Kapitel 3

Konstanten und Variablen

Nachdem Sie im letzten Kapitel gelernt haben, wie die grundlegende Struktur eines C++-Programms aussieht, und wie man eine Zeichenkette auf den Bildschirm ausgibt, werden wir uns nun mit den verschiedenen Möglichkeiten der Speicherung und Nutzung von Daten beschäftigen. Die Daten, die wir uns in diesem Kapitel näher ansehen werden, sind Zahlen, Zeichen und Zeichenketten.

3.1 Zeichenkonstanten

Bei einer Zeichenkonstante handelt es sich um ein einzelnes in einfache Anführungszeichen eingeschlossenes Zeichen. Als Zeichen gilt dabei alles, was man über die Tastatur eingeben kann. Zusätzlich kann man noch Sonderzeichen und Steuerzeichen verwenden, die man nicht direkt eintippen kann.

Ein Steuerzeichen, den Zeilenvorschub (`\n`), haben Sie schon kennen gelernt. Ein Steuerzeichen erkennt man daran, dass es sich immer um eine Kombination aus Backslash (\) und einem weiteren Zeichen handelt.

Obwohl die Steuerzeichen streng genommen aus zwei Zeichen bestehen, werden sie in C++ wie ein einzelnes Zeichen behandelt. Der Backslash dient dabei eigentlich nur als Kennzeichnung dafür, dass es sich um ein Steuerzeichen handelt.

Testen Sie die Funktionsweise der Steuerzeichen aus Tabelle 3.1 einfach selbst. Das Programm in Listing 3.1: zeigt dazu ein kleines Beispiel. Aber experimentieren Sie ruhig selber ein wenig mit verschiedenen Steuerzeichen, damit Sie genau verstehen, was sie bewirken.

Tabelle 3.1: Sonderzeichen in C++

Steuerzeichen	Bedeutung
\a	Signalton (Beep)
\b	Backspace (linkes Zeichen löschen)
\f	Formfeed (Seitenvorschub)
\n	Newline (Zeilenvorschub)
\r	Return (Wagenrücklauf)
\t	Tabulator horizontal
\v	Tabulator vertikal
\"	doppeltes Anführungszeichen
\'	einfaches Anführungszeichen
\\	Backslash

Listing 3.1: Nutzung von Zeichen und Steuerzeichen

```
 1: // Das folgende Programm zeigt die Verwendung von
 2: // Zeichen und Steuerzeichen in C++
 3:
 4:
 5: #include <iostream.h>
 6:
 7:
 8: void main(void)
 9: {
10:     cout << 'A' << 'B' << '1' << '2' << '\\' << '\n';
11:
12:     cout << 'x' << '\t' << 'y' << '\n';
13:
14:     cout << "Beep" << '\a' << '\n';
15: }
```

Legen Sie für das Programm ein neues Projekt an, wie Sie es in Kapitel 2 für HelloWorld gemacht haben. Verwenden Sie nun als Projektnamen *Steuerzeichen* und nennen Sie auch die Datei *Steuerzeichen.cpp*. Geben Sie den Quellcode aus Listing 3.1: ein, kompilieren und linken Sie (F7) das Programm und starten Sie es anschließend (Strg+F5). Sie können auch direkt Strg+F5 drücken, Visual C++ fragt dann, ob Sie das Projekt erstellen wollen. Wenn Sie bei diesem Dialog auf JA drücken, wird das Programm in einem Schritt kompiliert, gelinkt und, wenn keine Fehler auftreten, ausgeführt.

3.2 Sonderzeichen

Alle Sonderzeichen und auch alle übrigen Zeichen sind Teil des ASCII-Zeichensatzes. ASCII steht für American Standard Code for Information Interchange (amerikanischer Standardcode zum Informationsaustausch). Dieser Code umfasst 256 verschiedene Zeichen und wurde in den frühen sechziger Jahren entwickelt, um alle damals benutzten Codes zu ersetzen. Die Zeichen, die man nicht direkt über Tasten erzeugen kann, kann man durch die Tastenkombination [Alt]+ASCII-Code des gewünschten Zeichens in Visual C++ anzeigen. So erzeugt z.B. [Alt]+[1][7][3] ein umgekehrtes Ausrufezeichen.

3.3 Zeichenkettenkonstanten

Zeichenkettenkonstanten kennen Sie schon aus Ihrem ersten Programm, HelloWorld. Sie bestehen aus mehreren Zeichen, die durch doppelte Anführungszeichen begrenzt sind. In einer Zeichenkettenkonstanten können auch Steuerzeichen vorkommen.

In C++ ist es möglich, sehr lange Zeichenketten auf mehrere Zeilen zu verteilen. Listing 3.2 zeigt ein Beispiel für Zeichenkettenkonstanten.

Listing 3.2: Eine lange Zeichenkette in C++

```
 1: // Lange Zeichenketten in C++
 2:
 3: #include <iostream.h>
 4:
 5:
 6: void main(void)
 7: {
 8:     cout << "Dies ist ein Beispiel für eine"
 9:             "Zeichenkette, die sich über mehrere\n"
10:             "Zeilen erstreckt.";
11: }
```

Sie können für das Programm aus Listing 3.2 entweder wieder ein neues Projekt anlegen oder den Quelltext des letzten Beispiels, *Sonderzeichen.cpp*, durch den Quelltext aus Listing 3.2 ersetzen. Ihnen geht auch dann nichts verloren, da alle Beispiellistings auf der CD, nach Kapiteln sortiert, vorhanden sind.

3.4 Numerische Konstanten

Zuerst werden wir uns Konstanten genauer anschauen. Konstanten zeichnen sich dadurch aus, dass sie in der ganzen Zeit, in der das Programm läuft, immer den gleichen Wert beibehalten.

Eine Art von Konstanten, die in fast jedem Programm vorkommen, sind die numerischen Konstanten. Listing 3.3 zeigt die Verwendung von numerischen Konstanten.

Listing 3.3: Verwendung von numerischen Konstanten

```
 1: // Das folgende Programm zeigt die Nutzung von
 2: // numerischen Konstanten
 3:
 4: #include <iostream.h>
 5:
 6:
 7: void main(void)
 8: {
 9:    cout << "Zwei mal drei ist 6\n";
10:
11:    cout << "Zwei mal drei ist " << 6 << "\n";
12:
13:    cout << "Zwei mal drei ist " << 2*3 << "\n";
14: }
```

Die Programmzeilen 1-8 sind Ihnen sicher schon vertraut. In den ersten beiden Programmzeilen wird das Programm in zwei Kommentaren beschrieben. Danach wird die Datei `iostream.h` eingebunden, und die `main`-Funktion beginnt.

In Zeile 9 wird der String `Zwei mal drei ist 6` mit einem anschließenden Zeilenumbruch ausgegeben. Es scheint, dass die Zeilen 11 und 13 auch nichts anderes machen, da die Bildschirmausgabe gleich ist. Es gibt aber einige bemerkenswerte Unterschiede.

In der 9. Zeile ist die Zahl 6 Teil des Strings und wird deshalb auch nicht wie eine Zahl, sondern wie ein Zeichen behandelt. Die Zeilen 11 und 13 sind sich hingegen sehr ähnlich. In beiden Fällen wird zwar auch die Zahl 6 ausgegeben, diesmal handelt es sich aber jeweils um eine Zahl und nicht etwa um einen String wie in Zeile 9.

`cout` kann also nicht nur Zeichenketten, sondern auch Zahlen auf den Bildschirm ausgeben. In Zeile 9 wird eine Zeichenkette ausgegeben. In den Zeilen 11 und 13 wird erst eine Zeichenkette, dann eine Zahl und dann wieder eine Zeichenkette ausgegeben. Dafür ist es aber nicht nötig, dreimal `cout` zu verwenden. Es genügt, wenn man die verschiedenen Ausgaben mit dem Umleitungsoperator kombiniert.

Der Compiler berechnet die Multiplikation 2*3 in Zeile 13 schon vor der Übersetzung des Programms und ersetzt 2*3 durch 6. Durch diese Ersetzung wird die Berechnung zur Laufzeit des Programms nicht durchgeführt.

Immer wenn eine Berechnung schon vor der Übersetzung des Quelltextes durchgeführt werden kann, wird der Compiler sie durch das Ergebnis ersetzen. Dies ist ein Optimierungsschritt, den fast jeder C++-Compiler vor der Übersetzung ausführt. Die Zeilen 11 und 13 werden sich also nach der Übersetzung nicht mehr unterscheiden.

An diesem kurzen Beispiel hat sich gezeigt, dass es unterschiedliche Arten von Konstanten gibt. In unserem Beispiel waren das Zeichenkettenkonstanten, wie z.B. der String `Zwei mal drei ist 6` oder die numerischen Konstanten `6`, `2` und `3`.

3.5 Fließkommakonstanten

Innerhalb der numerischen Konstanten wird zwischen ganzzahligen Konstanten und Fließkommakonstanten unterschieden. Ganzzahlige Konstanten sind Ihnen in Listing 3.3 schon begegnet. Bei Fließkommakonstanten ist darauf zu achten, dass sie mit einem Dezimalpunkt anstatt des in Deutschland üblichen Kommas verwendet werden müssen.

Listing 3.4: Fließkommakonstanten

```
 1: // Das folgende Programm zeigt die Nutzung von
 2: // Fließkommakonstanten
 3:
 4:
 5: #include <iostream.h>
 6:
 7:
 8: void main(void)
 9: {
10:     cout << "PI  = " << 3.14159 << "\n";
11:
12:     cout << "1/2 = " << 1 / 2 << "\n";
13:
14:     cout << "1/2 = " << 1.0 / 2.0 << "\n";
15: }
```

Wenn Sie das Programm eingeben und ausführen, fällt zunächst auf, dass das Programm ein unerwartetes Ergebnis produziert. Bei der Division in Zeile 12 erwartet man, dass der Computer 0.5 und nicht 0 ausgibt. Bei der Berechnung in Zeile 14 gibt er ja schließlich auch das richtige Ergebnis aus.

Dieses scheinbar merkwürdige Verhalten ist damit zu erklären, dass der C++-Compiler bei einer Division von zwei Ganzzahlen davon ausgeht, dass das Ergebnis auch wieder eine Ganzzahl sein soll.

Wenn man dem Compiler, wie in Zeile 14, mitteilt, dass es sich um eine Division von Fließkommazahlen handeln soll, indem man die Zahlen jeweils um einen Dezimalpunkt und eine Null erweitert, gibt er auch das erwartete Ergebnis aus. Bei der Ausgabe handelt es sich also keinesfalls um ein fehlerhaftes Verhalten des Compilers oder des Computers, sondern lediglich um eine andere Interpretation durch den Compiler.

Achten Sie also immer genau darauf, welchen Typ Ihr Ergebnis haben soll und passen Sie in Abhängigkeit davon die verwendeten Konstanten an.

Fließkommakonstanten können in C++ auch in Exponentialschreibweise dargestellt werden. Dies ist besonders bei sehr großen oder sehr kleinen Zahlen nützlich. Die Zahl 30000000 kann auch als 3.0E7 geschrieben werden. Die Zahl 0.0000001 kann man durch 1.0E-7 abkürzen.

3.6 Datentypen in C++

Bevor wir uns mit einer weiteren Art von Konstanten, den symbolischen Konstanten und schließlich mit den Variabeln beschäftigen, ist es sinnvoll, dass Sie die Datentypen von C++ kennen lernen. Datentypen haben bestimmte Wertebereiche und benötigen unterschiedlich viel Platz im Speicher. Tabelle 3.2 zeigt die wichtigsten Datentypen, deren Größe und Wertebereich.

Tabelle 3.2: Die wichtigsten C++-Datentypen

Datentyp	**Größe in Byte**	**Wertebereich**
bool	1	true oder false
char	1	256 verschiedene Zeichen
short	2	-32768 bis +32767
int	4	-2147483648 bis +2147483647
float	4	3.4E +/- 38 (7 Ziffern)
double	8	1.7E +/- 308 (15 Ziffern)

Bei dem Datentyp `bool` handelt es sich um eine Abkürzung für Boolean. Werte dieses Datentyps können die Werte wahr (true) oder falsch (false) annehmen.

Der Datentyp `char` steht für character (Zeichen) und repräsentiert jeweils ein Zeichen aus dem ASCII-Zeichensatz.

Die Datentypen `int` und `short` sind ganzzahlige Datentypen, wobei `int` für integer (Ganzzahl) steht. Der Datentyp `short` steht für short integer, was soviel wie verkürzter Integer heißt. Anhand der Namen wird schon deutlich, dass `int` einen wesentlich größeren Wertebereich abdeckt als der Datentyp `short`. Dafür wird für den Datentyp `short` aber auch weniger Speicher benötigt.

Die Datentypen `float` und `double` repräsentieren Fließkommawerte und unterscheiden sich wie `short` und `int` auch im Wertebereich und im Speicherverbrauch. `float` steht für Fließkommawert (floating point value). Die Bezeichnung `double` steht für einen Fließkommawert mit doppelter Genauigkeit. Die doppelte Genauigkeit bezieht sich auf die größere Anzahl der Nachkommastellen, die dargestellt werden können. Dies ist möglich, da man bei diesem Typ doppelt so viel Speicher, wie bei einem `float`-Wert nutzen kann.

3.7 Symbolische Konstanten

Bei den Konstanten, die wir bisher kennen gelernt haben, handelt es sich um literale Konstanten. Bei einer literalen Konstante schreibt man den Wert direkt an die Stelle im Programm, an der er benötigt wird. Symbolische Konstanten werden hingegen über einen Bezeichner identifiziert. Ein Bezeichner oder ein Konstantenname besteht aus einer Kombination aus Buchstaben und Zahlen.

Stellen Sie sich vor, Sie schreiben ein Programm, das viele mathematische Berechnungen durchführt und die Konstante PI häufig benötigt. Wenn Sie bei jeder Verwendung von PI im Quellcode immer 3.14159 eintippen müssten, wäre das doch recht lästig und außerdem fehleranfällig. Deshalb kann man in C++ Konstanten einen Namen geben und diesen Namen anstatt der Konstante verwenden.

Listing 3.5: Symbolische Konstanten in C++

```
 1: // Symbolische Konstanten
 2:
 3: #include <iostream.h>
 4:
 5: // Konstantendeklarationen
 6:
 7: const float PI          = 3.14159f;
 8: const float Kreisradius = 6.0f;
 9: const char  NeueZeile   = '\n';
10:
```

```
11: void main(void)
12: {
13:     cout << "Der Umfang des Kreises beträgt : "
14:          << 2 * PI * Kreisradius
15:          << NeueZeile;
16:
17:     cout << "Die Fläche des Kreises beträgt : "
18:          << PI * Kreisradius * Kreisradius
19:          << NeueZeile;
20: }
```

In den Zeilen 7 und 8 in Listing 3.5 werden zwei Fließkommakonstanten definiert. Die Definition von Konstanten und Variablen nennt man auch Deklaration. Die Deklaration einer symbolischen Konstante beginnt in C++ mit dem Schlüsselwort `const`. Danach folgt der Typ der Konstanten. Die Konstanten in den Zeilen 7 und 8 haben den Typ `float`. Nach der Typangabe folgt der Bezeichner der Konstanten, ein Gleichheitszeichen und schließlich ihr Wert. Der Bezeichner ist der Name der Konstante, den man im Programm immer dann benutzt, wenn man den Wert der Konstanten verwenden will.

Dass nach dem Wert der Konstante ein `f` steht, hat wieder mit der Interpretation der Konstanten durch den Compiler zu tun. Wenn Sie kein `f` an diese Stelle schreiben würden, gäbe der Compiler eine Warnung aus. Der Compiler gibt jedem Fließkommawert standardmäßig den Typ `double`, erst wenn Sie der Konstanten ein `f` anfügen, interpretiert der Compiler den Wert als einen `float`-Wert und die Warnung verschwindet.

In der 9. Zeile wird die Konstante `NeueZeile` vom Typ `char` deklariert. Sie erhält den Wert des Zeilenumbruchs und kann so anstatt des Sonderzeichens `'\n'` verwendet werden. In den Zeilen 15 und 17 wird also keinesfalls die Zeichenkette `NeueZeile` ausgegeben. `NeueZeile` ist eine Zeichenkonstante, die mit dem Wert des Zeilenumbruchs belegt worden ist und nun an dessen Stelle verwendet werden kann.

Durch die Verwendung von symbolischen Konstanten kann man den Quellcode leichter lesen, und er wird schneller verständlich. Wenn man im Code PI liest, kann man damit wahrscheinlich mehr anfangen als mit 3.14159. Ein weiterer Vorteil ist, dass Sie die Konstante `Kreisradius` nur einmal in Zeile 8 verändern müssen, um die Fläche oder den Umfang eines anderen Kreises zu berechnen. Hätten Sie anstatt der symbolischen Konstante `Kreisradius` immer literale Konstanten verwendet, müssten Sie den Quellcode an mehreren Stellen ändern.

3.8 Variablen

Variablen kann man während des Programmablaufs, im Gegensatz zu Konstanten, immer wieder neue Werte zuweisen. Variablen werden wie symbolische Konstanten deklariert, der einzige Unterschied ist, dass das `const` wegfällt. Dadurch weiß der Compiler, dass es sich bei dem folgenden Bezeichner um einen Variablennamen handelt. Für Variablen gelten die gleichen Typen und Wertebereiche wie für Konstanten. Eine Konstante muss bei der Deklaration einen Wert erhalten. Einer Variablen kann bei der Deklaration ein Wert zugewiesen werden. Dies ist aber nicht zwingend notwendig, da Sie den Wert einer Variablen zur der Laufzeit des Programms ständig ändern können.

Listing 3.6: Deklaration und Verwendung von Variablen

```
 1: // Verwendung von Variablen
 2:
 3: #include <iostream.h>
 4:
 5: int x = 10;  // Variablendeklarationen
 6:
 7: void main(void)
 8: {
 9:    cout << "x hat jetzt den Wert " << x << '\n';
10:
11:    x = 20; // der Variablen x einen neuen Wert zuweisen
12:
13:    cout << "x hat jetzt den Wert " << x << '\n';
14: }
```

Listing 3.6 zeigt, wie man Variablen deklariert, und wie man den Wert auch zur Laufzeit des Programms verändern kann. In der 5. Zeile wird die Integer-Variable `x` deklariert. Die Variable erhält den Wert `10`. Das erste Setzen des Wertes einer Variable nennt man auch die Initialisierung der Variable.

In der 9. Zeile wird der Wert von `x` ausgegeben. `x` hat zu diesem Zeitpunkt den Wert 10. In der 11. Zeile wird `x` ein neuer Wert zugewiesen und in der 13. Zeile ausgegeben. Dies ist nur möglich, da `x` eine Variable ist. Wenn Sie `x` als Konstante deklariert hätten, würde der Compiler an dieser Stelle eine Fehlermeldung bringen.

Achten Sie immer darauf, dass der Typ, den Sie für Ihre Variablen wählen, einen ausreichend großen Wertebereich besitzt. In Listing 3.6 wurde für die Variable `x` der Typ `int` gewählt, der für das Beispiel auf jeden Fall groß genug ist. Wenn Sie aber z.B. Variablen vom Typ `short` nutzen und in Ihrem Programm komplexe Berechnungen durchführen, bei denen Werte außerhalb des Wertebereiches auftreten können, führt dies zu schwer auffind-

baren Fehlern. Am Ende dieses Kapitels, im Abschnitt Definition neuer Typen, finden Sie ein Beispiel dazu.

3.9 Globale und lokale Variablen

Bei den Variablen, die Sie bis jetzt genutzt haben, handelt es sich ausschließlich um globale Variablen. Globale Variablen stehen im gesamten Programm zur Verfügung und werden meist zu Beginn des Programms deklariert. Sie können Variablen aber auch lokal in der `main`-Funktion deklarieren. Abbildung 3.1 zeigt die schematische Darstellung eines Programms und dessen Gültigkeitsbereiche. Die Gültigkeitsbereiche werden jeweils von geschweiften Klammern eingeschlossen.

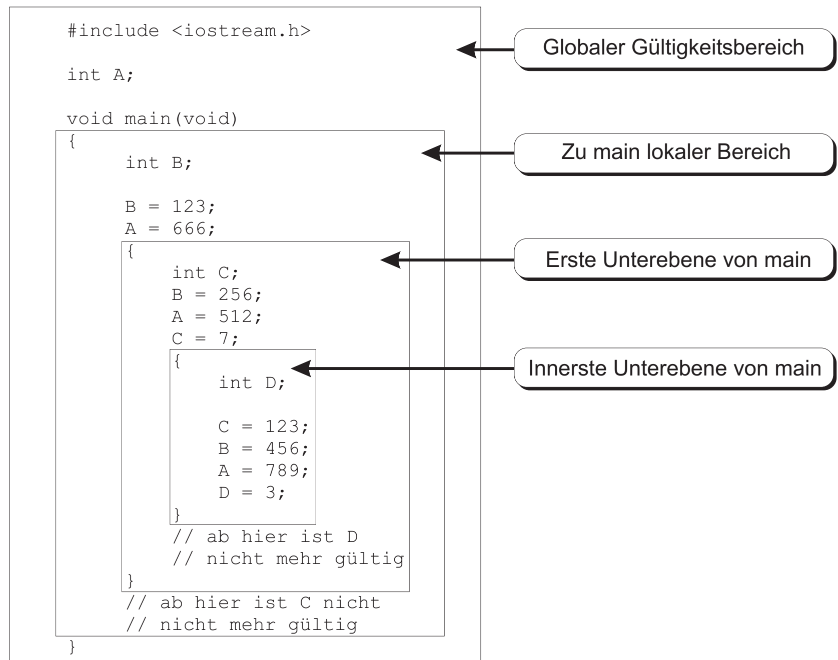


Abb. 3.1: Gültigkeitsbereiche in einem C++-Programm

Das Programm enthält nur verschachtelte Variablendeklarationen, führt keine Berechnung durch und hat auch keine Ausgabe. Es soll Ihnen nur verdeutlichen, wo die deklarierten Variablen gültig sind.

Die Variable `A` ist in der `main`-Funktion und (wie Sie später sehen werden) darüber hinaus gültig. Die Variable `A` ist eine globale Variable. Die Variable `B` kann nur in der `main`-Funktion benutzt werden, sie ist lokal zur `main`-Funk-

tion. Die Variable C ist in der ersten Unterebene der main-Funktion gültig. Die Variable D ist nur innerhalb des zweiten Blocks, der innersten Ebene gültig.

Wenn Sie versuchen, die Variablen D außerhalb Ihres Gültigkeitsbereichs zu nutzen (wie durch den Kommentar angedeutet), dann würde Ihnen der Compiler eine Fehlermeldung anzeigen, die Sie darauf aufmerksam macht, dass D an dieser Programmstelle unbekannt ist. Deshalb kann man auch nicht vor der Definition der Variablen D, in einer der höheren Ebenen auf D zugreifen.

Für globale Variablen wird permanent, also für die gesamte Laufzeit des Programms, Speicher reserviert. Für lokale Variablen wird erst dann Speicher reserviert, wenn das Programm während des Ablaufs in den Gültigkeitsbereich eintritt, und der Speicher wird auch nach dem Verlassen des Gültigkeitsbereichs der lokalen Variable wieder freigegeben.

Ein weiterer wichtiger Punkt ist, dass lokale Variablen immer höhere Priorität haben als globale Variablen. Listing 3.7 soll dies verdeutlichen.

Listing 3.7: Globale und lokale Variablen

```
 1: // Verwendung von globalen und lokalen Variablen
 2:
 3: #include <iostream.h>
 4:
 5: int x = 10;  // globale Variable
 6:
 7: void main(void)
 8: {
 9:    int x = 200;  // lokale Variable
10:
11:    cout << "x hat den Wert " << x << '\n';
12: }
```

Wie Sie sehen, wird in dem Programm sowohl eine globale als auch eine lokale Variable mit dem gleichen Namen, nämlich x, deklariert. Es ist zulässig, Variablen in verschiedenen Ebenen gleiche Namen zu geben. So könnten Sie auch in Abbildung 3.1 in jedem Block eine Variable mit gleichem Namen deklarieren. Ein Fehler tritt erst dann auf, wenn Sie zwei Variablen im identischen Gültigkeitsbereich den gleichen Namen geben.

Aber woher weiß der Compiler nun, welche Variable gemeint ist? Soll er die lokale oder die globale Variable verwenden? Dafür gibt es wieder eine festgelegte Regel: die Variable, die lokal zu dem Block ist, in dem man sich gerade befindet, wird benutzt. Die lokale Variable hat also immer »Vorfahrt«.

Sie sollten alle Variablen, die Sie deklarieren, auch immer initialisieren. Globale Variablen, die Sie nicht initialisieren, bekommen zwar von Visual C++ automatisch den Wert 0 zugewiesen, der Wert einer lokalen Variablen ist

aber unbestimmt. Außerdem übernehmen nicht alle Compiler die Initialisierung globaler Variablen automatisch für Sie. Um Fehler zu vermeiden, sollten Sie also grundsätzlich alle Variablen initialisieren.

3.10 Zusätze für Konstanten und Variablen

Bei der Deklaration von Konstanten und Variablen können Sie noch verschiedene Zusätze oder Modifizierer einsetzen.

const

Einen Zusatz haben Sie bereits kennen gelernt. Mit `const` wird aus einer Variablen eine Konstante. Konstanten müssen direkt bei der Deklaration initialisiert werden.

signed und unsigned

Mit den Zusätzen `signed` und `unsigned` können Sie festlegen, ob eine Variable vorzeichenbehaftet ist (`signed`) oder nicht (`unsigned`). Vorzeichenbehaftet bedeutet, dass die Variable auch negative Werte annehmen kann.

Die Idee dabei ist, dass man den Wertebereich bei 0 beginnen lässt, wenn man keine negativen Zahlen nutzen möchte. Dafür kann man dann mehr Werte in positiver Richtung nutzen. Wenn Sie eine Integer-Variable als `int Zahl;` deklarieren, kann diese Werte von -2147483648 bis +2147483647 annehmen. Wenn Sie den Zusatz `unsigned` verwenden, die Variable also als `unsigned int Zahl;` deklarieren, können Sie nun Werte von 0 bis 4294967295 verwenden. Eine Variable, die Sie als `unsigned short` deklarieren, kann somit Werte von 0 bis 65535 annehmen.

Wenn Sie keinen Zusatz verwenden, ist die Variable, die Sie deklariert haben immer vorzeichenbehaftet.

register

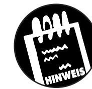
Der Zusatz `register` bedeutet, dass die Variable, wenn möglich direkt in eines der Prozessorregister gespeichert werden soll. Dies dient der Verbesserung der Geschwindigkeit eines Programms, da auf Variablen, die in Prozessorregistern stehen, schneller zugegriffen werden kann. Dieser Zusatz wird von Visual C++ aber nicht berücksichtigt. Der Compiler von Visual C++ ist ein optimierender Compiler und verwendet daher seine eigenen Optimierungsstrategien. Wenn Sie einen anderen Compiler einsetzen, ist es aber durchaus möglich, dass das `register`-Schlüsselwort seinen Zweck erfüllt.

Eine weitere Einschränkung ist, dass man den `register`-Zusatz nur für lokale Variablen verwenden kann.

Das folgende Listing zeigt Ihnen, wie man die verschiedenen Variablenzusätze verwendet.

```
 1: // Verwendung von Variablenzusätzen
 2:
 3: #include <iostream.h>
 4:
 5: unsigned int y = 0;     // y ist vorzeichenlos
 6:
 7: const int b = 10000;    // Konstante b initialisieren
 8:
 9: void main(void)
10: {
11:     // register hat in Visual C++ keine Auswirkungen
12:     register int x = 3333;
13:
14:     y = 3 * x + b;
15:
16:     cout << y << '\n';
17: }
```

Listing 3.8: Variablenzusätze



Es gibt noch weitere Variablenzusätze in C++, die Sie mit der Zeit, im Zusammenhang mit anderen Themen kennen lernen werden.

3.11 Konstanten mit #define definieren

Es gibt noch eine weitere Möglichkeit, Konstanten zu definieren und zwar mit der Präprozessoranweisung `#define`. Wenn Sie `#define` verwenden, müssen Sie zuerst den Namen und dann den Wert angeben. Vor der Kompilierung ersetzt der Präprozessor jedes Auftreten der Konstante mit ihrem Wert. Für Konstanten, die mit `#define` erzeugt wurden, wird kein Speicherplatz reserviert. Listing 3.9 zeigt den Einsatz von `#define`.

```
 1: // Einsatz der Präprozessoranweisung #define
 2:
 3: #include <iostream.h>
 4:
 5:
 6: #define PI 3.14159
 7: #define RADIUS 3
```

Listing 3.9: Verwendung der Präprozessoranweisung #define

```
 8:
 9: void main(void)
10: {
11:     cout << 2 * PI * RADIUS;
12. }
```

In den Zeilen 6 und 7 werden die zwei Konstanten `PI` und `RADIUS` definiert und in der Zeile 11 verwendet. Beachten Sie, dass `PI` und `RADIUS` über keine Typinformationen verfügen. Vor der Kompilierung wird der Präprozessor `PI` und `RADIUS` durch die definierten Werte ersetzen, so dass der Compiler die Zeile 11 als

```
cout << 2 * 3.14159 * 3;
```

übergeben bekommt und dann übersetzt.

Der Vorteil von `#define` liegt darin, dass auf diese Weise kein Speicher verbraucht wird. Ein Nachteil ist, dass man sehr genau auf die Interpretation der Typen durch den Compiler achten muss, um nicht unerwünschte Effekte wie in Listing 3.4 zu erhalten.

3.12 Definition neuer Typen

In C++ gibt es die Möglichkeit, neue Typen mit Hilfe von `typedef` zu erzeugen. Die Syntax von `typedef` lautet:

`typedef` *Bekannter-Typ Neuer-Typ*.

Bekannter-Typ ist dabei einer der C++-Datentypen oder ein Datentyp, den Sie schon vorher mit `typedef` definiert haben. *Neuer-Typ* ist der Name, den Sie für Ihren Datentyp verwenden wollen.

Listing 3.10: typedef und Bereichsüberschreitung

```
 1: // Einsatz von typedef
 2:
 3: #include <iostream.h>
 4:
 5:
 6: #define NZ '\n'
 7:
 8: typedef unsigned short GEHALT;
 9:
10: short  GehaltMeier;
11: GEHALT GehaltMueller;
12:
13: void main(void)
14: {
15:     // Gehälter für Herrn Meier und Herren Mueller in DM
16:     GehaltMeier   = 30000;
```

```
17:     GehaltMueller = 30000;
18:
19:     cout << GehaltMeier << NZ << GehaltMueller << NZ;
20:
21:     // Herr Müller und Herr Meier bekommen eine
22:     // Lohnerhöhung von 5000 DM
23:     GehaltMeier   = GehaltMeier   + 5000;
24:     GehaltMueller = GehaltMueller + 5000;
25:
26:     cout << GehaltMeier << NZ << GehaltMueller << NZ;
27: }
```

Listing 3.10 zeigt den Einsatz von `typedef` und einer damit im Zusammenhang stehenden Wertebereichsüberschreitung.

In Zeile 8 wird mit `typedef` der neue Datentyp `GEHALT` erzeugt. Dies ist dann sinnvoll, wenn man häufig z.B. Variablen vom Typ `unsigned short` verwenden möchte, sich aber den Tippaufwand sparen möchte oder aber, wenn man durch die Typbezeichnung den Zweck beschreiben will.

Hier wird der neue Typ `GEHALT`, der das Gehalt eines Mitarbeiters speichern soll, definiert. Für das Gehalt des Herrn Meier wird eine `short` Variable deklariert. Im Gegensatz dazu wird für das Gehalt von Herrn Müller eine Variable des gerade definierten Typs `GEHALT` genutzt.

Wenn Sie das Programm eingeben und ausführen, werden Sie sehen, dass Herr Meier nicht glücklich über seine Gehaltserhöhung sein wird. In der Zeile 23 tritt eine Wertebereichsüberschreitung auf, und so bekommt Herr Meier wesentlich weniger Geld als zuvor.

Der Wert -30536 für das Gehalt von Herrn Meier kommt folgendermaßen zustande. Herr Meier hat ein Gehalt von 30000 DM, und sein Gehalt wird in einer `short`-Variablen gespeichert, in der Werte von -32768 bis +32767 (vgl. Tabelle 3.2) gespeichert werden können. Wenn nur 2767 zu den 30000 hinzugekommen wären, bliebe das Gehalt positiv. Wenn aber 32767 überschritten wird, wird bei -32768 weitergerechnet, und die verbleibende Summe wird hinzugerechnet (-32768 + 2233 = -30536).

Zusammenfassung

In diesem Kapitel haben Sie eine Menge neuer Informationen über die Programmierung mit C++ erhalten und viele neue Aspekte kennen gelernt. Nehmen Sie sich genug Zeit, um alle Beispiele genau nachzuvollziehen und probieren Sie das, was Sie gelernt haben, selbst aus. Nur so wird es Ihnen gelingen, wirklich alles im Detail zu verstehen. Wenn Sie an einen Punkt stoßen, an dem Sie mehr wissen wollen oder nicht so recht weiterkommen,

werfen Sie einen Blick in die Hilfe. Hier finden sich eine Fülle von Zusatzinformationen und Beispielen.

Nachdem Sie dieses Kapitel durchgearbeitet haben, sollten Sie wissen, wie man Konstanten und Variablen deklariert und diese verwendet. Sie sollten die vorgestellten Datentypen und deren Wertebereiche sowie die Unterschiede in der Nutzung von globalen und lokalen Variablen kennen.

Außerdem sollten Sie sich merken, dass die Präprozessoranweisung `#define` eine Textersetzung im Quellcode vornimmt, und dass man mit `typedef` neue Typen in C++ definieren kann.