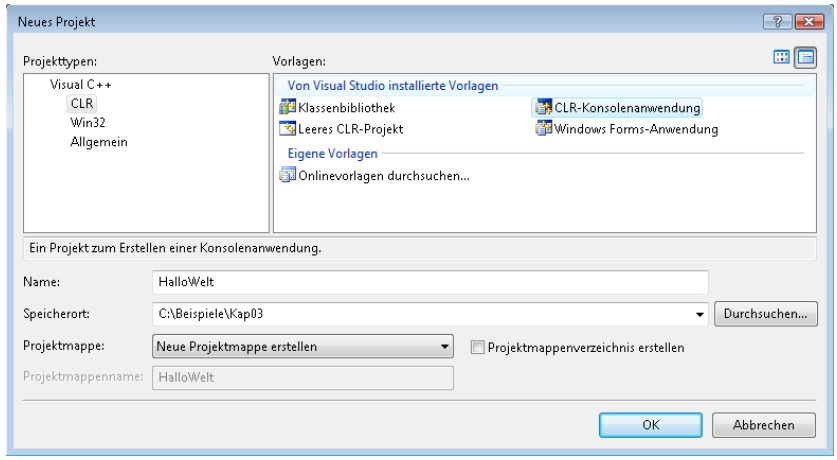


Abb. 3.1: Eine Konsolenanwendung mit Visual C++ anlegen

Im Editor von Visual C++ wird das Programmgerüst der Konsolenanwendung, *HalloWelt.cpp*, angezeigt. Es sieht ein wenig anders aus als das Minimalprogramm aus Listing 3.2. Eine `#include`-Anweisung ist hinzugekommen, die aber im Grunde überflüssig ist. (Visual C++ benötigt sie, um die Kompilation mit Hilfe eines vorkompilierten Headers beschleunigen zu können.) Und die `main()`-Funktion definiert zwischen den runden Klammern einen Array-Parameter zum Entgegennehmen von Argumenten aus dem Programmaufruf. Beides können wir getrost ignorieren.

In den nachfolgenden Listings sind die `#include`-Direktive für `stdafx` und die Parameterdefinition von `main()` nicht mit abgedruckt, da sie für den Quelltext ohne Bedeutung sind. Sie müssen sie deswegen aber nicht jedes Mal weglöschen, wenn Sie ein neues Projekt anlegen, um eines der Beispiele aus dem Buch nachzustellen.

Sollten Sie sich aber an der Direktive oder der Parameterdefinition stören, können Sie sie einfach weglöschen. Wenn Sie die `#include`-Direktive löschen, müssen Sie dem Compiler zusätzlich mitteilen, dass er keine vorkompilierten Header erstellen soll. Rufen Sie dazu den Befehl PROJEKT/...-EIGENSCHAFTEN auf und setzen Sie im Dialogfeld <PROJEKT>-EIGENSCHAFTENSEITEN auf der Seite C/C++/VORKOMPILIERTE HEADER die Option für die vorkompilierten Header auf *Vorkompilierte Header nicht verwenden*.

Passen wir nun das vorgegebene Programmgerüst an, indem wir den altbekannten Programmierergruß »Hallo Welt!« ausgeben.

2. Ersetzen Sie den Inhalt der runden Klammern von `WriteLine()` durch `"HalloWelt!"`:

   ```
   Console::WriteLine("Hallo Welt!");
   ```

3. Speichern Sie das Programm (Strg ⇧ S).
4. Kompilieren Sie es (Menübefehl ERSTELLEN/<PROJEKT> ERSTELLEN) und führen Sie es anschließend aus (Strg F5).

Es öffnet sich das Konsolenfenster, mit der von Ihnen vorgegebenen Begrüßung. Durch Drücken einer beliebigen Taste lässt sich das Konsolenfenster wieder schließen.

3.1.3 Konsolenanwendungen außerhalb von Visual C++ ausführen

Wenn Sie Konsolenanwendungen wie Windows-Anwendungen starten, also etwa durch Doppelklicken im Windows Explorer oder eines zuvor angelegten Desktop-Symbols (vgl. Kapitel 2.5), ergibt sich unter Umständen ein Problem: Sie sehen nichts von dem Programm!

Konsolenanwendungen stammen aus einer Zeit, als es noch keine grafischen Desktops wie unter Windows gab. Damals war der ganze Bildschirm ausgefüllt von einer schwarzen Konsole, auf der der Benutzer Befehle eintippen konnte. Über diese Konsole nahm die Anwendung Eingaben entgegen und auf sie gab die Anwendung ihre Ausgaben aus.

Seitdem haben sich die Zeiten gründlich gewandelt. Wenn Sie heute eine Konsolenanwendung unter Windows aufrufen, fehlt der Anwendung die Konsole. Das Windows-Betriebssystem springt der Anwendung daher zu

Hilfe und stellt ihr automatisch ein passendes Konsolenfenster zur Verfügung. Nur leider schließt Windows dieses Konsolenfenster auch automatisch, wenn die Konsolenanwendung beendet ist. Dies führt schnell dazu, dass dem Anwender keine Zeit bleibt, die Ausgaben der Konsolenanwendung zu lesen.

Die Lösung zu diesem Problem ist, zuerst ein Konsolenfenster zu öffnen und dann von diesem aus die Konsolenanwendung zu starten. Dann wird das Konsolenfenster nicht mehr von Windows verwaltet, sondern von Ihnen, und Sie haben alle Zeit der Welt, um die Ausgaben der Anwendung zu lesen oder sie ein zweites Mal auszuführen oder noch weitere Anwendungen zu starten.

Um die Konsolenanwendung *HalloWelt* von einer Konsole aus zu starten, gehen Sie wie folgt vor:

1. Öffnen Sie ein Konsolenfenster.

   Unter Windows XP/Vista öffnen Sie dazu das Start-Menü und wählen unter ALLE PROGRAMME\ZUBEHÖR den Eintrag EINGABEAUFFORDERUNG. Unter älteren Windows-Betriebssystemen heißt die Konsole manchmal auch MS-DOS-EINGABEAUFFORDERUNG oder ist direkt unter START/PROGRAMME zu finden.

2. Wechseln Sie in der Konsole mit Hilfe des *cd*-Befehls in das Verzeichnis mit der *.exe*-Datei, also beispielsweise:

   ```
   Prompt> cd C:\Beispiele\Kap03\HalloWelt\Debug
   ```

3. Starten Sie die Anwendung über ihren Namen.

   ```
   Prompt> HalloWelt
   ```

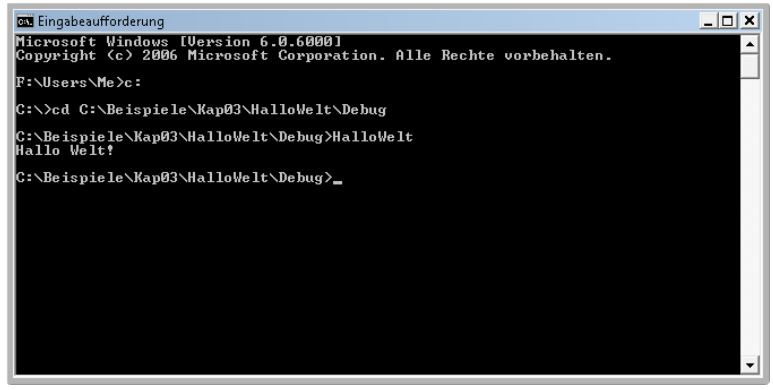


Abb. 3.2: Ausführung einer Konsolenanwendung

Die Konsole ist eine Schnittstelle zum Betriebssystem, über die Sie Zeile für Zeile Befehle abschicken können. Die Eingabe erfolgt in der jeweils untersten Zeile nach dem sogenannten Prompt (der per Voreinstellung meist das aktuelle Verzeichnis angibt). Jeder Befehl muss durch Drücken der ↵-Taste abgeschickt werden. (Mehr Informationen zur Bedienung der Konsole finden Sie als Tutorial zum Herunterladen auf der Website *www.carpelibrum.de*.)

Die Konsole merkt sich die zuletzt eingegebenen Befehle. Wenn Sie also eine Anwendung mehrmals ausführen wollen, müssen Sie dazu den Namen der Anwendung nicht jedes Mal neu eingeben, sondern können mit Hilfe der Pfeiltasten durch die Konsolen-History laufen, bis der Name wieder in der Eingabezeile steht.

3.2 Datentypen und Variablen

Die Aufgabe eines jeden Computerprogramms ist die Verarbeitung von irgendwelchen Informationen, die im Computerjargon meist Daten genannt werden. Das können Zahlen sein, aber auch Buchstaben, ganze Texte oder Bilder und Zeichnungen. Dem Rechner ist diese Unterscheidung gleich, da er letztlich alle Daten in Form von endlosen Zahlenkolonnen in Binärdarstellung (nur Nullen und Einsen) verarbeitet.

Stellen Sie sich vor, wir schreiben das Jahr 1960 und Sie sind stolzer Besitzer einer Rechenmaschine, die Zahlen und Text verarbeiten kann. Beides allerdings in Binärformat. Um Ihre Freunde zu beeindrucken, lassen Sie den »Computer« eine kleine Subtraktion berechnen, sagen wir:

```
8754 – 398 = ?
```

Zuerst rechnen Sie die beiden Zahlen durch fortgesetzte Division mit 2 ins Binärsystem um (wobei die nicht teilbaren Reste der aufeinander folgenden Divisionen, von rechts nach links geschrieben, die gewünschte Binärzahl ergeben).

```
10001000110010 – 110001110 = ?
```

Die Binärzahlen stanzen Sie sodann als Lochkarte und lassen diese von Ihrem Computer einlesen. Dann drücken Sie noch die Taste für Subtraktion und ohne Verzögerung erscheint das korrekte Ergebnis:

```
10000010100100
```

Zweifelsohne werden Ihre Freunde von dieser Maschine äußerst beeindruckt sein. Trotzdem lässt sich nicht leugnen, dass die Interpretation der Binärzahlen etwas unhandlich ist, und zwar erst recht, wenn man neben einfachen ganzen Zahlen auch Gleitkommazahlen, Texte und Bitmaps in Binärformat speichert.

Für die Anwender von Computern ist dies natürlich nicht zumutbar und die Computer-Revolution – die vierte der großen Revolutionen (nach der Glorious Revolution, England 1688, der Französischen Revolution von 1789 und der Oktoberrevolution, 1917 in Russland) – konnte nur stattfinden, weil man einen Ausweg fand. Dieser bestand einfach darin, es der Software – dem laufenden Programm – zu überlassen, die vom Anwender eingegebenen Daten (seien es Zahlen, Text, Bilder etc.) ins Binärformat umzuwandeln und umgekehrt die auszugebenden Daten wieder vom Binärformat in eine leicht lesbare Form zu verwandeln.

»Gemeinheit«, werden Sie aufbegehren, »da wurde das Problem ja nur vom Anwender auf den Programmierer abgewälzt«. Ganz so schlimm ist es nicht. Der C++-Compiler nimmt uns hier das Gröbste ab. Alles, was wir zu tun haben, ist, dem Compiler anzugeben, mit welchen Daten wir arbeiten möchten und welchem Datentyp diese Daten angehören (sprich, ob es sich um Zahlen, Text oder Sonstiges handelt).

Betrachten wir ein Beispiel:

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

    ersteZahl = 8754;
    zweiteZahl = 398;

    ...
}
```


Obiger Quelltextauszug stellt kein vollständiges Programm dar. Die Auslassungpunkte ... sind kein korrekter C++-Code, sie sollen Ihnen nur anzeigen, dass hier weiterer Code stehen könnte oder sollte.

In der `main()`-Funktion werden zuerst die benötigten Variablen definiert.

Variablen

Die Variablen eines Programms sind nicht mit den Variablen mathematischer Berechnungen gleichzusetzen. *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. 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 an, sondern auch deren Datentyp. Dieser Datentyp verrät dem Compiler, wie der Inhalt des Speicherbereichs der Variablen zu interpretieren ist. Im obigen Beispiel benutzen wir den Datentyp `int`, der für einfache Ganzzahlen steht.

Zu jeder Variablendefinition gehört auch die Angabe eines *Datentyps*. Dieser gibt dem Compiler an, wie der Speicherinhalt der Variablen zu interpretieren ist.

```
int ersteZahl;
```

Dem Konzept der Datentypen verdanken wir es, dass wir der Variablen `ersteZahl` eine Ganzzahl zuweisen können und nicht etwa wie im obigen Beispiel des Lochkartenrechners die Dezimalzahl in Binärcode umrechnen müssen.

```
ersteZahl = 8754;
```

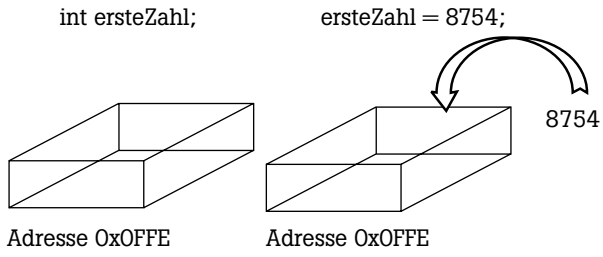


Abb. 3.3: Definition und Zuweisung

3.2.1 Der »Wert« der Variablen

Wenn eine Variable einen Speicherbereich bezeichnet, dann ist der Wert einer Variablen der interpretierte Inhalt dieses Speicherbereichs. Im obigen Beispiel ist der Wert der Variablen `ersteZahl` nach der Anweisung

```
ersteZahl = 8754;
```

also 8754. Wenn Sie der Variablen danach einen anderen Wert zuweisen, beispielsweise

```
ersteZahl = 5;
```

ist der Wert in der Folge gleich 5.

Was die Variablen für den Programmierer aber so wertvoll macht, ist, dass er sich nicht mehr um die Speicherverwaltung zu kümmern braucht. Es ist zwar von Vorteil, wenn man weiß, dass hinter einer Variablen ein Speicherbereich steht, für die tägliche Programmierarbeit ist es aber meist nicht erforderlich. Wir sprechen nicht davon, dass wir mit Hilfe des Variablennamens einen eindeutig bezeichneten Platz im Arbeitsspeicher referenzieren und in diesen einen Wert schreiben; wir sagen einfach, dass wir der Variablen einen Wert zuweisen. Wir sprechen nicht davon, dass das interpretierte Bitmuster in dem Speicherbereich der `int`-Variablen `ersteZahl` gleich 5 ist; wir sagen einfach, `ersteZahl` ist gleich 5. Wir sprechen nicht davon, dass wir mit Hilfe des Variablennamens einen eindeutig bezeichneten Platz im Arbeitsspeicher referenzieren und dessen Wert auslesen; wir sagen einfach, dass wir den Wert der Variablen auslesen.

3.2.2 Variablen bei der Definition initialisieren

Wenn Sie möchten, können Sie einer Variablen auch direkt bei der Definition einen Anfangswert zuweisen. In einem solchen Falle spricht man von *Initialisierung*.

```
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 eine Formel einzubauen oder auszugeben.

Listing 3.3: Programmieren mit Variablen (aus dem Projekt Variablen)

```
using namespace System;

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

    ergebnis = ersteZahl - zweiteZahl;

    Console::WriteLine("8754 - 398  = " + ergebnis);

    return 0;
}
```

Der hier verwendete Aufruf von `WriteLine()` bedarf einer Erklärung. Unsere Intention ist es, die Ausgabe

8754 – 398 = 8356

zu erzeugen. Die Aufgabenstellung bis zum Gleichheitszeichen geben wir als String vor, das Ergebnis soll aus dem Wert der Variablen `ergebnis` übernommen werden. Hierfür gibt es verschiedene Möglichkeiten:

Eine Möglichkeit ist, wie im Beispiel, die Ausgabe mit Hilfe des +-Operators aus Teilstring und Variable zusammenzusetzen:

```
Console::WriteLine("8754 - 398  = " + ergebnis);
```

Eine andere Möglichkeit ist, in den auszugebenden String einen nummerierten Platzhalter einzufügen und der `WriteLine()`-Methode hinter dem String die Variable zu übergeben, durch deren Wert der Platzhalter ersetzt werden soll (mehr dazu in Kapitel 7.1).

Variablen, die Sie innerhalb einer Funktion (oder Methode) definieren, sind nur in der Funktion und nur in den nachfolgenden Anweisungen gültig.

3.2.4 Die elementaren Datentypen

Zu jeder Variablendefinition gehört die Angabe des Datentyps, der dem Compiler mitteilt, welche Werte der Variablen zugewiesen werden können. Neben dem Datentyp `int` für Ganzzahlen kennt C++ noch eine Reihe weiterer elementarer Datentypen:

Tabelle 3.1: Die wichtigsten elementaren Datentypen

Datentyp	Beschreibung	Werte	.NET-Typ
`bool`	Boolescher Wert (wahr, falsch)	`true`, `false`	`System::Boolean`
`char`	Zeichen, Buchstabe	`'a'`, `'2'`	`System::Byte`
`wchar_t`	Zeichen, Buchstabe	`'a'`, `'2'` `'\u00A3'` (Unicode für das Zeichen £)	`System::Char`
`short`	ganze Zahl	–32.768 bis 32.767	`System::Int16`
`int`	ganze Zahl	–2.147.483.648 bis +2.147.483.647	`System::Int32`
`long`	ganze Zahl	–9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807 auf anderen Systemen, bei anderen Compilern identisch zu `int`	`System::Int64`
`float`	Gleitkommazahl	ca. 1,5 x 10^{-38} bis 3,4 x 10^{38}	`System::Single`
`double`[1]	Gleitkommazahl	ca. 1,7 x 10^{-308} bis 1,7 x 10^{308}	`System::Double`

1. Für Gleitkommazahlen (Zahlen mit Nachkommaanteil) sollten Sie generell den Datentyp `double` verwenden. Mit diesem lassen sich nicht nur größere Werte darstellen, die Zahlen lassen sich zudem in größerer Präzision (mit mehr Nachkommastellen) darstellen.

Unicode

Unicode ist ein standardisierter Zeichensatz mit 65.536 Zeichen, mit dem alle diversen Umlaute und Sonderzeichen aller gängigen Sprachen, ja sogar japanische und chinesische Schriftzeichen, dargestellt werden können!

Wie Sie sehen, gibt es verschiedene Datentypen mit unterschiedlichen Wertebereichen. Um zum Beispiel eine ganze Zahl abzuspeichern, haben

Sie die Wahl zwischen `short`, `int` und `long`! Die größeren Wertebereiche erkauft man sich mit einem höheren Speicherverbrauch. Eine `long`-Variable benötigt beispielsweise doppelt so viel Speicher wie eine `int`-Variable. Glücklicherweise ist Arbeitsspeicher kein allzu großes Problem mehr und viele Programmierer verwenden standardmäßig `int` oder `long` für ganzzahlige Werte und `double` für Gleitkommazahlen.

Der *Datentyp* legt also nicht nur fest, wie der Wert der Variablen zu interpretieren ist, er gibt auch an, wie groß der für die Variable bereitzustellende Speicherbereich sein muss.

Hier einige Beispiele:

```
int ganzeZahl;
double krummeZahl;
bool antwort;
short klein = -4;
char buchstabe = 'Ü';
char ziffer = '4';
```

Es ist auch möglich, mehrere Variablen des gleichen Typs durch Komma getrennt auf einmal zu definieren:

```
bool ja, nein, oder_doch;
```

Die .NET-Typen für die elementare Datentypen

In Tabelle 3.1 ist zu jedem C++-Typ noch ein .NET-Typ angegeben. In diesem Buch werden wir nur selten mit diesen Datentypen zu tun haben, aber Sie sollten Sie zumindest kennen und zuordnen können. Und manchmal werden sie ja auch benötigt.

Denken Sie daran, dass die .NET-Bibliothek nicht nur sehr umfangreich, sondern auch sprachunabhängig ist. D.h. ihre Klassen und Methoden können zum Speichern von Werten nicht die C++-Typen wie `int` oder `double` verwenden. Die .NET-Bibliothek präsentiert die elementaren Datentypen daher durch eigene Datentypen – eben die in der Tabelle aufgeführten Typen wie `System::Boolean` und so weiter.

Wenn wir nun eine .NET-Methode aufrufen, die als Parameter einen `System::Int32`-Wert verlangt, ist dies kein Problem. Wir übergeben einfach einen `int`-Wert und überlassen es dem Compiler, den `int`-Wert in ein `Int32`-Objekt umzuwandeln. Für komplexere Datentypen ist diese Vorgehensweise allerdings nicht möglich. Dann ist es einfacher, die zu übergebenden Variablen direkt mit den .NET-Typen zu definieren (siehe z.B. den Beispielcode zur `String::Split()`-Methode in Tabelle 3.4.

3.2.5 Strings

Strings haben wir bisher fast ausschließlich als String-Konstanten verwendet – beispielsweise in den Aufrufen von `Console::WriteLine()`:

```
Console::WriteLine("Hallo Welt!");
```


Vielleicht ist Ihnen in dem von Visual C++ vorgegebenen Grundgerüst, das »L« vor dem String aufgefallen? Dieses `L` teilt dem Compiler mit, dass es sich um einen String mit Unicode-Zeichen handelt (Datentyp `wchar_t` statt `char`). Sie müssen das `L` nur setzen, wenn Sie besondere Zeichen wie z.B. verwenden.

String-Konstanten

Wie Sie bereits gesehen haben, werden String-Konstanten dadurch gekennzeichnet, dass sie mit doppelten Anführungszeichen beginnen und mit doppelten Anführungszeichen enden.

Die Frage ist nur, was passiert, wenn ein String selbst doppelte Anführungszeichen enthält.

```
"Der Pfarrer sprach: "Der Tod ist der Sünde Sold.""
```

Wenn Sie versuchen, einen solchen String auszugeben oder anderweitig zu verarbeiten, werden Sie dafür vom Compiler einen Haufen wütender Fehlermeldungen empfangen. Der Grund ist, dass der Compiler den String nicht nach dem letzten, sondern direkt nach dem zweiten doppelten Anführungszeichen für beendet ansieht. Aus Sicht des Compilers besteht die obige Zeile aus dem String `"Der Pfarrer sprach: "`, gefolgt von den Zeichen `Der Tod ist der Sünde Sold.`, mit denen er überhaupt nichts anfangen kann, und schließlich einem zweiten, leeren String `""`.

Um dennoch Strings mit doppelten Anführungszeichen definieren und verarbeiten zu können, bräuchte man einen Weg, wie man dem Compiler anzeigen kann, dass das folgende doppelte Anführungszeichen eben nicht das Ende des Strings signalisiert, sondern als normales Zeichen zu behandeln ist. Diese Möglichkeit eröffnet uns das Escape-Zeichen.

Escape-Sequenzen

Mit Hilfe des *Escape-Zeichens* `\` kann man zum einen Zeichen, die für den Compiler eine besondere Bedeutung haben (`"` oder `\`), in Strings als einfache Textzeichen kennzeichnen (`\"` oder `\\`), zum anderen kann man bestimmte Sonderzeichen in einen Text einfügen (beispielsweise `\t` zum Einfügen eines Tabulators).

Escape-Sequenz	Beschreibung
\'	Einfaches Anführungszeichen
\"	Doppeltes Anführungszeichen
\\	Backslash
\0	Null-Zeichen
\a	Signalton
\b	Rückschritttaste
\f	Seitenvorschub
\n	Neue Zeile (Zeilenumbruch)
\t	Horizontaler Tabulator
\v	Vertikaler Tabulator

Tabelle 3.2: Die Escape-Sequenzen

Um beispielsweise doppelte Anführungszeichen in einem String zu verwenden, brauchen Sie den Anführungszeichen im String lediglich das Escape-Zeichen \ voranzustellen:

```
"Der Pfarrer sprach: \"Der Tod ist der Sünde Sold.\""
```

Mit Hilfe der Escape-Sequenzen \n und \t können Sie Textausgaben übersichtlicher gestalten:

```
using namespace System;

int main()
{
    Console::WriteLine();
    Console::WriteLine(" Daten von Dirk Louis: \n");
    Console::WriteLine("\t Alter     : 41");
    Console::WriteLine("\t Größe     : 178 cm");
    Console::WriteLine("\t Gewicht   : 77 kg");
    Console::WriteLine("\n");
}
```

Listing 3.4: Umbrüche und Tabulatoren in Strings (aus dem Projekt Escapezeichen)

Abb. 3.4: Mit Hilfe von Tabulatoren, Leerzeichen und Umbrüchen formatierte Ausgabe

Beachten Sie, dass die Doppelpunkte in den Zeilen durch Leerzeichen untereinander ausgerichtet wurden. Würde man hinter den einzelnen Eigenschaften Tabulatoren einfügen, wäre nicht sichergestellt, dass die Doppelpunkte untereinander stehen. Dies liegt daran, dass Tabulatoren bis zu vorgegebenen Zeichenspalten vorrücken. Wenn das Ende von `Alter` zufällig vor und das Ende von `Groesse` und `Gewicht` hinter einer solchen Spalte liegen, springen die Tabulatoren hinter den Wörtern zu unterschiedlichen Spalten.

Zeilenumbrüche und Aneinanderreihung von Strings

Manchmal müssen Sie mit Strings arbeiten, die so lang sind, dass Sie sie am liebsten auf mehrere Zeilen verteilen würde.

```
Console::WriteLine("Dies ist ein langer Text, den man im Quelltext am
liebsten auf mehrere Zeilen verteilen würde!");
```

Dieser Text ist nur für den Ausdruck in diesem Buch umgebrochen worden. Im Quelltext eines Programms würde er in einer Zeile stehen – mit dem Effekt, dass das Ende des Textes ohne Scrollen nicht mehr zu lesen ist.

Wenn Sie aber im Quelltext einen Umbruch in eine String-Konstante einfügen, ernten Sie dafür später beim Kompilieren eine Fehlermeldung, denn Umbrüche in String-Konstanten sind nicht erlaubt.

Die Lösung zu diesem Problem besteht darin, den String in zwei oder mehrere Teilstrings aufzubrechen und diese mit Hilfe des Operators + wieder aneinanderzuhängen. Die einzelnen Teilstrings können Sie dann auf mehrere Zeilen verteilen:

```
Console::WriteLine("Dies ist ein langer Text, den man"
                 + "im Quelltext am liebsten auf mehrere"
                 + "Zeilen verteilen würde!");
```


Den +-Operator für Strings nennt man auch Verkettungs- oder *Konkatenationsoperator*. Er ist der einzige der arithmetischen Operatoren, der auch für Strings definiert ist.

String-Variablen

Mit Hilfe des Datentyps `String` können Sie Variablen für Strings definieren.

```
String meinStr;
```

Diesen können Sie String-Konstanten oder Strings aus anderen `String`-Variablen zuweisen.

```
String ^str1 = "Hallo Welt!";
String ^str2;

str2 = str1;
```

Beachten Sie das ^-Zeichen vor den Variablennamen. Dieses Zeichen muss immer dann bei der Definition vor dem Variablennamen stehen, wenn die Variable vom Typ einer .NET-Klasse ist – und genau dies ist bei `String` der Fall, es ist die Klasse `String` aus dem Namespace `System`.

Obwohl es ein wenig lästig ist, den Variablen von .NET-Typen bei der Definition ein ^ voranstellen zu müssen (zumal das Dachsymbol immer erst erscheint, wenn ein weiteres Zeichen eingegeben wurde), hat es doch den Vorteil, dass wir mit einem Blick erkennen können, hinter welchen Variablen sich .NET-Objekte verbergen.

.NET unterscheidet zwischen Verweis- und Werttypen

Die meisten Datentypen in der .NET-Bibliothek sind Verweistypen. Ihre Variablen werden wie hier am Beispiel der String-Variablen mit dem ^-Symbol definiert. Außerdem werden Methoden für diese Variablen mit Hilfe des Operators -> aufgerufen (siehe Tabelle 3.4).

Daneben gibt es noch die sogenannten Werttypen, zu denen beispielsweise die oben in Tabelle 3.1 vorgestellten Typen für die elementaren Datentypen gehören. Variablen dieser Typen werden ganz normal, ohne ^-Symbol, definiert und wenn Sie Methoden dieser Variablen aufrufen, geschieht dies mit Hilfe des `.`-Operators.

C++

Wenn Sie für das .NET Framework programmieren, verwenden Sie für String-Variablen am besten die .NET-Klasse `System::String`. Wenn Sie ein traditionelles C++-Programm schreiben und mit der C++-Standardbibliothek arbeiten, verwenden Sie die Klasse `string` (Achtung! Kleinschreibung) aus der Standardbibliothek (Headerdatei `<String>`) und definieren die Variablen ohne das ^-Zeichen. Beachten Sie auch, dass die nachfolgend beschriebenen String-Methoden zur .NET-Klasse gehören. Die `string`-Klasse der Standardbibliothek stellt zum Teil andere Methoden zur Verfügung.

Strings vergleichen

Um festzustellen, ob zwei Strings gleich oder ungleich sind, können Sie die Operatoren == und != verwenden:

```
bool vergleich;
String ^str1 = "Genau";
String ^str2 = "Genua";
vergleich = (str1 == str2);      // false
vergleich = (str1 != str2);      // true
```

Zusätzlich gibt es in der Klasse `String` spezielle Methoden zum Vergleichen.

```
int String::Compare( str1, str2 )
```

```
int String::CompareOrdinal( str1, str2 )
```

Beide Methoden vergleichen `str1` mit `str2`. Der Rückgabewert zeigt an, ob `str1` kleiner, gleich oder größer als `str2` ist. Die Methode `Compare()` berücksichtigt beim Vergleichen allerdings nationale Eigenheiten (abhängig von der landesspezifischen Einstellung des Computers), die Methode `CompareOrdinal()` berücksichtigt solche Eigenheiten nicht.

Tabelle 3.3: Rückgabewerte der Vergleichsmethoden

Rückgabewert	Bedeutung
-1	`str1` kleiner `str2`
0	`str1` gleich `str2`
1	`str1` größer `str2`

```
int vergleich;
String ^str1 = "Genau";
String ^str2 = "Genua";
vergleich = String::Compare( str1, str2 );      // -1
```

Strings vergleichen

Strings werden lexikografisch verglichen, d.h., der Compiler geht die beiden Strings von vorne nach hinten durch und vergleicht Zeichen für Zeichen, bis er auf das erste Zeichen trifft, das in beiden Strings unterschiedlich ist. Dann stellt er fest, welches dieser Zeichen im Alphabet zuerst kommt. Der String, zu dem dieses Zeichen gehört, ist der lexikografisch kleinere der beiden verglichenen Strings.

Sind die beiden Strings bis zum Ende eines Strings gleich, entscheidet die String-Länge. Ist auch diese gleich, sind die beiden Strings gleich.

Weitere Methoden der String-Klasse

Die Klasse `String` stellt noch eine Reihe weiterer interessanter Methoden und Eigenschaften zur Verfügung. Einige dieser Methoden sind sogenannte statische Methoden, die über den Namen der Klasse und den `::`-Operator aufgerufen werden, andere sind nicht-statische Methoden, die nur über ein Objekt und mittels des Operators `->` aufgerufen werden.

Tabelle 3.4: Interessante Methoden der Klasse String

Methode/Eigenschaft	Beschreibung
int Length	Die Länge des Strings (sprich die Anzahl Zeichen im String).

String String::Copy(str)	Kopiert den String `str` und liefert die Kopie als Ergebniswert zurück.
	`String ^str1 = "Hallo Programmierer!";`
	`String ^str2;`
	`str2 = String::Copy(str1);`

int IndexOf(str2)	Sucht nach dem ersten Vorkommen von `str2` in `str1` und liefert die gefundene Position als `int`-Wert zurück bzw. `-1`, falls der String `str2` nicht in `str1` vorkommt.
	`String ^str = "123456";`
	`int pos;`
	`pos = str->IndexOf("3");  // pos = 2`

int Insert(startPos, str2)	Fügt den String `str2` an der Position `startPos` in den String `str1` ein.
	`String ^str = "123456";`
	`str = str->Insert(3,"abcd");`
	`// str enthält jetzt "123abcd456"`

String Remove (startPos, anzahl)	Löscht ab der Position `startPos` genau `anzahl` Zeichen aus dem String.
	`String ^str = "123abcd456";`
	`str = str->Remove(3,4);  // str = "123456"`
	Achten Sie darauf, nicht über das Ende des Strings hinaus Zeichen zu löschen.

Tabelle 3.4: Interessante Methoden der Klasse String (Forts.)

Methode/Eigenschaft	Beschreibung

String Replace (alterTeilstring, neuerTeilstring)	Ersetzt alle Vorkommen eines Teilstrings durch einen anderen Teilstring und liefert das Ergebnis als Kopie zurück.
	`String ^str = "Hallo XXXXXXXX!";`
	`str = str->Replace('X', '_');`

String Substring (pos, laenge)	Liefert einen Teil des Strings als Kopie zurück. Der Teilstring beginnt bei Position `pos` und geht bis zum Ende des Strings oder hat `laenge` Zeichen.
	`String ^str1 = "Hallo Programmierer!";`
	`String ^str2;`
	
	`str2 = str1->Substring(6);`
	
	`// str2 enthält jetzt "Programmierer"`

String ToLower() String ToUpper()	Methoden zur Umwandlung in Klein- oder Großbuchstaben. Der umgewandelte String wird als Ergebnis zurückgeliefert.
	`String ^str = "Hallo Programmierer!";`
	`str = str->ToUpper();`
	
	`// str enthält "HALLO PROGRAMMIERER!"`

String Trim()	Entfernt sogenannten »Whitespace« vom Anfang und Ende eines Strings und liefert das Ergebnis als neuen String zurück.
	`String ^str = "\n\t  Hallo Programmierer! \n\t";`
	
	`str = str->Trim();`
	`Console::WriteLine(str);`
	Ausgabe
	`Hallo Programmierer!`
	Hinweis: Als *Whitespace* bezeichnet man Zeichen, die im Ausdruck nur als Leerraum zu sehen sind. Dies sind vor allem Leerzeichen, Tabulator und Zeilenumbruch.

Methode/Eigenschaft	Beschreibung

String[] Split(trennzeichen)	Diese Methode arbeitet mit Arrays von Strings und einzelnen Zeichen. Arrays werden wir zwar erst weiter unten in Abschnitt 3.6 besprechen, doch die `String`-Methode `Split()` ist zu interessant, um sie hier zu übergehen. Mit ihrer Hilfe kann man nämlich auf elegante Weise einen String in Teilstrings zerlegen. Sie müssen der Methode dazu nur angeben, welches Zeichen oder welche Zeichen als Trennzeichen zu interpretieren sind.
	
	Der folgende Codeauszug zerlegt einen Satz in einzelne Wörter. Die hier verwendeten Techniken zur Definition und zum Durchlaufen von Arrays werden später besprochen.
	
	`String ^str = "Zerlege mich in Wörter!";`
	`array<String^> ^woerter;`
	`array<Char> ^trennzeichen = {' '};`
	
	`woerter = str->Split(trennzeichen);`
	
	`for each (String ^s in woerter)`
	`{`
	`  Console::WriteLine( s );`
	`}`
	
	Ausgabe
	
	`Zerlege`
	`mich`
	`in`
	`Wörter!`

Tabelle 3.4: Interessante Methoden der Klasse String (Forts.)

3.2.6 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.

Betrachten wir dazu noch einmal das Programm *Variablen.cpp* aus Listing 3.3:

```
using namespace System;

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

    ergebnis = ersteZahl - zweiteZahl;

    Console::WriteLine("8754 - 398  = " + ergebnis);

    return 0;
}
```

In der letzten Zeile wird der Wert der `int`-Variablen `ergebnis` auf die Konsole ausgegeben. Auf die Konsole kann man aber nur Strings ausgeben! Des Rätsels Lösung ist, dass die Methode `Console::WriteLine()` den Wert von `ergebnis`, in unserem Beispiel die Zahl 8356, in den String "8356" umwandelt. Dieser wird dann von der Methode ausgegeben.

Was die Methode `Console::WriteLine()` kann, würden wir auch gerne können. Die Frage ist nur wie? Wenn Sie nämlich versuchen, eine Zahl an eine `String`-Variable zuzuweisen:

```
String ^str;
str  = 8754;
```

ernten Sie vom Compiler lediglich eine Fehlermeldung der Form

'int' kann nicht in 'System::String ^' konvertiert werden.

Was ist zu tun?

Die Lösung bringt eine spezielle .NET-Klasse namens `Convert`. Diese definiert eine Reihe von Methoden, mit denen man die Werte elementarer Datentypen in andere elementare Datentypen umwandeln kann.

Tabelle 3.5: Auswahl der wichtigsten Umwandlungsmethoden

Methode	Umwandlung in
`ToBoolean()`	`bool`
`ToByte()`	`byte`
`ToChar()`	`char`
`ToDecimal()`	`decimal`
`ToDouble()`	`double`

Methode	Umwandlung in
`ToInt16()`	`short`
`ToInt32()`	`int`
`ToInt64()`	`long`
`ToString()`	`String`

Tabelle 3.5: Auswahl der wichtigsten Umwandlungsmethoden (Forts.)

Zusätzlich können elementare Datentypen problemlos an die zugehörigen .NET-Typen zugewiesen werden.

```
double zahl1;
Double zahl2 = zahl1;
```

Und alle .NET-Klassen besitzen eine Methode `ToString()`, die Stringdarstellungen ihrer Objekte zurückliefert. In Kombination mit obiger Typumwandlung können Sie auf diese Weise Konstanten und die Werte von Variablen elementarer C++-Datentypen in Strings umwandeln:

```
int    zahl  = 8754;
String ^str;

str  = (8754).ToString();
str  = zahl.ToString();
```

Ja, und dann gibt es noch die Typumwandlungen, die jeder C++-Compiler automatisch vornimmt. Hierzu gehört insbesondere die Umwandlung zwischen elementaren Datentypen, wenn der Zieltyp einen größeren Wertebereich als der Quelltyp haben muss.

Doch Vorsicht! Nicht jede mögliche Umwandlung ist auch eine sinnvolle Umwandlung. Sehen wir uns hierzu einige Beispiele an.

Umwandlung von Zahlen in Strings

```
double zahl = -8754.3;
String ^str;

Double temp = zahl;
str = temp.ToString();

Console::WriteLine(str);
```

Was halten Sie von diesem Code? Wird die gewünschte Umwandlung erfolgreich sein? Sie wird! Testen Sie den Code ruhig aus.

Die Umwandlung ist für den Compiler eigentlich gar kein Problem; er braucht nur den Zahlenwert (-8754.3) in die Zeichenfolge (`"-8754.3"`) zu

verwandeln – ganz wie es bei der Übergabe einer Zahlen-Variablen an `Console::WriteLine()` geschieht.

Warum wird hier `temp` ohne das ^-Zeichen definiert und die Methode `ToString()` mit `.` statt mit `->` aufgerufen? Dies liegt daran, dass die .NET-Klasse `Double` kein Verweistyp, sondern ein Werttyp ist (siehe Kasten zu Verweis- und Datentypen in Abschnitt 3.2.5.

Umwandlung von Strings in Zahlen – Teil 1

```
int zahl;
String ^str = "123";

zahl = Convert::ToInt32(str);

Console::WriteLine(zahl);
```

Was halten Sie von diesem Code?

Auch diese Umwandlung ist unkritisch. In dem String steht die korrekte String-Repräsentation einer Zahl, die der Compiler ohne Probleme in einen Ganzzahlenwert umwandeln kann.

Umwandlung von Strings in Zahlen – Teil 2

```
double zahl;
String ^eingabe;

Console::WriteLine("Geben Sie bitte eine Zahl ein: ");
eingabe = Console::ReadLine();

zahl = Convert::ToDouble(eingabe);

Console::WriteLine(zahl);
```

Und wie sieht es hiermit aus?

In diesem Code wird der Anwender aufgefordert, eine Zahl einzugeben. Die Eingabe wird mit Hilfe der Methode `Console::ReadLine()` entgegen genommen und in der `String`-Variablen `eingabe` abgespeichert.

Tastatureingaben sind immer Strings. Die Methode `Console::ReadLine()` wandelt diese nicht automatisch um, sondern liefert die gesamte Eingabezeile unverändert als String zurück.

Danach wird die Eingabe mit Hilfe der Methode `Convert::ToDouble()` in eine Zahl umgewandelt und in `zahl` abgespeichert. Wir verwenden hier

`ToDouble()` statt `ToInt32()`, damit der Anwender sowohl Ganzzahlen als auch Zahlen mit Nachkommastellen eingeben kann.

Die Frage ist nun, ob die Umwandlung der `eingabe` in einen `double`-Wert glücken wird?

Tatsächlich lässt sich diese Frage nicht beantworten, da das Ergebnis der Umwandlung nicht vorhersehbar ist – es hängt davon ab, was der Anwender als Eingabe eintippt.

Tippt der Anwender als Zahl beispielsweise 123 oder -3,54 ein, wird die Umwandlung glücken.

Tippt der Anwender eine Gleitkommazahl in einem von C++ nicht akzeptierten Format oder gar einen Text wie »Meine Zahl lautet 123« ein, wird die Umwandlung nicht glücken. Die Laufzeitumgebung wird eine Fehlermeldung ausgeben und das Programm wird vorzeitig beendet.

Programmabbrüche durch Benutzereingaben vermeiden

Dass Anwender falsche Eingaben an ein Programm schicken (etwa Text statt Zahlen), ist nicht schön, aber leider unvermeidbar. Sie müssen immer damit rechnen, dass ein Anwender, den Sie auffordern, eine Zahl zwischen 1 und 10 einzugeben, »sieben« eintippt.

Mindestens ebenso unschön ist es aber, wenn Ihr Programm bei solchen Gelegenheiten gleich ganz außer Kontrolle gerät und sich sang- und klanglos verabschiedet. Sie können dies verhindern, indem Sie die Typumwandlung in Ausnahmebehandlungscode einfassen, siehe Kapitel 4.3.

Umwandlung von Gleitkommazahlen in Ganzzahlen

```
double zahl1 = 1234.567;
int zahl2;

zahl2 = Convert::ToInt32(zahl1);

Console::WriteLine(zahl2);
```

Neues Spiel, neues Glück! Wie sieht es mit dieser Typumwandlung aus?

Hier wird eine Gleitkommazahl mit einem Nachkommaanteil in eine Ganzzahl ohne Nachkommastellen umgewandelt. Dies ist meist unproblematisch (solange die Gleitkommazahl vom Betrag her nicht so groß ist, dass sie außerhalb des Wertebereichs des Ganzzahltyps liegt). Allerdings wird der Nachkommaanteil bei der Umwandlung auf- oder abgerundet. Wenn Sie dies nicht stört, können Sie eine solche Umwandlung ohne Probleme vornehmen.

Für die Umwandlung zwischen den elementaren Datentypen für Ganz- und Gleitkommazahlen gibt es übrigens noch zwei andere Verfahren zur Typumwandlung.

Wenn der Wertebereich des Quelldatentyps eine Teilmenge des Wertebereichs des Zieldatentyps darstellt, können Sie den Wert direkt zuweisen – der Compiler führt die Typumwandlung automatisch durch (implizite Typumwandlung).

```
short  s_zahl = 110;
int    i_zahl = 220;
double d_zahl = 330;

i_zahl = s_zahl;
d_zahl = i_zahl;
```

Wenn der Wertebereich des Quelldatentyps größer als der Wertebereich des Zieldatentyps ist, können Sie den Wert durch Voranstellung des Zieltyps umwandeln (explizite Typumwandlung).

```
short  s_zahl = 110;
int    i_zahl = 220;
double d_zahl = 330;

i_zahl = (int) d_zahl;
s_zahl = (short) i_zahl;
```


Bei der impliziten oder expliziten Umwandlung eines `double`-Werts in einen `int`-Wert wird nicht gerundet (wie es `Convert` macht). Stattdessen wird der Nachkommaanteil einfach gelöscht.

Die explizite Typumwandlung mit dem `()`-Operator wird in der Programmierung auch als *Casting* bezeichnet.

3.2.7 C++ für Pedanten

Auch wenn Ihnen die Syntax von C++ einerseits viele Möglichkeiten offen lässt, ist sie andererseits doch recht starr vorgegeben und der Compiler wacht penibel darüber, dass Sie sich an die korrekte Syntax halten.

Wenn Sie es sich also nicht mit dem Compiler verderben wollen, sollten Sie insbesondere auf folgende Punkte achten:

- Alle *Anweisungen* (also Zuweisungen, Methodenaufrufe und Variablendefinitionen) müssen mit einem Semikolon abgeschlossen werden.

  ```
  krummeZahl = 47.11;
  ```

- C++ unterscheidet streng zwischen *Groß- und Kleinschreibung*. Wenn Sie also eine Variable namens `krummeZahl` definiert haben, dann müssen Sie auch `krummeZahl` schreiben, wenn Sie auf die Variable zugreifen wollen, und nicht `krummezahl`, `KrummeZahl` oder `KRUMMEZAHL`.

Und natürlich gibt es auch Regeln für die Einführung von Bezeichnern, also beispielsweise Variablennamen. Wir wollen uns an dieser Stelle aber nicht mit den endlosen Vorschriften befassen, welche die Konstruktion von Variablennamen regeln. Merken Sie sich einfach Folgendes:

- *Variablennamen* können beliebig lang sein, müssen mit einem Buchstaben oder einem Unterstrich '_' beginnen und dürfen nicht identisch mit einem Schlüsselwort der Sprache sein.

Sie dürfen also Ihre Variable nicht `class` nennen, da dies ein reserviertes Wort, ein sogenanntes *Schlüsselwort*, ist. Im Anhang zur C++-Syntax finden Sie eine Liste der C++-Schlüsselwörter.

Der Visual C++-Compiler ist bezüglich der Vergabe von Variablennamen und anderen Bezeichnern nicht so streng wie andere C++-Compiler. Er erlaubt in den Bezeichnern neben Buchstaben und Ziffern auch fast alle Unicode-Zeichen. Insbesondere sind Umlaute erlaubt. Wenn Sie also eine Variable `Begrüßung` nennen wollen, brauchen Sie sie nicht als `Begruessung` zu definieren, sondern können ruhig `Begrüßung` schreiben. Ob man allerdings von dieser Option auch Gebrauch machen sollte, ist eine andere Frage. International ist das Englische die Sprache der Programmierer, und auch wenn Sie nicht unbedingt gleich dazu übergehen müssen, Ihren Klassen, Methoden und Variablen englische Namen zu geben, so empfiehlt es sich in Hinblick auf den weltweiten Austausch von Code doch, in Bezeichnern nur den englischen Zeichensatz zu verwenden (also das romanische Alphabet ohne Umlaute).

3.3 Variablen kontra Konstanten

Muss man wirklich erst erwähnen, dass man den Wert einer Variablen ändern kann (indem man ihr einen neuen Wert zuweist), wohingegen der Wert einer Konstanten unveränderlich ist? Wohl nicht. Interessanter ist es schon zu erfahren, wie man mit Konstanten arbeitet. Dazu gibt es zwei Möglichkeiten:

Erstens: Sie tippen die Konstante direkt als Wert ein. Man spricht dann von sogenannten *Literalen*.

```
krummeZahl = 47.11;                    // Zuweisung eines Literals

krummeZahl = ganzeZahl + 47.11;        // Addition des Werts aus der
                                       // Variablen ganzeZahl und
                                       // des Literals 47.11
```

Da mit einem Literal kein Datentyp verbunden ist, muss der Compiler den Datentyp aus der Syntax des Literals ablesen:

Tabelle 3.6: Literale

Datentyp	Literal
`bool`	`true`, `false`
`char`	`'c'`, `'U'`
`char` (Sonderzeichen)	`'\n'`, `'\\'`
`wchar_t` (Unicode)	`'\u1234'`
`String`	`"Dies ist ein String"`
	`L"Dies ist ein Unicode-String"`
`int`	`12`, `-128`
	`0xFF1F` (hexadezimal)
`long`	`12L`, `5000000000`
`float`	`12.4f`, `10e-2f`
`double`	`47.11`, `1e5`

Zweitens: Sie definieren eine Variable mit dem Schlüsselwort `const`:

```
const double krummeZahl = 47.11;
const double PI = 3.14159265358979323846;
```

Da der Wert einer solchen »konstanten Variablen« nach der Definition nicht mehr verändert werden kann, folgt zwangsläufig, dass sie bei der Definition initialisiert werden muss.

3.4 Operatoren

Nachdem wir nun gesehen haben, was Variablen sind, wie man sie definiert und wie ihnen Werte zugewiesen werden, können wir nun endlich damit beginnen, mit ihnen auch etwas Sinnvolles anzufangen. Dazu dienen bei den bisher vorgestellten elementaren Datentypen vor allem sogenannte Operatoren.

Listing 3.5: Programmieren mit Operatoren (aus dem Projekt Operatoren)

```
using namespace System;

int main()
{
    int x, y, z;
    int ergebnis_1, ergebnis_2;

    x = 1;
```

```
    y = 2;
    z = 3;

    ergebnis_1 = x + y * z;             // = 7
    ergebnis_2 = (5 - 3) * z;           // = 6
    Console::WriteLine(ergebnis_1);
    Console::WriteLine(ergebnis_2);

    x = x + z;                          // = 4
    Console::WriteLine(x);
    x += z;                             // = 7
    Console::WriteLine(x);
    x += 1;                             // = 8
    Console::WriteLine(x);
    x++;                                // = 9
    Console::WriteLine(x);

    return 0;
}
```

Das schaut doch ziemlich vertraut aus, oder? Eigentlich genauso, wie man es von algebraischen Gleichungen her kennt. Aber achten Sie bitte auf die letzten Zeilen des Beispiels. Hier sehen wir seltsame Konstruktionen, die wir nun erklären wollen:

```
x = x + z;
```

Diese Anweisung bewirkt, dass der Computer die aktuellen Werte von x und z zusammenaddiert und dann in x speichert, d.h., die Variable x enthält nach Ausführung dieser Zeile als neuen Wert die Summe aus ihrem alten Wert und z.

Da das Hinzuaddieren eines Wertes zum Wert einer Variablen sehr häufig vorkommt, gibt es dafür eine Kurzschreibweise, nämlich:

```
x += z;
```

Dies teilt dem Rechner mit, dass er zum Wert von x den Inhalt von z hinzuaddieren und das Ergebnis wieder in x speichern soll.

Sehr oft möchte man eine numerische Variable hochzählen (*inkrementieren*). C++ kennt auch hierfür einen speziellen Operator: ++.

```
x++;
```

Diese Anweisung erhöht den Wert von x um 1. Äquivalente Anweisungen wären:

```
x = x + 1;  oder x += 1;
```

Aber Programmierer sind schreibfaul und x++ sieht ja auch viel geheimnisvoller aus!

Das oben Gesagte gilt gleichermaßen für die anderen Grundrechenarten (–, *, /) und das Dekrementieren von Variablen (`--`).

3.4.1 Die verschiedenen Operatoren

In C++ gibt es natürlich noch andere Operatoren. Die wichtigsten sind:

Tabelle 3.7: Ausgesuchte Operatoren

Operator	Beschreibung	Beispiel
++ --	Inkrement, Dekrement	Erhöht oder erniedrigt den Wert einer Variablen um 1.
!	logisches NICHT	Negiert den Wahrheitswert einer Aussage (beispielsweise eines Vergleichs). Wird meist in Kontrollstrukturen (siehe Kapitel 4.2) verwendet.
*	Multiplikation	Multiplikation
/	Division	Division
%	Modulo	Liefert den Rest einer ganzzahligen Division: `7 % 3` liefert z.B. 1
– +	Subtraktion, Addition	Subtraktion und Addition
<= < > >=	Vergleich	Zum Vergleich zweier Werte. Die Operatoren liefern `true` oder `false` zurück.
== !=	Vergleich (gleich / ungleich)	Zum Vergleich auf Gleichheit oder Ungleichheit. Die Operatoren liefern `true` oder `false` zurück.
&&	logisches UND	Verknüpft zwei Boolesche Aussagen. Liefert `true`, wenn beide Aussagen `true` sind. `if ( (x < 1) && (y > 1) )`
||	logisches ODER	Verknüpft zwei Boolesche Aussagen. Liefert `true`, wenn eine der beiden Aussagen `true` ist. `if ( (x < 1) || (y > 1) )`

Die Reihenfolge in der Tabelle deutet die *Priorität* der Operatoren bei der Auswertung von Ausdrücken an. Beispielsweise sind * und / höher eingestuft als + und –, was genau der altbekannten Schulregel entspricht »Punktrechnung vor Strichrechnung«.

Ausdrücke

Ein *Ausdruck* ist eine Berechnung aus Variablen, Konstanten und Operatoren, die auf der rechten Seite einer Zuweisung steht.

Wenn Sie sich bei der Reihenfolge nicht ganz sicher sind oder eine bestimmte Reihenfolge der Auswertung erzwingen möchten, können Sie dies durch die Verwendung von Klammern erreichen. Aber auch wenn keine direkte Notwendigkeit zum Setzen von Klammern besteht, können Sie diese verwenden, um eine Berechnung besser lesbar zu machen.

```
z *= ((2*loop)/(2*loop-1)) * ((2*loop)/(2*loop+1));
```

Falls Sie mehr Mathematik benötigen, als die bisher genannten Operatoren bieten, dann seien Sie getröstet. Es gibt noch eine Reihe von mathematischen Methoden für alle gängigen Probleme wie Sinus, Kosinus, Quadratwurzel usw. Hier eine Aufstellung der wichtigsten:

Methode	Berechnung
`double Math::Acos(double x)`	Arkuskosinus von x
`double Math::Asin(double x)`	Arkussinus von x
`double Math::Atan(double x)`	Arkustangens von x
`double Math::Cos(double x)`	Kosinus von x
`double Math::Exp(double x)`	e hoch x
`double Math::Log(double x)`	Logarithmus zur Basis e von x
`double Math::Pow(double x, double y)`	Die Potenz x^y
`double Math::Sin(double x)`	Sinus von x
`double Math::Sqrt(double x)`	Quadratwurzel aus x
`double Math::Tan(double x)`	Tangens von x

Tabelle 3.8: Methoden für mathematische Funktionen

Die Angabe von beispielsweise `double Math::Tan(double x)` bedeutet dabei, dass als Argument ein `double`-Wert erwartet und als Ergebnis ein `double`-Wert zurückgeliefert wird.

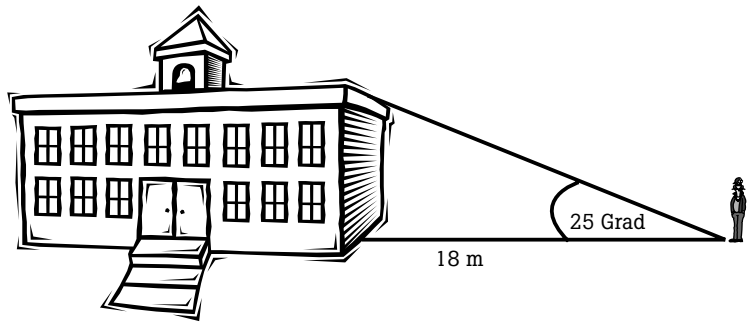
Der Einsatz erfolgt genau, wie Sie es von der Schule her noch kennen, z.B.

```
double y;
double z = 100;
y = Math::Sqrt(z);
```

3.4.2 Verwendung der trigonometrische Methoden

Wenn Sie mit den trigonometrischen Funktionen arbeiten, müssen Sie beachten, dass diese Methoden als Parameter stets Werte in Bogenmaß (Radiant) erwarten. Beim Bogenmaß wird der Winkel nicht in Grad, sondern als Länge des Bogens angegeben, den der Winkel aus dem Einheitskreis (Gesamtumfang 2π) ausschneidet: 1 rad $= 360^{\circ}/2\pi$; $1^{\circ} = 2\pi$ /360 rad.

Abb. 3.5: Höhe eines Hauses berechnen



Nehmen wir an, Sie wollen die Höhe eines Bungalows ermitteln. Dazu stellen Sie sich in einiger Entfernung vor das Haus und markieren diesen Punkt am Boden. Von diesem Punkt aus peilen Sie die Dachkante an und messen den Winkel zwischen der Peillinie zur Dachkante und dem Boden, sagen wir 25 Grad. Jetzt müssen Sie noch am Boden die Entfernung zum Haus messen, sagen wir 18 Meter. Nach der Formel

```
h = distanz * tan a
```

können Sie jetzt die Höhe berechnen.

Listing 3.6: Einsatz trigonometrischer Methoden (aus Haushöhe)

```
using namespace System;

int main()
{
    double winkel;   // Winkel Peillinie - Boden
                     // (in Grad)
    double distanz;  // Entfernung
                     // Peilpunkt - Hauswand
    double hoehe;

    // Daten einlesen
    Console::Write("Winkel in Grad eingeben: ");
    winkel = Convert::ToDouble(Console::ReadLine());

    Console::Write("Distanz in Meter eingeben: ");
    distanz = Convert::ToDouble(Console::ReadLine());
```

```
    // Höhe berechnen und ausgeben
    winkel = winkel * 2 * Math::PI / 360;
    // Umrechnung in Bogenmaß

    hoehe = distanz * Math::Tan(winkel);

    Console::WriteLine();
    Console::WriteLine("Das Haus ist " + hoehe +
                       " Meter hoch.");
    Console::WriteLine();
    Console::WriteLine();

    return 0;
}
```

Abb. 3.6: Ausgabe der berechneten Haushöhe

Sehen wir uns den Quelltext ein wenig genauer an.

Zuerst werden die Werte für den Winkel und die Entfernung vom Betrachter zur Hauswand mit Hilfe der Methode `Console::ReadLine()` eingelesen und in `double`-Werte umgewandelt. Danach beginnt die eigentliche Berechnung.

Da der Winkel in Grad und nicht im Bogenmaß eingelesen wurde, müssen wir den Wert zuerst ins Bogenmaß umrechnen. Dies leistet die Formel:

```
winkel_bogenmaß = winkel_grad * 2 * Math::Pi / 360;
```

Pi steht dabei für die trigonometrische Konstante π, die den Wert 3.141592653... hat. Glücklicherweise müssen wir den Wert der Konstanten nicht im Kopf haben, wir müssen ihn auch nicht abtippen, wir können einfach die Konstante `PI` der Klasse `Math` verwenden:

```
winkel = winkel * 2 * Math::PI / 360;       // Umrechnung
```

Beachten Sie, dass wir hier keine zweite Variable für den Winkel in Bogenmaß einrichten. Wir ersetzen einfach den Grad-Wert durch den Bogenmaß-Wert.

Jetzt können wir den Winkel an die Methode `Tan()` übergeben und die Haushöhe berechnen:

```
hoehe = distanz * Math::Tan(winkel);
```

Das Ergebnis wird schließlich auf die Konsole ausgegeben.

3.4.3 Division

Zum Abschluss noch ein kleines Lehrstück zur Verwendung des Divisionsoperators. In dem folgenden Programm soll ein einfacher Bruch berechnet werden. So einfach und doch ein Stolperstein für viele Programmierer.

```
int main()
{
    int x, y;
    double z1, z2;

    x = 3;
    y = 4;
    z1 = 3 / 4;
    z2 = x / y;
    Console::WriteLine(z1);
    Console::WriteLine(z2);
}
```

Was glauben Sie, welche Werte `z1` und `z2` haben? Bestimmt nicht 0,75, wie man leichtfertig annehmen könnte. Beide sind 0! Wie kommt denn das?

Denken Sie an die Mentalität des Compilers! Er wertet Schritt für Schritt und streng nach Vorschrift die Ausdrücke aus.

Nehmen wir zum Beispiel die Zuweisung `z1 = 3/4;`. Zuerst führt der Compiler die Division 3/4 aus. Da beide beteiligten Operanden ganzzahlig sind, wird nach einer »internen Dienstanweisung« auch das Ergebnis 0.75 in einen ganzzahligen Wert konvertiert, d.h., der Nachkommateil fällt weg und es bleibt eine Null übrig. Nun erst erfolgt die Zuweisung an die `double`-Variable `z1`. Pflichtbewusst wird daher die `int`-0 in eine `double`-0.0 konvertiert und an `z1` zugewiesen. Analoges passiert bei `z2 = x/y`.

Was kann man nun tun, um das gewünschte Ergebnis zu erhalten?

Eine weitere »interne Dienstvorschrift« sagt dem Compiler, dass alle Operanden eines Ausdrucks den gleichen Datentyp haben müssen, und zwar den »größten«, der auftaucht. Es reicht also, wenn wir einen Operanden explizit umwandeln lassen:

```
z1 = (double) 3 / 4;
```

```
z2 = (double) x/y;
```

Das Voranstellen des gewünschten Datentyps in Klammern veranlasst den Compiler, aus der ganzzahligen 3 eine `double`-3.0 zu machen. Dadurch greift beim nachfolgenden Auswerten der Division die besagte Regel, dass alle Operanden den größten auftretenden Typ haben müssen. Der Compiler wandelt daher auch die 4 in 4.0 um und wir haben eine reine `double`-Division `3.0 / 4.0` vorliegen. Das Ergebnis ist daher auch ein `double`-Wert und `z1` und `z2` erhalten beide den korrekten Wert 0.75.

Bei Zahlenkonstanten wie 3/4 kann man auch gleich eine `double`-Zahl schreiben, also `z1 = 3.0/4.0;`.

3.5 Objekte und Klassen

Neben den beschriebenen elementaren Datentypen kennt C++ auch noch verschiedene komplexe Datentypen, allen voran die Klassen.

3.5.1 C++ für Philosophen

Bevor wir uns konkret ansehen, wie man eigene Klassen erstellt und bereits vordefinierte Klassen in seinen Programmen verwendet, wollen wir einen kurzen Blick auf die Philosophie werfen, die hinter dem Schlagwort *Objektorientierung* steckt, denn OOP (objektorientierte Programmierung) steht mehr für eine spezielle Sichtweise als eine ganz neue Programmiertechnik.

Zäumen wir das Pferd von hinten auf und stellen wir uns zunächst die Frage: Wie sieht denn die nicht objektorientierte Programmierung aus?

Nun, man definiert die notwendigen Variablen ähnlich wie in den kleinen Beispielen von vorhin, und dann setzt man die Anweisungen auf, die mit diesen Variablen arbeiten. Fast alle Programmiersprachen bieten dabei die Möglichkeit, Anweisungen in sogenannten Methoden oder Funktionen zu bündeln und auszulagern. Der Programmierer hat dann die Möglichkeit, seinen Code in mehrere Methoden aufzuteilen, die jede eine bestimmte Aufgabe erfüllen (beispielsweise das Einlesen von Daten aus einer Datei, die Berechnung einer mathematischen Methode, die Ausgabe des Ergebnisses auf dem Bildschirm). Damit diese Methoden zusammenarbeiten können, tauschen sie auf verschiedenen Wegen Variablen und Variablenwerte aus.

Bei diesem Modell haben wir auf der einen Seite die Daten (abgespeichert in Variablen) und auf der anderen Seite die Methoden, die mit Daten arbeiten. Dabei sind beide Seiten prinzipiell vollkommen unabhängig voneinander. Welche Beziehung zwischen den einzelnen Methoden einerseits und den Methoden und den Daten andererseits besteht, wird erst klar, wenn

man versucht nachzuvollziehen, wie die Methoden bei Ausführung des Programms Daten austauschen.

Die Erfahrungen mit diesem Modell haben gezeigt, dass bei Programmprojekten, die etwas größer werden, sich sehr leicht Fehler einschleichen: Da verändert eine Methode A nebenbei eine Variable, die später eine Methode B an ganz anderer Stelle im Programm zum Absturz bringt. Die Fehlersuche dauert dann entsprechend lange, weil die Zusammenarbeit von Daten und Methoden kaum nachzuvollziehen ist! Ferner tendieren solche Programme dazu, sehr chaotisch zu sein. Eine Wartung (Modifizierung, Erweiterung) zu einem späteren Zeitpunkt ist oft ein kühnes Unterfangen, vor allem, wenn es nicht mehr derselbe Programmierer ist, der nun verzweifelt zwischen Hunderten von Methoden umherirrt und versucht, die Zusammenhänge und Wirkungsweise zu verstehen.

Schlaue Köpfe kamen daher auf die Idee, eine ganz andere Sichtweise anzunehmen und diese in der Programmiersprache umzusetzen. Ausgangspunkt war dabei die Vorstellung, dass bestimmte Daten und die Methoden, die mit diesen Daten arbeiten, untrennbar zusammengehören. Eine solche Einheit von logisch zusammengehörigen Daten (Variablen) und Methoden bildet ein Objekt. Der Datentyp, der die gleichzeitige Definition von Datenelementen und Methoden erlaubt, ist die Klasse, angezeigt durch das Schlüsselwort `class`.

Objekte und alte Datentypen

Im Grunde ist dies gar nicht so neu. Denken Sie nur an die einfachen Datentypen und die Operatoren. Stellen Sie sich eine `int`-Variable einfach als ein Objekt mit einem einzigen Datenelement, eben der `int`-Variablen, vor. Die Methoden, die mit diesem Objekt verbunden sind, sind dann die Operatoren, die auf `int`-Variablen angewendet werden können (Addition, Subtraktion, Vergleiche etc.). Der Vorteil der Klassen liegt allerdings darin, dass in einer Klasse mehrere Datenelemente vereinigt werden können und dass Sie in Form der Methoden der Klasse selbst festlegen können, welche Operationen auf den Variablen der Klasse erlaubt sind.

3.5.2 Klassen definieren

Der erste Schritt bei der objektorientierten Programmerstellung ist die Zerlegung des Problems in geeignete Objekte und die Festlegung der Merkmale und Verhaltensweisen, sprich der Felder (Datenelemente) und der Methoden, die diese Objekte haben sollten. Dies ist zum Teil sehr schwierig und braucht oft viel Erfahrung, damit durch sinnvolles Bestimmen der Programmobjekte auch die Vorteile der Objektorientiertheit zum Tragen kommen können!

Überlegen wir uns gleich eine kleine Aufgabe. Angenommen Sie sollen für Ihre Firma ein Programm zur Verwaltung der Mitarbeiter schreiben. Wie

könnte eine Aufteilung in Objekte aussehen? Welche Felder und Methoden sind erforderlich?

Eine nahe liegende Lösung ist, die Mitarbeiter als die Objekte anzusehen. Schaffen wir uns also den Prototyp eines Mitarbeiters und implementieren wir diesen in Form der Klasse `Mitarbeiter`.

1. Rufen Sie den Menübefehl Datei/Neu/Projekt auf, wählen Sie in dem Dialogfeld als Vorlage CLR-Konsolenanwendung, geben Sie als Programmnamen *Mitarbeiterverwaltung* ein, wählen Sie einen Speicherort und klicken Sie auf OK.
2. Fügen Sie dem Projekt eine weitere *.cpp*-Quelltextdatei für die neue Klasse hinzu. (Hinweis: Grundsätzlich sollte jede Klasse in einer eigenen Quelltextdatei definiert werden.)

   Um dem Projekt eine Quelltextdatei für eine Klasse hinzuzufügen, rufen Sie im Menü Projekt den Befehl Klasse hinzufügen auf. Wählen Sie im gleichnamigen Dialogfeld unter der Kategorie C++ die Vorlage C++-Klasse aus und klicken Sie auf Hinzufügen.

   Es erscheint der Klassenassistent. Geben Sie in seinem Dialog als Namen für die Klasse *Mitarbeiter* ein. Das Feld für die Basisklasse lassen Sie frei. Kontrollieren Sie, ob für den Zugriff public ausgewählt und von den Kontrollkästchen nur die Verwaltet-Option gesetzt ist. Klicken Sie auf Fertig stellen.

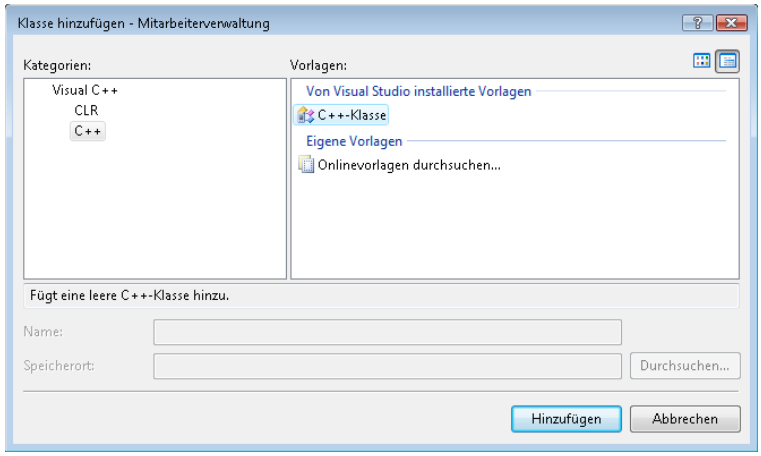


Abb. 3.7: Hinzufügen einer neuen Quelltextdatei für eine Klasse

Visual C++ legt daraufhin eine Quelltextdatei (*Mitarbeiter.cpp*) und eine Headerdatei (*Mitarbeiter.h*) für die neue Klasse an. Auf diese beiden Dateien verteilt sich der C++-Code der Klassendefinition.

Die Headerdatei

Die Headerdatei enthält die `class`-Definition, die den neuen Klassentyp einführt, mit einem Namen verbindet (`Mitarbeiter`) und beschreibt, aus welchen Elementen die Klasse aufgebaut ist. Das von Visual C++ vorgegebene Gerüst deklariert allerdings nur ein einziges Element: den Konstruktor.

Listing 3.7: Mitarbeiter.h

```
ref class Mitarbeiter
{
public:
    Mitarbeiter(void);
};
```

Visual C++ hätte die Klasse auch einfacher definieren können, beispielsweise als

```
class Mitarbeiter                  // definiert Klasse Mitarbeiter
{
    Mitarbeiter();                 // deklariert Konstruktor
};
```

Es hat aber natürlich Gründe, dass Visual C++ die zusätzlichen »Dekorationen« gesetzt hat.

- `void`

  Das Schlüsselwort `void` in den runden Klammern des `Mitarbeiter`-Konstruktors soll dem Compiler anzeigen, dass der Konstruktor keine Parameter definiert.

  Die Verwendung von `void` ist in diesem Kontext sehr fürsorglich, im Grunde aber unnötig. Wenn Sie die runden Klammern leer lassen, weiß der Compiler auch, dass es keine Parameter gibt.

- `public:`

  Klassen können selbst festlegen, welche ihrer Elemente von wem verwendet werden dürfen. Gesteuert wird der Zugriff auf die Elemente dabei über eine spezielle Gruppe von Schlüsselwörtern: den Zugriffsmodifizierern, die wir uns in Kapitel 5.3.3 noch näher anschauen werden.

  Im Moment soll uns genügen, dass `public`-Elemente frei verfügbar sind, während `private`-Elemente nur innerhalb der Klassendefinition verwendet werden können. Die Zugriffsmodifizierer gelten immer für alle nachfolgend deklarierten Klassenelemente. Klassenelemente, die vor dem ersten Zugriffsmodifizierer deklariert werden, sind automatisch `private`. Es ist also sehr wichtig, dass der Konstruktor als `public` deklariert wird, sonst könnte er später gar nicht zur Erzeugung von Objekten der Klasse verwendet werden.

- `ref`

  Hier haben wir es wieder mit einem .NET-spezifischen Schlüsselwort zu tun. Visual C++ hat es vergeben, weil wir im Klassenassistenten die Option VERWALTET gesetzt haben. Damit haben wir uns dafür entschieden, dass die Objekte unserer Klasse – wie die Objekte der .NET-Klassen – vom .NET Framework verwaltet werden. D.h., das .NET Framework legt die Objekte im Speicher an und gibt diesen Speicher automatisch wieder frei, wenn die Objekte nicht mehr benötigt werden. Die automatische Speicherverwaltung ist eine so große Hilfe, dass wir nicht auf sie verzichten möchten. Daher definieren wir unsere Klasse als verwaltet (Schlüsselwort `ref`) und nehmen dafür in Kauf, dass wir die Variablen unserer Klasse wie die Variablen der .NET-Klassen mit dem ^-Symbol definieren müssen.

Die Quelltextdatei

Werfen wir nun einen Blick in die *.cpp*-Quelltextdatei. In ihr steht die Implementierung des `Mitarbeiter`-Konstruktors – und natürlich die `#include`-Direktive für die Headerdatei der Klassendefinition.

```
#include "Mitarbeiter.h"

Mitarbeiter::Mitarbeiter(void)
{
}
```

In Kürze werden wir diese Konstruktordefinition ausbauen und mit Code füllen. Zuvor aber wollen wir uns Gedanken darüber machen, warum Visual C++ fast die gesamte Klassendefinition in die Headerdatei und nur die Konstruktordefinition in die *.cpp*-Datei schreibt.

Erinnern Sie sich an die Grundregel, dass Sie in einer C++-Datei nur solche Elemente verwenden dürfen, die dem Compiler zuvor per Deklaration bekannt gemacht wurden (siehe Kapitel 1.4). Wenn ja, werden Sie sich auch daran erinnern, dass Bibliotheken aus diesem Grunde Headerdateien zur Verfügung stellen, in denen die in ihnen definierten Elemente (Klassen, Funktionen etc.) deklariert sind.

Wenn Sie für ein Programm eine Klasse definieren, damit Sie in Ihrem Programm Objekte der Klassen erzeugen und nutzen können, ist dies fast so, als würden Sie eine kleine Bibliothek aufbauen. Sie können die Klasse in den Quelltextdateien des Programms nur nutzen, wenn Sie die Klassendeklaration in die Quelltextdateien einfügen. Und dies geschieht am bequemsten, wenn Sie eine Headerdatei für die Klasse zur Verfügung haben.

Wie aber sieht eine Klassendeklaration aus? Nun, eigentlich ist die gesamte Klassendefinition eine Deklaration, sofern Sie darauf achten, dass Sie die Methoden in der Klasse nur deklarieren und nicht definieren. Wenn Sie also, wie Visual C++ es vorgemacht hat, in der Klassendefinition nur angeben, wie die Methoden heißen, welche Parameter sie haben und welchen Rückgabetyp sie besitzen, und die Methodenimplementierungen (den Rumpf mit den geschweiften Klammern) in eine eigene Datei auslagern, haben Sie nicht nur die Klasse definiert, sondern gleich auch schon die passende Headerdatei zur Verfügung.[1]

Damit genug der Analyse. Schauen wir uns an, wie wir die von Visual C++ angelegte Klasse ausbauen können.

Felder von Klassen

Was brauchen wir, um einen Mitarbeiter zu beschreiben? Na klar, einen Namen und Vornamen wird er wohl haben. Und ein Gehalt wird er wohl auch beziehen. Erweitern wir also die Klasse um diese Felder in Form von geeigneten Variablen:

Listing 3.8: Felderdeklaration (aus Mitarbeiter.h)

```
using namespace System;

ref class Mitarbeiter
{
    String ^name;
    String ^vorname;
    int    gehalt;          // Monatsgehalt

public:

    Mitarbeiter(void);
};
```

Instanzvariablen

Die hier definierten Felder werden übrigens auch als *Instanzvariablen* bezeichnet, weil Sie später bei der Erzeugung von Objekten (Instanzen) der Klasse an diese weitergegeben werden. Alle Methoden der Klasse können auf diese Variablen zugreifen.

Das Gegenstück zu den Instanzvariablen sind die Klassenvariablen. Dies sind Felder, die mit dem Schlüsselwort `static` definiert werden. Doch dazu später mehr.

1. Ab und an werden Sie C++-Klassendefinitionen sehen, in denen die Methoden innerhalb der Klassendefinition vollständig definiert sind. Der Compiler betrachtet diese Methoden dann als sogenannte Inline-Funktionen, d.h. er erzeugt keine Methodendefinition, sondern ersetzt beim Kompilieren die Methodenaufrufe durch den Code der Methode. Auf diese Weise bleibt die Klassendefinition eine Deklaration.

Methoden von Klassen

Bevor wir daran gehen, unsere erste eigene Methode zu schreiben, wollen wir uns einmal in die Lage eines Compilers oder Sprachentwicklers versetzen und uns überlegen, wie eine Methodendefinition im Allgemeinen aussehen sollte.

- Zuerst braucht die Methode einen Namen, damit sie später aufgerufen werden kann.

  Wie bei den Variablennamen verbirgt sich hinter dem Methodennamen eine Speicheradresse. Diese weist bei den Methoden allerdings nicht auf einen Speicherbereich, in dem ein Wert abgelegt ist, sondern auf den Code der Methode. (Tatsächlich werden beim Aufruf eines Programms ja nicht nur die Daten in den Arbeitsspeicher kopiert, auch der Programmcode, die auszuführenden Maschinenbefehle, wird in den Speicher geladen.)

  Wird eine Methode aufgerufen, sorgt der Compiler dafür, dass der Code der Methode ausgeführt wird. Nach der Abarbeitung der Anweisungen der Methode wird das Programm hinter dem Aufruf der Methode weitergeführt. Damit hätten wir auch schon den zweiten wichtigen Bestandteil unserer Methodendefinition:

- Die Anweisungen, die bei Aufruf der Methode ausgeführt werden sollen.

  Denken Sie dabei daran, dass zusammengehörende Anweisungsblöcke in geschweifte Klammern gefasst werden.

- Letztlich sollte der Compiler schnell erkennen können, dass ein Name eine Methode bezeichnet. Vereinbaren wir daher einfach, dass auf den Methodennamen zwei Klammern folgen sollen.

Unsere Methodendefinition sieht damit folgendermaßen aus:

```
MethodenName()
{
    Anweisungen;
}
```

Bezeichner

Mittlerweile haben wir die dritte Art von *Bezeichnern* (Namen, die der Programmierer einführt und per Definition dem Compiler bekannt gibt) kennen gelernt. Die erste Art von Bezeichnern waren die Variablennamen, die zweite Art von Bezeichnern stellen die Namen dar, die wir den selbst definierten Klassen geben, und die dritte Art von Bezeichnern sind die Methodennamen.

Woher nimmt die Methode die Daten, mit denen sie arbeitet?

- Nun, zum einen ist eine Methode ja Bestandteil einer Klassendefinition. Für die Methode bedeutet dies, dass sie auf alle Felder ihrer Klasse zugreifen kann. (Zur Erinnerung: Dies sind die Variablen, die innerhalb der Klasse aber außerhalb jeder Methode definiert sind.)
- Zum anderen kann eine Methode natürlich auch eigene, sogenannte *lokale Variablen* definieren. Von diesen haben wir in den vorangegangenen Abschnitten bereits eifrig Gebrauch gemacht: Alle in unserer `main()`-Funktion[1] definierten Variablen waren lokale Variablen. Die lokalen Variablen, die in einer Methode definiert werden, sind keine Klassenelemente! Folglich können sie nicht in jeder beliebigen Methode der Klasse benutzt werden, sondern nur innerhalb der Methode, in der sie definiert sind.

Wie aber, wenn zwei Methoden unterschiedlicher Klassen Daten austauschen sollen?

- Für den Austausch über Klassengrenzen hinweg sehen wir sogenannte *Parameter* vor. Dieses sind Variablen, die innerhalb der Klammern der Methodendefinition definiert werden. Bei Aufruf der Methode werden diesen Parametern Werte übergeben (die sogenannten Argumente), die dann innerhalb der Methode wie lokale Variablen benutzt werden können.
- Schließlich soll die Methode auch noch Daten nach außen exportieren. Zu diesem Zweck definiert jede Methode einen Rückgabewert, dessen Datentyp vor den Methodennamen gestellt wird. Später werden wir dann noch sehen, wie mit Hilfe des Schlüsselwortes `return` dieser Rückgabewert an den Aufrufer der Methode zurückgeliefert wird.

Eine vollständige Methodendefinition würde jetzt folgendem Schema folgen:

```
Rückgabetyp MethodenName(Definitionen_der_Parameter)
{
    lokaleVariablen;
    Anweisungen;
}
```

Setzen wir nun den Ausbau unserer eigenen Klasse fort. Nehmen wir an, dass Ihr Chef von Ihrem Programm erwartet, dass es folgende Dinge kann:

- die persönlichen Daten eines Mitarbeiters ausgeben
- sein Gehalt erhöhen

1. Damit es keine Verwirrung gibt, sei noch einmal erwähnt, dass Methoden nichts anderes als Funktionen sind, die als Teil einer Klasse definiert wurden.

Sie scheinen einen netten Chef zu haben! Auf den Gedanken, das Gehalt zu senken, kommt er gar nicht. Lassen wir ihm keine Chance, es sich anders zu überlegen, und versuchen wir, seinen Anforderungen zu entsprechen.

Da *Persönliche-Daten-ausgeben* und *Gehalt-erhöhen* Aktionen sind, die auf den Daten der Mitarbeiter operieren, definieren wir diese Aktionen als Methoden der Klasse `Mitarbeiter`.

Listing 3.9: Deklaration der Methoden (aus Mitarbeiter.h)

```
using namespace System;

ref class Mitarbeiter
{
public:
    ...

    void GehaltErhoehen(int erhoehung);
    void DatenAusgeben();
};
```

Die Klasse `Mitarbeiter` besitzt nun zwei Methoden mit den Namen `GehaltErhoehen()` und `DatenAusgeben()`, von denen die erste einen `int`-Parameter definiert. Das Schlüsselwort `void`, welches an der Position steht, wo der Compiler die Angabe des Rückgabetyps erwartet, besagt, dass die Methode keinen Ergebniswert zurückliefert, es folglich auch keinen Rückgabetyp gibt.

Was noch fehlt, ist die Implementierung der Methoden – also die Verbindung der Methode mit einem Anweisungsblock, der bei Aufruf der Methode auszuführen ist. Wie in C++ üblich, schreiben wir die Methodenimplementierungen in die *.cpp*-Datei der Klasse.

```
#include "Mitarbeiter.h"
...

void Mitarbeiter::DatenAusgeben()
{
   Console::WriteLine();
   Console::WriteLine("Name    : " + name);
   Console::WriteLine("Vorname : " + vorname);
   Console::WriteLine("Gehalt  : " + gehalt + " Euro");
}

void Mitarbeiter::GehaltErhoehen(int erhoehung)
{
   this->gehalt += erhoehung;
}
```

Die Methodendefinitionen sehen genauso aus wie die Deklarationen in der Headerdatei, nur dass Sie dem Methodennamen den Namen der Klasse voranstellen müssen und natürlich noch den Anweisungsblock (der Block zwischen den geschweiften Klammern) aufsetzen.

Beachten Sie, dass Sie innerhalb der Methoden ohne Einschränkung mit den Feldern der Klasse (hier `name`, `vorname` oder `gehalt`) arbeiten können. Sie können diese Felder, wie im Übrigen auch die anderen Methoden der Klasse, direkt über ihren Namen (siehe `DatenAusgeben()`) oder über das Schlüsselwort `this` (siehe `GehaltErhoehen()`) ansprechen.

Das Schlüsselwort `this` haben Sie bereits in Kapitel 2.2.2 kennen gelernt. Es ist ein Schlüsselwort, welches als Stellvertreter für das aktuelle Objekt steht. Sie können es setzen, um den Quelltext für Programmierer besser lesbar zu machen (man erkennt dann auf einen Blick, hinter welchen Variablennamen Felder stehen und welches nur lokale Variablen oder Parameter sind). Unbedingt nötig ist das Schlüsselwort `this`, wenn Sie Felder der Klasse von gleichnamigen lokalen Variablen oder Parameter der aktuellen Methode unterscheiden müssen. Diese Situation ergibt sich häufig bei der Definition von Konstruktoren.

Konstruktoren von Klassen

Nun zu der Methode, die den gleichen Namen trägt wie die Klasse und keinen Rückgabetyp hat:

```
ref class Mitarbeiter
{
public:
    ...

    Mitarbeiter(void);
};
```

Dies ist eine ganz besondere Methode, nämlich ein *Konstruktor*. Jede Klasse braucht einen oder sogar mehrere Konstruktoren, die beim Erzeugen der Objekte der Klasse behilflich sind. Wir, die Autoren einer Klasse, benutzen den Konstruktor beispielsweise dazu, den Feldern der Klasse sinnvolle Anfangswerte zuzuweisen. Im Falle unserer `Mitarbeiter`-Klasse wäre es schön, wenn wir dem Konstruktor die persönlichen Daten des anzulegenden Mitarbeiters übergeben können, damit der Konstruktor die Daten in den zugehörigen Instanzvariablen speichert.

Zu diesem Zweck erweitern wir die Konstruktordeklaration um drei Parameter für Name, Vorname und Gehalt.

Listing 3.10: Die vollständige Klassendefinition (Mitarbeiter.h)

```
using namespace System;

ref class Mitarbeiter
{
    String ^name;
    String ^vorname;
    int    gehalt;           // Monatsgehalt

public:
    Mitarbeiter(String ^name, String ^vorname, int gehalt);

    void GehaltErhoehen(int erhoehung);
    void DatenAusgeben();
};
```

Im Anweisungsrumpf des Konstruktors weisen wir die Parameterwerte an die Felder zu.

Listing 3.11: Die Konstruktordefinition (aus Mitarbeiter.cpp)

```
#include "Mitarbeiter.h"

Mitarbeiter::Mitarbeiter(String ^name, String ^vorname, int gehalt)
{
    this->name = name;
    this->vorname = vorname;
    this->gehalt = gehalt;
}
...
```

Bitte beachten Sie, dass wir den Instanzvariablen hier unbedingt das Schlüsselwort `this` voranstellen müssen, damit der Compiler die Instanzvariablen von den gleichnamigen Parametern unterscheiden kann. Wir hätten die Parameter natürlich auch `n`, `v` und `g` nennen können, aber so ist der Text besser lesbar.

Jede Klasse braucht einen Konstruktor. Wenn Sie selbst keinen Konstruktor vorsehen, weist der Compiler der Klasse einen Ersatzkonstruktor zu. Der Ersatzkonstruktor führt keinen Code aus und definiert keine Parameter.

Konstruktoren, die keine Parameter definieren, werden auch als *Standardkonstruktoren* bezeichnet. (Der vom Compiler bei Bedarf zugewiesene Ersatzkonstruktor ist also ein Standardkonstruktor.)

Damit ist die `Mitarbeiter`-Klasse fürs Erste vollendet! Das war doch nicht allzu schwer?! Nun wollen wir diese Klasse auch benutzen. Wir nehmen das Grundgerüst für ein C++-Programm, fügen die Klassendefinition von

Mitarbeiter hinzu und erzeugen dann in der main()-Funktion einige Objekte (Instanzen) unserer neuen Klasse.

3.5.3 Mit Klassen programmieren

3. Laden Sie nun die Datei *MitarbeiterVerwaltung.cpp* in den Editor und instanzieren Sie in main() die Klasse Mitarbeiter.

Listing 3.12: Mitarbeiter-Verwaltung.cpp des Projekts Mitarbeiterverwaltung

```
#include "Mitarbeiter.h"

using namespace System;

int main(array<System::String ^> ^args)
{
    Mitarbeiter ^billy =
                    gcnew Mitarbeiter("Gates", "Bill", 3000);
    Mitarbeiter ^stevie =
                    gcnew Mitarbeiter("Jobs", "Steve", 3500);

    // Daten ausgeben
    billy->DatenAusgeben();
    stevie->DatenAusgeben();

    // Gehalt von billy erhöhen
    billy->GehaltErhoehen(500);

    // Kontrolle
    billy->DatenAusgeben();
    stevie->DatenAusgeben();

    return 0;
}
```

4. Kompilieren Sie das Programm und führen Sie es anschließend aus (Strg F5).

Wenn die erste Begeisterung über das funktionierende Programm vorbei ist, können Sie sich wieder setzen und die nachfolgenden Erläuterungen lesen.

Instanzierung mit gcnew

Spannend sind vor allem die ersten Zeilen in der main()-Funktion. Dort werden Instanzen der Klasse Mitarbeiter angelegt:

```
Mitarbeiter ^billy = gcnew Mitarbeiter("Gates","Bill",3000);
```

Was macht der Compiler, wenn er diese Zeile antrifft? Er legt eine neue Variable mit Namen billy an. Das sagt ihm die Seite links von dem Gleichheitszeichen. Die rechte Seite teilt ihm mit, dass er auch noch ein Objekt

(Instanz) der Klasse `Mitarbeiter` erzeugen soll. Dazu wird mit Hilfe des Schlüsselwortes `gcnew` der Konstruktor der Klasse aufgerufen, der drei Parameter erwartet, die wir ihm ordnungsgemäß übergeben.

Die obige Syntax ist übrigens kein reines C++. Dass wir die Variable mit dem ^-Symbol definieren und das Objekt mit dem Schlüsselwort `gcnew` erzeugen, liegt daran, dass wir die Klasse `Mitarbeiter` als verwaltete Klasse gekennzeichnet haben (Schlüsselwort `ref` in Klassendefinition). D.h. wir übergeben die Verantwortung für die Speicherverwaltung für die Objekte an das .NET Framework. Für eine nicht verwaltete Klasse würden wir die folgende Syntax wählen

```
Mitarbeiter *billy = new Mitarbeiter("Gates","Bill",3000);
```

und müssten das Objekt nach Gebrauch selbst löschen

```
delete billy;
```


Instanzen von verwalteten Klassen werden mit dem Operator `gcnew` erzeugt.

Variablen von verwalteten Klassentypen speichern Referenzen

Damit Sie später nicht den Durchblick verlieren, wollen wir an dieser Stelle etwas technischer werden, denn es besteht ein fundamentaler Unterschied zwischen den Variablendefinitionen

```
int ersteZahl = 8754;
```

und

```
Mitarbeiter ^billy = gcnew Mitarbeiter("Gates","Bill",3000);
```

int ersteZahl

Im ersten Fall wird eine `int`-Variable angelegt, d.h., der Compiler ordnet dem Namen `ersteZahl` einen bestimmten Speicherbereich zu. Gleichzeitig initialisieren wir die Variable mit dem Wert 8754, wobei der Wert direkt in dem Speicherbereich der Variablen abgelegt wird (siehe Abbildung 3.8).

Mitarbeiter billy

Im zweiten Fall wird zwar ebenfalls eine Variable angelegt (`billy`), aber in ihr wird nicht einfach ein Wert abgelegt. Stattdessen wird mit Hilfe des `gcnew`-Operators der Konstruktor der Klasse `Mitarbeiter` aufgerufen und eine Instanz der Klasse erzeugt. Die Instanz wird im Speicher angelegt – aber nicht in dem Speicherbereich, der für die Variable `billy` eingerichtet wurde. Tatsächlich existiert die Instanz ganz unabhängig irgendwo im Arbeitsspeicher. Bei der Zuweisung der Instanz an die Variable `billy` wird dann nicht etwa der Inhalt aus der Instanz in den Speicherbereich der

Variablen `billy` kopiert. Nein, stattdessen wird in der Variablen `billy` die *Adresse* des Speicherbereichs der Instanz abgespeichert. Ist dies erst einmal geschehen, sprechen wir wieder einfach von der Instanz `billy` und sehen großzügig darüber hinweg, dass `billy` eigentlich nur eine Variable ist, die eine Speicherzelle bezeichnet, in der ein Verweis (eine *Referenz*) auf die eigentliche Instanz abgespeichert ist.

Noch deutlicher werden die Vorgänge, wenn wir die Instanzbildung in zwei Schritte zerlegen:

```
Mitarbeiter ^billy;

billy = gcnew Mitarbeiter("Gates","Bill",3000);
```

Zuerst wird eine Variable vom Typ `Mitarbeiter` definiert, die zu diesem Zeitpunkt noch keinen gültigen Wert besitzt. In der zweiten Zeile wird mit dem `gcnew`-Operator eine neue Instanz erzeugt und `billy` erhält dann den Verweis auf den Speicherbereich, wo die Instanz der Klasse zu finden ist.

Dies ist eine äquivalente Möglichkeit. Meistens werden Sie in Programmen die kompakte Variante sehen. Sie wissen ja, Programmierer sind schreibfaul und lieben das Kryptische ...

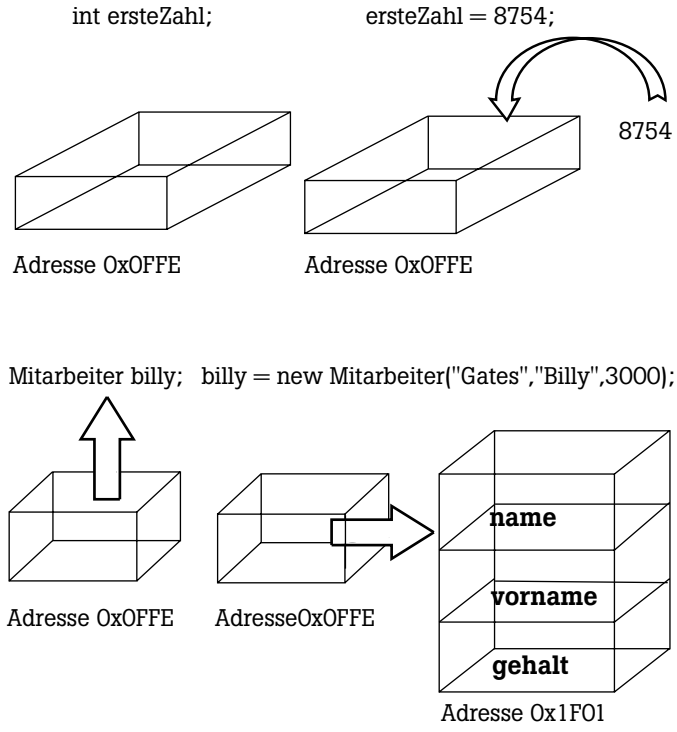


Abb. 3.8: Instanzbildung

In Variablen von verwalteten Klassentypen werden nie die Objekte selbst, sondern immer nur Verweise auf die Objekte gespeichert.

Variablen, die Verweise speichern, werden in traditionellem C++ Zeiger genannt und mit einem Sternchen definiert:


```
int *zeiger;  // Zeiger auf einen int-Wert
```

Die Variablen, in denen wir die Verweise auf unsere verwalteten Objekte speichern, könnte man vielleicht .NET-Zeiger nennen. Informatiker verbinden mit dem Begriff »Zeiger« allerdings meist die Vorstellung, dass der Verweis im Grunde aus einer festen Speicheradresse besteht. Dies ist aber bei den verwalteten Objekten nicht der Fall, weil die Laufzeitumgebung des .NET Frameworks sie unter Umständen verschiebt. Microsoft spricht daher statt von Zeigern von Handles.

Stack, Heap und das .NET-Typensystem

Die C++-Programmierung kennt traditionell zwei, sich ergänzende Formen der Speicherverwaltung. Die erste speichert Variablen auf dem Stack, die zweite auf dem Heap. Mit dem .NET Framework kommt eine dritte Form hinzu: der sogenannte verwaltete Heap.

Das Standardverfahren sieht so aus, dass der Speicher für die Variablen auf dem Stack reserviert wird. Der Stack wird in einzelne Abschnitte (Frames) unterteilt, die dynamisch erzeugt und wieder aufgelöst werden. Für die Variablen in einem Stackframe bedeutet dies, dass sie automatisch aufgelöst werden, wenn ihr Frame gelöscht wird. (Beispielsweise erzeugt C++ für jeden Methodenaufruf einen eigenen Stackframe, speichert in diesem die lokalen Variablen der Methode und löst den Frame auf, wenn die Methode beendet ist.) Der Stack-Speicher wird also automatisch verwaltet, hat aber den Nachteil, dass die Variablen nur so lange existieren, wie ihr Stackframe.

Das Pendant zum Stack ist der Heap. Auf dem Heap können nur Objekte von Klassen gespeichert werden. Ob ein Objekt auf dem Stack oder dem Heap landet, entscheidet der Programmierer dadurch, dass er bei der Objekterzeugung (Konstruktoraufruf) das Schlüsselwort `new` verwendet (Reservierung auf dem Heap) oder nicht (Reservierung auf dem Stack). Objekte auf dem Heap bleiben so lange bestehen, bis der Programmierer sie explizit auflöst (mit dem Schlüsselwort `delete`) oder das Programm beendet wird. Für den Programmierer ist es sehr reizvoll, Objekte auf dem

Heap zu reservieren. Die Programmierung mit Heap-Objekten ist allerdings auch gefährlich. Versäumt der Programmierer, die Objekte nach Gebrauch zu löschen, läuft er Gefahr den Speicher aufzubrauchen (Speicherleck). Löscht er ein Objekt zu früh und versucht später darauf zuzugreifen, stürzt das Programm ab.

Der ideale Kompromiss zwischen Stack und Heap ist der verwaltete Heap des .NET Frameworks. Der Programmierer kann beliebig Objekte auf dem verwalteten Heap reservieren, die so lange existent bleiben, wie sie benötigt werden (sprich solange es mindestens eine Variable gibt, die auf das Objekt verweist). Anschließend werden sie automatisch vom Garbage Collector der CLR gelöscht. Damit die Objekte einer Klasse auf dem verwalteten Heap angelegt werden, muss die Klasse mit dem Schlüsselwort `ref` definiert sein. Die Objekte müssen mit `gcnew` erzeugt werden.

Mit dieser Hintergrundinformation können wir die Instanzierung unser Klasse `Mitarbeiter` noch besser verstehen. Die Anweisung

```
int main()
{
   Mitarbeiter ^billy = gcnew Mitarbeiter("Gates","Bill",3000);
```

legt eine neue lokale Variable `billy` im Stackframe der `main()`-Funktion an. Danach wird im verwalteten Heap ein Objekt der Klasse `Mitarbeiter` erzeugt und der Verweis auf dieses Objekt wird in der Variablen `billy` gespeichert. Wenn das Ende der `main()`-Funktion erreicht wird, wird die Variable `billy` sofort mit dem Stackframe der `main()`-Funktion aufgelöst. Danach gibt es keinen Verweis mehr auf das `Mitarbeiter`-Objekt im verwalteten Heap. Das .NET Framework erkennt dies und löscht das Objekt.

Zugriff auf Instanzen

Wie erfolgt nun der Zugriff auf die Instanzen `billy` und `stevie`? Nehmen wir beispielsweise die Anweisung `billy->DatenAusgeben()`. Man greift zuerst auf die Instanz zu (über die Variable `billy`) und ruft dann mit Hilfe des Operators `->` die gewünschte Methode auf.

Voraussetzung ist allerdings, dass die aufgerufene Methode in der Klasse als `public` definiert wurde, sonst verweigert der Compiler den Zugriff! (Gleiches gilt für den Zugriff auf Felder.)

```
billy->gehalt = 4000;           // syntaktisch eigentlich korrekt,
                                // ist aber nicht erlaubt, weil
                                // das Klassenelement gehalt nicht
                                // public ist.

billy->GehaltErhoehen(4000);    // so ist´s besser und vom Autor
                                // der Klasse vorgesehen
```

Verfügt die Methode über Parameter, werden diesen in der Klammer Argumente übergeben. Wichtig bei der Parameterübergabe ist vor allem die Reihenfolge. Sie muss identisch sein mit der Reihenfolge in der Definition der Methode. Der Konstruktor der Klasse `Mitarbeiter` muss also immer zuerst zwei Zeichenketten für Name und Vorname erhalten und dann eine Zahl für das Gehalt.

Auf die Elemente von Objekten verwalteter Klasse greifen Sie immer vermittels des Operators `->` zu. Wenn Sie ein Objekt einer nicht verwalteten Klassen mit `new` instanzieren und den Verweis in einer Zeigervariablen (`*name`) speichern, greifen Sie ebenfalls mit `->` auf die Elemente des Objekts zu. Wenn Sie das Objekt nicht mit `new` erzeugen, wird es wie die Werte der elementaren Datentypen direkt in der Variablen gespeichert. Dann erfolgt der Zugriff auf die Elemente mittels des `.`-Punktoperators:

```
Mitarbeiter jim("Jim", "Software", 1000);      // Mitarbeiter sei
                                               // ohne ref
definiert
jim.GehaltErhoehen(1000);
```

Vorsicht, Objektvariable ohne Verweis!

Variablen vom Typ einer Klasse bezeichnen wir auch als *Objektvariablen*. Wenn Sie eine solche Objektvariable definieren, verweist diese anfangs auf kein Objekt. Trotzdem können Sie versuchen, auf Elemente der Klasse zuzugreifen.

```
ref class Demo
{
public:
    void TueWas()
    {
        ...
    }
};

Demo ^obj;
obj->TueWas();     // Zugriff auf eine Methode, obwohl obj
                   // auf kein Objekt verweist!?
```

Syntaktisch ist ein solcher Zugriff korrekt, aber es ist natürlich ein eklatanter Fehler, denn es gibt ja überhaupt kein Objekt, für das die Methode `TueWas()` aufgerufen werden könnte. Ein solcher Code wird in C++ entweder vom Compiler abgefangen, der Sie mit einer Fehlermeldung oder einer Warnung darauf hinweist, dass `obj` auf kein Objekt verweist, oder das Programm stürzt bei der Ausführung des Zugriffs mit einer `NullReferenceException` ab.

Achten Sie also darauf, Ihren Objektvariablen stets ein Objekt zuzuweisen.

Wenn Sie einer Objektvariablen anfangs erst einmal kein Objekt zuweisen möchten oder den Objektverweis in einer Variablen irgendwann löschen möchten, weisen Sie der Objektvariablen die `nullptr`-Referenz zu.

```
Demo ^obj = nullptr;
```


Das Schlüsselwort `nullptr` gehört zu den Microsoft-Erweiterungen. Traditionelles C++ verwendet zur Initialisierung seiner Zeiger den Wert NULL: `NichtVerwalteteKlasse *zeigervariable = NULL;`.

Im nachfolgenden Code können Sie dann, sofern Sie unsicher sind, ob die Variable auf ein Objekt verweist oder die `nullptr`-Referenz enthält, mit einer einfachen `if`-Bedingung[1] prüfen, ob ein Objekt vorhanden ist:

```
if (obj != nullptr)
   obj->TueWas();
```


Die `nullptr`-Zuweisung ist übrigens auch der einzige Weg, um Objekte auf dem verwalteten Heap explizit zu löschen. Einmal erzeugte Objekte bleiben grundsätzlich so lange erhalten, wie das Programm ausgeführt wird. Gibt es allerdings irgendwann keine Verweise auf das Objekt mehr, weil die Variablen mit den Verweisen gelöscht oder auf `nullptr` gesetzt wurden, wird das Objekt von der .NET Framework-Müllabfuhr, der sogenannten Speicherbereinigung (*Garbage Collector*), aufgesammelt und gelöscht.

3.6 Arrays

Nun wäre es ziemlich unpraktisch, wenn wir uns in dem obigen Beispiel für jeden neuen Mitarbeiter, für den wir eine Instanz der Klasse `Mitarbeiter` anlegen, auch einen neuen Variablennamen ausdenken müssten. Wenn die Firma etwas größer ist, dann kommen wir schon in arge Bedrängnis. Aber glücklicherweise gibt es dafür eine Konstruktion, die man Array nennt. Am besten schauen wir uns gleich ein Beispiel für die Definition eines Arrays an.

1. siehe Kapitel 4.2

3.6.1 Arrays definieren

```
int werte[100];
```

Obige Definition erzeugt ein Array mit dem Namen `werte` und 100 Elementen, wobei als Elemente nur `int`-Werte erlaubt sind. Die Elemente eines Arrays gehören also immer dem gleichen Datentyp an. Wenn Sie z.B. sowohl `int`- als auch `double`-Werte in Arrays verwalten möchten, müssen Sie zwei Arrays definieren:

```
int intWerte[100];
double dblWerte[100];
```

Im Falle unseres Mitarbeiterprogramms hilft uns die obige Syntax allerdings nicht weiter, denn in den traditionellen C++-Arrays können keine Verweise auf Objekte von verwalteten Klassen gespeichert werden. Glücklicherweise gibt es aber eine Microsoft-Erweiterung, mit deren Hilfe es möglich ist, auch Arrays von verwalteten Objekten zu erzeugen:

```
array<Mitarbeiter ^> ^personalListe
                                = gcnew array<Mitarbeiter ^>(100);
```

Keine Scheu, so kompliziert wie diese Syntax auf den ersten Blick erscheint, ist sie gar nicht. Sie beginnt mit der Typangabe `array` – was schon einmal ganz angenehm ist, denn so lässt sich die Array-Definition gut im Code erkennen.

Es folgt in spitzen Klammern die Angabe des Datentyps der Elemente. Hier ist ein klein wenig Vorsicht geboten. Wie im Falle unserer Objektvariablen speichern wir in den Array-Elementen nicht die Objekte der Klasse Mitarbeiter direkt, sondern Verweise auf die Objekte. Der Typ der Elemente ist daher nicht `Mitarbeiter`, sondern `Mitarbeiter^`.

Anschließend wird eine Objektvariable für das Array definiert (hier `personalListe`) und das Array mit `gcnew` und der gewünschten Größenangabe (hier 100) erzeugt.

Während die traditionellen C++-Arrays keine Objekte von verwalteten Klasse verwahren können, ist `array` durchaus in der Lage, Elemente beliebiger Datentypen zu verwalten, beispielsweise auch von Werten elementarer Datentypen:

```
array<int> ^werte = gcnew array<int>(3);
```

3.6.2 Auf Array-Elemente zugreifen

Der []-Operator wird in C++ nicht nur bei der Definition der Arrays, sondern auch zum Zugriff auf einzelne Elemente der Arrays verwendet: Sie brauchen lediglich in den eckigen Klammern anzugeben, auf das wievielte Element zugegriffen werden soll. Die Zahl in den Klammern nennt man daher auch Index und die Art des Zugriffs »indizierten Zugriff«.

Vermittels des indizierten Zugriffs können Sie Werte als Elemente im Array ablegen oder umgekehrt die Werte der Elemente abfragen. Achten Sie aber darauf, immer nur die Werte von Array-Elementen abzufragen, denen Sie zuvor auch Werte zugewiesen haben.

Das erste Element eines Arrays hat den Index 0 (in Worten NULL!).

```
werte[0] = 10;                          // Werte als Elemente in
werte[1] = 20;                          // einem Array ablegen
werte[2] = 30;

Console::WriteLine(werte[0]);           // Werte der Array-Elemente
Console::WriteLine(werte[1]);           // abfragen
Console::WriteLine(werte[2]);
```

Der Zugriff auf `array`-Elemente sieht ganz analog aus:

```
personalListe[0] = gcnew Mitarbeiter("Gates","Bill",3000);
personalListe[1] = gcnew Mitarbeiter("Jobs", "Steve", 3500);

personalListe[0]->DatenAusgeben();
```

Sie sehen, Arrays sind ganz einfach zu verwenden. Aber eine Regel müssen Sie sich besonders nachhaltig einprägen:

Der Umstand, dass die Array-Indexierung, wie in etlichen anderen Sprachen auch, mit 0 beginnt, hat schon Tausenden von Programmierern zahllose Stunden an Fehlersuche verursacht. Wieso? Der Grund liegt wohl in der menschlichen Psyche. Wenn Sie wie oben das Array `personalListe` mit 100 Einträgen definiert haben, erfordert es geradezu übermenschliche Kräfte, um Ihrem Gehirn die fixe Idee auszutreiben, dass der Eintrag `personalListe[100]` existiert. Da bei der Definition des Arrays aber die Anzahl an Elementen angegeben wird und der erste Eintrag bei 0 beginnt, ist das letzte gültige Element definitiv `personalListe[99]`.

Der Parameter von main()

Mit unserem neu erworbenen Wissen über Arrays sind wir endlich auch in der Lage, die Definition der `main()`-Funktion in unseren Konsolenanwendungen etwas besser zu verstehen:

```
int main(array<System::String ^> ^args)
```

Wie Sie wissen, stehen in den runden Klammern die Parameter, die diese Methode erwartet. Im Falle von `main()` wird nur ein Parameter definiert: `array<System::String ^> ^args`, also ein Array von `String`-Objekten. In Kapitel 7.5 werden wir noch ein kleines Beispiel dazu sehen, wie man mit Hilfe von `args` Befehlszeilenargumente einliest und innerhalb des Programms verarbeitet.

3.6.3 Arrays initialisieren

Um den Elementen in einem Array Anfangswerte zuzuweisen, gibt es verschiedene Möglichkeiten:

- Sie weisen den einzelnen Elementen nacheinander Werte zu (wozu meist Schleifen, siehe Kapitel 4.2.4, eingesetzt werden)
- Sie initialisieren das Array bei der Definition der Array-Variablen.

Um ein Array direkt bei der Definition zu initialisieren, setzen Sie die Werte für die Array-Elemente als eine durch Kommata getrennte Liste auf und weisen diese der Array-Variablen zu.

```
int werte[3] = {10, 20, 30};
```

Der Compiler erzeugt dann automatisch ein passendes Array und weist dessen Elementen die gewünschten Werte zu. Dies gilt für die traditionellen C++-Arrays ebenso wie für die .NET-Arrays:

```
array<Mitarbeiter^> ^personalListe = {
                            gcnew Mitarbeiter("Gates","Bill",3000),
                            gcnew Mitarbeiter("Jobs", "Steve", 3500)
                            };

personalListe[0]->DatenAusgeben();
personalListe[1]->DatenAusgeben();
```

.NET-Arrays sind Klasseninstanzen

Noch eine letzte Bemerkung zu den .NET-Arrays: Jedes Array, das Sie mit `array` anlegen, ist selbst automatisch eine Instanz der Klasse `Array` (auch so eine von den vielen vordefinierten .NET-Klassen). Es gibt daher auch Methoden und Variablen, auf die Sie über Ihre Array-Variablen zugreifen können. So können Sie die Größe eines Arrays, d.h. die Anzahl der Elemente im Array, beispielsweise über das Klassenelement `Length` abfragen:

```
array<Mitarbeiter^> ^personalListe = gcnew
array<Mitarbeiter^>(100);
personalListe[0] = gcnew Mitarbeiter("Gates","Bill",3000);
personalListe[1] = gcnew Mitarbeiter("Jobs", "Steve", 3500);

// Gibt die definierte Größe aus, also 100
Console::WriteLine("Array-Größe ist " + personalListe->Length);
```

3.7 Die vordefinierte .NET-Klassen

Sie haben nun bereits einige .NET-Klassen kennen gelernt, allen voran natürlich `Console` mit seiner `WriteLine()`-Methode, aber auch Klassen wie `Convert`, `Math` und `Array`.

Wenn Sie sich darüber informieren wollen, welche weiteren .NET-Klassen es gibt, rufen Sie die Visual Studio-Hilfe auf (Befehl Hilfe/Index) und geben Sie als Suchbegriff *System* ein. In dem linksseitigen Listenfeld können Sie dann durch die Auflistung der in `System` und seinen Unter-Namespaces definierten Bibliothekselemente scrollen.

Und wenn Sie sich darüber informieren möchten, welche Elemente eine .NET-Klasse definiert, setzen Sie den Filter der Index-Hilfe auf *.NET Framework* oder *ungefiltert*, tippen Sie in das Suchfeld den Klassennamen und den Zusatz »-Klasse« ein und klicken Sie anschließend auf den Untereintrag alle Member.

3.8 Zusammenfassung

Daten werden in Programmen durch Variablen repräsentiert. Jeder Variablen entspricht ein Speicherbereich, in dem der aktuelle Wert der Variablen abgelegt wird. Variablen müssen definiert werden, um dem Compiler den Namen der Variablen bekannt zu machen. Bei der Definition der Variablen wird auch der Datentyp der Variablen angegeben, der festlegt, welche Wer-

te die Variable aufnehmen kann und welche Operationen auf der Variablen durchgeführt werden können.

Verwaltete Klassen werden mit dem Schlüsselwort `ref` definiert. Objekte (Instanzen) von verwalteten Klassen werden mit dem Operator `gcnew` und durch Aufruf des Konstruktors der Klasse erzeugt.

```
ref class Klassenname
{
};
...
Klassenname ^obj = gcnew Klassenname();
```

Traditionelle, nicht verwaltete Klassen werden ohne `ref` definiert. Ihre Objekte können wahlweise in den Variablen des Stacks oder mit dem Operator `new` auf dem nicht verwalteten Heap erzeugt werden.

```
class Klassenname
{
};
...
Klassenname obj1();                          // Erzeugung in Variable
Klassenname obj2 = new Klassenname();        // Erzeugung auf Heap
```

3.9 Fragen und Antworten

1. Ich rechne in meinem Programm vorwiegend mit kleinen Zahlen. Würden Sie mir raten, die zugehörigen Variablen mit dem Datentyp `short` zu definieren, um Speicherplatz zu sparen?

Nein, ich würde Ihnen sogar abraten, und zwar aus vier Gründen. Erstens gibt es Ärger, wenn Ihre Variablen doch irgendwann größere Werte aufnehmen müssen. Zweitens erschwert dies den Datenaustausch mit den verbliebenen `int`-Variablen, da Zuweisungen von `int` nach `short` eine explizite Typumwandlung erfordern. Drittens werden `int`- oder `long`-Variablen auf den meisten Prozessoren schneller verarbeitet als die kleineren Datentypen. Und viertens ist der eingesparte Speicherplatz vermutlich nicht annähernd groß genug, um den Aufwand zu rechtfertigen.

2. Kann man in einem Programm eigentlich zwei Variablen gleichen Namens definieren?

Grundsätzlich müssen alle Namen (Bezeichner) eindeutig sein. Der Compiler unterscheidet allerdings unterschiedliche Gültigkeitsbereiche. Zu diesen gehören beispielsweise die einzelnen Methoden, Klassen und Namespaces. So ist es strikt verboten, innerhalb einer Methode zwei gleichnamige Variablen zu definieren, wohl aber ist es erlaubt, in

zwei verschiedenen Methoden zwei gleichnamige Variablen zu definieren oder einer lokalen Variablen in einer Methode den gleichen Namen zu geben wie einem Feld der Klasse. (Mehr zu diesem Thema in Kapitel 5.3.)

3. **Sind die Begriffe Standardkonstruktor und Ersatzkonstruktor synonym?**

   Nein! Ein Standardkonstruktor ist ein Konstruktor ohne Parameter. Ein Ersatzkonstruktor ist dagegen ein Konstruktor, der vom Compiler an Klassen zugeteilt wird, die keinen eigenen Konstruktor definieren. Der Ersatzkonstruktor hat keine Parameter (er ist also ein Standardkonstruktor).

3.10 Übungen

1. Datentypen sind das A und O der Variablendefinition. Zählen Sie die Schlüsselwörter auf, mit denen die verschiedenen elementaren Datentypen bei der Variablendefinition spezifiziert werden.
2. Welche der folgenden Variablendefinitionen sind nicht zulässig?

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

3. Wodurch sind String-Literale gekennzeichnet?
4. Wie kann man folgende Zeichen in Strings einbauen?
   - b
   - "
   - Zeilenumbruch
   - Tabulator
   - å (siehe Unicode-Tabelle im Anhang)
5. Was ist die Länge eines Strings und wie kann man sie ermitteln?
6. Worin liegt der Fehler bei der folgenden Klassendefinition?

```
Demo
{
    int x;
    int y;
}
```

7. Worin liegt der Fehler bei der folgenden Klassendefinition?

```
ref class Demo
{
   int x;
   int y;

   public: demo()
   {
      x = 0;
      y = 0;
   }
};
```

8. Welche der nachfolgenden Klassen besitzt einen Standardkonstruktor (Konstruktor ohne Parameter)?

```
ref class Demo1
{
};

ref class Demo2
{
    int wert;

    Demo2(int i)
    {
      wert = i;
    }
};

ref class Demo3
{
    int wert;

    Demo3(int i)
    {
       wert = i;
    }

    Demo3()
    {
       wert = 1;
    }
};
```

9. Erzeugen Sie ein Array `objekte`, das fünf Instanzen der nachfolgend definierten Klasse `Demo` enthält.

```
ref class Demo
{
public:
  int x;

  Demo(int i)
  {
    x = i;
  }
};
```