6

€ 4,-

KnowWare EXTRA

C++

für Einsteiger

```
#include <iostream.h>
#include <string>

int main()
{
char *tell = new char;
if (tell == NULL) {cout << "Kein Speicher!" << "\n";}
*tell = 'H';

cout << *tell << "all";
char x = 111;

string *watergate = new string;
if (watergate == NULL) {cout << "Kein Speicher!" << "\n";}

*watergate = " C++!\n;
cout << x << *watergate;

return 5;
}
```

Dirk Ammelburger

www.KnowWare.de

Deutschland: 4,- EUR Österreich: 4,60 EUR
Schweiz: 8 SFR Luxemburg: 4,70 EUR Italien: 5,50 EUR

Acrobat Reader: Wie ...

F5/F6 öffnet/schließt die Ansicht **Lesezeichen**

Strg+F sucht

Im Menü Ansicht stellst du ein, wie die Datei gezeigt wird

STRG+0 = Ganze Seite **STRG+1** = Originalgrösse **STRG+2** = Fensterbreite

Im selben Menü kannst du folgendes einstellen:: **Einzelne Seite**, **Fortlaufend** oder **Fortlaufend - Doppelseiten** .. Probiere es aus, um die Unterschiede zu sehen.

Navigation

Pfeil Links/Rechts: eine Seite vor/zurück

Alt+ Pfeil Links/Rechts: Wie im Browser: Vorwärts/Zurück

Strg+**+** vergrößert und **Strg**+**–** verkleinert

Bestellung und Vertrieb für den Buchhandel

Bonner Pressevertrieb, Postfach 3920, D-49029 Osnabrück
Tel.: +49 (0)541 33145-20
Fax: +49 (0)541 33145-33
bestellung@knowware.de
www.knowware.de/bestellen

Autoren gesucht

Der KnowWare-Verlag sucht ständig neue Autoren. Hast du ein Thema, daß dir unter den Fingern brennt? - ein Thema, das du anderen Leuten leicht verständlich erklären kannst?

Schicke uns einfach ein paar Beispielseiten und ein vorläufiges Inhaltsverzeichnis an folgende Adresse:

lektorat@knowware.de

Wir werden uns deinen Vorschlag ansehen und dir so schnell wie möglich eine Antwort senden.

www.knowware.de

Vorwort

Jetzt sitze ich wieder einmal vor einem Vorwort und weiß nicht recht, wie ich anfangen soll. Keine Angst – das Heft selbst ist weit zielstrebiger als die folgenden Zeilen, aber ein Vorwort ist immer etwas undankbar. Aber was soll's: Ich versuche es einfach mal!

Der Titel verspricht ja einiges: C++ für Einsteiger! Aber was heißt das? Oder besser: an wen richtet sich dieses Heft und was bekommst du geboten?

Die Zielgruppe ist schnell bestimmt und lässt sich mit wenigen Worten beschreiben. Jeder, der sich für Programmierung interessiert und bisher weder den Mut noch Zeit besessen hat, sich näher mit der Materie auseinander zusetzen, darf sich von diesem Titel angesprochen fühlen. Voraussetzung für diesen Kurs ist nur ein wenig Erfahrung mit einem Computer. Ich werde hier nicht erklären was ein Editor ist oder wie man ein DOS-Fenster öffnet – das solltest du bereits wissen. Ansonsten gilt: Jeder darf mitmachen. Wir werden bei Null anfangen und uns langsam in die Materie einarbeiten, so dass jeder auf seine Kosten kommt.

Trotzdem gehört dieses Heft sicher zu den eher anspruchsvollen Werken aus der KnowWare-Reihe, auch wenn es sich an die Riege der Anfänger richtet. So lässt sich z.B. „Word für Einsteiger" nicht mit dem Erlernen einer komplexen Programmiersprache wie C++ vergleichen, die uns in so alltäglichen und oft genutzten Programmen wie Windows oder Word begegnet. Das soll dich natürlich nicht abschrecken – ganz im Gegenteil! Aber es rückt die Relation dieses Heftes zu anderen KnowWare-Heften ins rechte Licht.

Das Heft richtet sich speziell an Anfänger. Ich habe also versucht die Syntax so anschaulich wie möglich zu gestalten, damit die teilweise sehr abstrakte Materie etwas klarer wird. Es ist gut möglich, dass ich an einigen Stellen auf Kosten der Genauigkeit auf weiterführende Erläuterungen verzichtet habe oder an anderen Stellen den Stoff so weit vereinfachen musste, dass manche Details unter den Tisch gefallen sind. Ich möchte damit niemanden etwas vorenthalten, sondern schlicht und einfach den Stoff verständlicher machen.

Ich denke, jeder Anfänger ist für eine anschauliche Erklärung dankbar, wenn er bisher immer nur graue Theorie vorfand. Mein Ziel ist, dass du so schnell wie möglich Ergebnisse siehst und etwas mit dem erlernten Wissen anzufangen weißt.

Fühlt sich also irgendein „Profi" berufen, mich auf Ungenauigkeiten im Heft hinzuweisen, sei ihm gesagt, dass er seine Zeit verschwendet.

Für konstruktive Kritik aller Art bin ich selbstverständlich nach wie vor immer offen – vor allem wenn es um Fragen oder Vorschläge für die vorgestellten Programme geht.

> Alle Programme sind getestet und nach besten Wissen und Gewissen geschrieben – sie sollten also funktionieren! Sollte das nicht der Fall sein, bitte ich um eine Mail mit den Angaben zum benutzten Compiler und dem Betriebssystem. Ich bin natürlich nicht Jesus. Sofern meine Zeit aber reicht, kann ich sicher ein paar Tipps geben.

In diesem Sinne: Viel Spaß beim Lesen und Ausprobieren!

Danksagung

Bevor wir endgültig loslegen, möchte ich noch ein paar netten Menschen danken, die mir bei diesem Heft sehr geholfen haben. Zunächst geht mein Dank an Karl Antz und Michael Maardt vom KnowWare Verlag, die mir stets mit Rat und Tat zur Seite standen. Dank ihnen schaffte ich es, die Tücken von Word schließlich doch zu umschiffen.

Dann möchte ich wieder einmal meiner Freundin Gaby danken, die wie fast immer meine durchschriebenen Wochenenden geduldig ertrug. Dein Langmut hat mir immer den Rücken gestärkt...

Zu guter letzt geht mein Dank an alle Leser der KnowWare-Hefte, die das hier erst ermöglichen. Speziell den hilfreichen Tipps der Autorenberatung gebührt Anerkennung. Aber auch die Leser meines letzten Heftes „CGI mit Perl" möchte ich hier erwähnen. Dank ihnen entschloss ich mich, mir noch mal die Nächte um die Ohren zu hauen. ☺

Dirk Ammelburger

Was ist C++ ?

Die Programmiersprache *C++* ist weltweit die beliebteste Entwicklersprache für professionelle Software. Mit ihr kannst du sehr systemnah komplexe und schnelle Programme schreiben, die auf jedes Betriebsystem portierbar sind. Ich will dich nicht groß mit Geschichte langweilen, darum hier nur ein paar kurze Daten: C++ ist als Nachfolger der 'Ursprache' C eine recht junge Sprache. Die vorherige Generation C wurde um einige wichtige Punkte erweitert. Im großen und ganzen verdanken wir das einem Mann namens Bjarne Stroustrup. Was er hinzufügte, sind die Merkmale der objektorientierten Programmierung. Die Pluszeichen hinter dem C stehen für die Erhöhung oder, wie das in der offiziellen Syntax heißt, die *Inkrementierung* von C.

C wie auch C++ sind Compilersprachen. Was besagt, dass der Quellcode vor der Ausführung in Maschinensprache übersetzt werden muss. Das geschieht mit Hilfe eines *Compilers*. Ein solcher Compiler ist eine Grundvoraussetzung für das Schaffen von C++-Programmen. Wie du ihn erhältst, erfährst du in den nächsten Kapiteln.

C++ ist eine Obermenge von C. Das besagt, dass jedes gültige C-Programm mit einem C++-Compiler übersetzt werden kann. Was es nun aber keinesfalls bedeutet, ist, dass man C-Kenntnisse haben müsste, um C++ zu erlernen – ganz im Gegenteil ist es oft besser, völlig unvoreingenommen an diese Sprache heranzugehen, da man so nicht in festgefahrenen Strukturen denkt. C++ ist nicht annähernd so eng mit C verwandt wie man anfänglich glauben möchte. Darum sollte man auch nie von C auf C++ schließen.

Wenn du C++ programmierst, spielt die Wahl des Betriebssystems keine Rolle. Darum gehe ich in diesem Heft auch nicht von irgend welchen Annahmen über deinen Computer aus. C++ ist komplett portabel, d.h. es läuft auf jedem Rechner, sei es ein Windows-PC, ein Mac oder ein Großrechner in Silicon Valley. Vor C++ sind alle gleich. Einzige Voraussetzung ist ein kompatibler Compiler.

Welche C++-Version?

Es gibt verschiedene Arten C++ zu programmieren. In diesem Heft behandle ich ausschließlich ANSI-C++, das gewissermaßen einen Standard dieser Programmiersprache darstellt. Die meisten Programme sollten aber, wenn man ein paar kleine Änderungen vornimmt, auch für viele andere C++-Varianten gültig sein.

Was kostet C++ ?

Zunächst eine frohe Botschaft: Der C++-Compiler kostet dich keinen Pfennig! Hast du das Glück auf einem Linux- oder UNIX-System zu arbeiten, kannst du wahrscheinlich dieses und das nächste Kapitel getrost überspringen, da bei den meisten Distributionen ein C++-Compiler mitgeliefert wird. Er ist auf deinem Rechner bereits installiert und eingerichtet und wird mit **`gcc`** gestartet.

Hast du – wie ich – einen Windows-Rechner, ist das auch kein Problem, da viele kostenlose Compiler-Versionen im Internet auf den Download warten. Alle Programme in diesem Heft wurden mit dem *Borland C++-Windows-Compiler V. 5.5* erstellt und compiliert, der als Freeware im Internet steht. Über folgenden Link kannst du die ca. 8 MB große Datei direkt herunterladen.

`ftp://ftpd.inprise.com/download/bcppbuilder/freecommandLinetools.exe`

Einfach in die Explorerleiste eingeben und bestätigen (das **`ftp`** am Anfang ist wichtig!), dann sollte sich ein Downloadfenster öffnen. Ist der Link nicht mehr gültig, kannst du die aktuelle Version hier aus dem Netz ziehen:

`http://www.borland.com/bcppbuilder/freecompiler/`

Einen weiteren beliebten C++-Compiler findest du bei *Cygnus*:

`http://sourceware.cygnus.com/cygwin/`

Willst du keinen dieser Compiler nutzen, kannst du über jede Suchmaschine mit den Stichwörtern *C++* und *Compiler* jede Menge weiterer Adressen mit kostenlosen Compilern finden. Für den ersten Anfang empfehle ich allerdings den Borland-Compiler, weil er sehr einfach einzurichten ist.

Compiler einrichten

Bevor wir loslegen und unsere ersten Programme schreiben, muss der Compiler auf dem System eingerichtet werden. Ich erkläre diesen Vorgang anhand des Borland-Compilers.

Nach dem Download hast du eine Datei mit dem Namen FREECOMMANDLINETOOLS.EXE auf dem Rechner. Diese selbstentpackende Datei startest du mit einem Doppelklick, worauf sie sich mehr oder weniger von alleine auf deinem Rechner installiert. Danach sollte er ein neues Verzeichnis namens C:\BORLAND\BCC55 haben. Je nach Version kann sich das natürlich ändern. Die Unterverzeichnisse gliedern sich wie folgt auf:

BIN	Enthält das Compilerprogramm und weitere Hilfsprogramme für den Programmierer
EXAMPLES	Beispielprogramme in C++
HELP	Windows-Hilfe für den Borlandcompiler
INCLUDE	Dateien mit C++-Code, der in Programme eingebunden werden kann.
LIB	Programmbibliotheken für C++

Zusätzlich findest du eine README.TXT-Datei, in der die weiteren Installationsschritte erklärt werden. Die sind auch schnell gemacht. Im Verzeichnis BIN erstellst du eine neue Datei namens BCC32.CFG. Diese Datei muss folgende Zeilen enthalten:

```
-I"c:\Borland\Bcc55\include"
-L"c:\Borland\Bcc55\lib"
```

Den Pfad musst du natürlich an deine Maschine anpassen. Anschließend erstellst du eine Datei mit dem Namen **`ilink32.cfg`**, die folgende Zeile enthalten muss:

```
-L"c:\Borland\Bcc55\lib"
```

Hier musst du ebenfalls den Pfad anpassen.

Jetzt sind wir fast fertig. Um es uns ein wenig bequemer zu machen, legen wir einen direkten Pfad auf unser BIN-Verzeichnis, damit Windows jederzeit den Compiler findet. Du wählst START| AUSFÜHREN und schreibst **`sysedit`**, wodurch sich die Datei AUTOEXEC.BAT öffnet

> unter Windows ME benutzt du die Datei CMDINIT.BAT. Da ich den Compiler nicht unter Windows getestet habe, möchte ich keine Garantie für die Lauffähigkeit abgeben!

Hier fügst du folgende Zeile ein:

```
path=c:\Borland\BCC55\bin
```

Dann startest du deinen Rechner erneut. Jetzt sollte der Compiler überall verfügbar sein. Gibt es in der AUTOEXEC.BAT bereits eine **`path`**-Anweisung, trägst du den Pfad durch ein Semikolon (`;`) getrennt hinter dem letzten Eintrag ein. Neustarten nicht vergessen!

Hat alles geklappt, ist der Compiler einsatzbereit.

Schreiben und Compilieren von Programmen

Der Ordnung halber legst du ein Verzeichnis mit dem Namen PROGRAMME an, das deine Arbeiten enthalten soll – am besten unmittelbar neben dem BIN-Verzeichnis, denn so hast du alles griffbereit nebeneinander.

Willst du ein neues Programm schreiben, benutzt du einen einfachen Texteditor wie z.B. *Notepad*. Programme wie *Wordpad* oder *Word* sind dafür ungeeignet, da sie Formatierungsanweisungen in den Texten hinterlassen, wodurch die Datei für den Compiler unbrauchbar wird. Also öffnest du jeweils eine neue Datei mit *Notepad* und schreibst hier dein Programm. Anschließend speicherst du das fertige Programm mit einem sinnvollen Namen in deinem PROGRAMME-Verzeichnis. Wichtig ist, dass die Datei die Endung .CPP hat – das steht für *C Plus Plus*. Gute Namen sind z.B. HALLO.CPP – das soll unser erstes Programm werden – oder SCHLEIFENTEST.CPP.

Jetzt ist das Programm bereit für den Compiler, der sich hinter dem Namen BCC32 versteckt. Bei anderen Compilern heißt er sicher anders; du findest den Namen in der Begleitlektüre. Der Compiler ist ein Programm für die Eingabeaufforderung im DOS-Fenster. Es gibt keine grafische Oberfläche, um das Programm zu starten – also musst du dich an das triste Schwarz gewöhnen.

Du öffnest ein DOS-Fenster und wechselst mit **`cd <Verzeichnisname>`** in das Verzeichnis mit deinen Programmen. Dann startest du mit BCC32 NAME.CPP den Compiler.

Das bedeutet, dass z.B. der Befehl BCC32 HALLO.CPP den Compiler für unser erstes Programm startet, das wir im nächsten Kapitel sehen. Dieser Öffnungsvorgang ist der Augenblick der Wahrheit: eventuelle Fehler werden offenkundig, indem sich der Compiler lautstark beschwert und die Zeile mit dem gefundenen Fehler zurückgibt. Ist das Programm fehlerfrei, siehst du in deinem Verzeichnis drei neue Dateien: eine EXE-Datei, eine Objekt(.OBJ)-Datei und eine TDS-Datei. Die letzteren sind Überbleibsel seitens des Compilers, der hier Daten zwischengespeichert hat Die EXE-Datei ihrerseits ist das fertige Programm, das voll lauffähig ist. Du startest es, indem du seinen Namen über die Befehlszeile aufrufst. Bleibt zu hoffen, dass es seinen Zweck tatsächlich erfüllt ...

Mein erstes C++-Programm

Seit – zumindest für die Computerwelt – uralten Zeiten heißt das erste Programm eines Lehrbuchs HALLO WELT! Seine Aufgabe ist die Ausgabe dieses Textes auf dem Bildschirm. Nicht gerade aufregend – aber ein erster Schritt. *C++* aber ist anders – es wird dein Leben völlig verändern! Also machen wir eine kleine Revolution. Tippe folgendes Programm ab und sonne dich im Lichte dieser revolutionären Änderung!

```
#include <iostream.h>
// Das Hallo C++ Programm
int main()
{
cout << "Hallo C++!\n";
return 0;
}
```

Kompiliere die Datei und führe sie aus. Nach dem Start der EXE-Datei sollte auf dem Bildschirm erscheinen: **Hallo C++!**

Ist das tatsächlich der Fall? – na dann: herzlichen Glückwunsch! Du hast nicht nur die Ketten der Tradition gesprengt, sondern auch dein erstes *C++*-Programm geschrieben!

Sollte eine Fehlermeldung erscheinen, liegt womöglich ein Tippfehler vor. Einige Compiler verlangen, dass die Zeile **`int main();`** unmittelbar vor der **`main`**-Anweisung steht. Ist das bei deinem Compiler der Fall, fügst du sie ein, und das Problem dürfte gelöst sein. Vermisst der Compiler die Datei IOSTREAM.H, solltest du die Pfadangaben zu den INCLUDE-Dateien auf deinem Compiler überprüfen. Sichere dich, dass dieses Programm tatsächlich läuft – sonst bringen dir die restlichen Programme auch nicht viel ...

Die Anweisung include

Gehen wir unser Beispiel Schritt für Schritt durch:

In der ersten Zeile siehst du die Anweisung **`include`**, die von dem Hashzeichen (**`#`**) eingeleitet wird. Solltest du diese Syntax nicht verstehen, muss ich etwas weiter ausholen.

Das Übersetzen des Programms in Maschinensprache erfolgt genau betrachtet in zwei Schritten.

Bevor der Compiler die Anweisungen übersetzt, wird der sogenannte Präprozessor aufgerufen, der den gesamten Code nach Anweisungen durchsucht, die mit einem Hash beginnen. An solchen Stellen modifiziert der Präprozessor den Code – und leitet ihn erst dann an den Compiler weiter. Die **`Include`**-Anweisung bewirkt, dass der Quellcode der angegebenen Datei eben hier in deinem Programm eingesetzt wird. Die Datei IOSTREAM.H enthält alle Funktionen, die für die Ein- und Ausgabe per Tastatur und Monitor nötig sind. Diese Anweisung wird dir in fast jedem Programm begegnen, das wir schreiben werden.

Kommentare

Die nächste Zeile beginnt mit einem doppelten Slash (**`//`**). So erfährt der Compiler, dass er diese Zeile ignorieren kann – und das ermöglicht die Einfügung von Kommentaren. Alles, was hinter einem solchen doppelten Slash steht, wird nicht ausgeführt.

> Es gibt noch eine Variante für Kommentare – hier benutzt du die Kombination **`/*...*/`** Der Compiler wird dadurch veranlasst, alles zwischen dem **`/*`** und dem **`*/`** zu ignorieren.

Die Funktion main()

Das eigentliche Programm beginnt mit der Funktion **`main()`** in Zeile 3. Diese Funktion ist das Kernstück jedes C++-Programms; sie wird bei seinem Start automatisch aufgerufen. Allgemein ist eine Funktion eine Reihe von Befehlen, die nacheinander abgearbeitet werden.

Normalerweise werden Funktionen von anderen Funktionen aufgerufen. **`main()`** stellt hier eine Ausnahme dar, da diese Funktion automatisch gestartet wird. Jedes C++-Programm muss eine **`main()`**-Funktion haben, da es sonst nicht lauffähig ist. Wie bei jeder anderen Funktion muss auch bei **`main()`** festgelegt werden, welcher Wert zurück gegeben wird. **`main()`** gibt grundsätzlich eine ganze Zahl zurück, was durch **`int`** (integer) festgelegt wird – alles andere ist nicht zulässig.

Der eigentliche Inhalt einer Funktion wird in den nachfolgenden geschweiften Klammern definiert. Jede Funktion wird durch eine sich öffnende Klammer eingeleitet und mit der schließenden Klammer abgeschlossen.

Das Objekt cout

Die eigentliche Funktion dieses Programms wird durch das Objekt **`cout`** realisiert. Das Objekt ist ein Teil der Datei **`iostream.h`**, die wir am Anfang eingebunden haben; es ermöglicht die Ausgabe auf dem Bildschirm. Die Umleitungssymbole **`<<`** zeigen an, was auf den Bildschirm gedruckt werden soll. Möchtest du eine Zeichenfolge ausgeben, also einen String, muss dieser in Anführungszeichen gesetzt werden. Das Sonderzeichen **`\n`** steht für einen Zeilenumbruch nach der Ausgabe – das Programm springt in eine neue Zeile. Wichtig ist das Semikolon am Zeilenende, das das Ende der Anweisung anzeigt. Jede Anweisung in C++ muss mit einem Semikolon abgeschlossen werden, sonst gibt es eine Fehlermeldung.

Die Anweisung return

Die letzte Zeile der Funktion **`main()`** ist im Grunde reine Formsache. Da die Funktion mit einem Rückgabewert in Form einer Ganzzahl definiert wurde, muss hier jetzt noch explizit eine **`int`**-Zahl zurückgegeben werden. Das erledigt die Funktion **`return`**. Da es in diesem Fall gleichgültig ist, was **`main()`** zurück gibt, schreiben wir einfach **`return 0`**

Auch diese Anweisung wird mit einem Semikolon beendet. Danach wird die **`main()`**-Funktion mit der geschweiften Klammer abgeschlossen. Das Programm ist fertig.

Funktionen verwenden

Funktionen sind fast das Wichtigste in einem C++-Programm, denn hier wird dem Rechner gesagt was er tun soll. Die Hauptfunktion in jedem Programm ist die Funktion **`main()`**, die wir schon in unserem ersten Programm kennen gelernt haben. Sie nimmt eine Sonderstellung ein, denn sie wird beim Start jedes Programms automatisch aufgerufen. Der Programmierer muss sich also nicht darum kümmern, wo der Compiler seinen Einstieg in das Programm findet.

Die **`main()`**-Funktion wird bei ihrer Deklaration mit der Art des Rückgabewertes initialisiert. Das Kürzel **`int`** vor dem Aufruf steht für integre Zahlen, also Ganzzahlen, und sagt dem Compiler, dass diese Funktion eine Integerzahl zurückgibt. Das ist festgelegt und nicht zu ändern.

Nach der Deklaration der Funktion durch Name und Rückgabewert folgt ein Klammerpaar mit den Parametern. Die Funktion **main()** hat keine Parameter, also sind die Klammern leer. Es folgt der Hauptteil der Funktion, der angibt was zu tun ist. Die Befehle stehen in Schweifklammern. Die folgende Form gilt für alle Funktionen in C++:

```
Rückgabewert
Funktionsname(Parameter1,
Parameter2, Parameter3,...)
{
Befehle
}
```

Neben der Hauptfunktion **main()** kann ein Programm beliebig viele weitere Funktionen enthalten. Meist übernehmen sie spezielle Aufgaben, wodurch das Programm übersichtlich wird. Funktionen liefern oft benötigte Aufgaben als frei zugängliches Werkzeug. Hier addiert die Funktion zwei Zahlen:

```
#include <iostream.h>
// Addition in einer Unterfunktion
int addiere(int a, int b)
{
int c=a+b;
return c;
}
int main()
{
cout << „Geben Sie zwei Zahlen
ein!\n";
int zahl1;
int zahl2;
cin >> zahl1;
cin >> zahl2;
//zwei Zahlen werden eingegeben...

cout << addiere(zahl1, zahl2);
//...und addiert wieder ausgeben!
return 0;
}
```

In den ersten Zeilen wird die Funktion **addiere(int a, int b)** definiert. Sie hat den Rückgabewert einer Ganzzahl und zwei Parameter. Innerhalb der Funktion werden die Parameter addiert und dann zurück gegeben. Diese Funktion hat der Rechner jetzt quasi im Hinterkopf, wenn er zur **main()**-Funktion kommt, denn bis jetzt ist noch nichts passiert! Die Funktion **main()** beginnt mit der Deklaration zweier Variablen, die auf die Eingabewerte gesetzt werden. In diesem Zusammenhang lernen wir auch einen neuen Befehl kennen, mit dem wir Daten von der Eingabeaufforderung auslesen können: **cin** funktioniert genau umgekehrt wie **cout**. Der Eingabewert wird mit ENTER bestätigt und in der angegebenen Variable gespeichert.

Der wichtige Aufruf im Programm steht in der vorletzten Zeile: hier wird der Rückgabewert der Unterfunktion **addiere()** an den Befehl **cout** weitergegeben und auf den Bildschirm gedruckt. Erst hier wird der Code vom Rechner ausgeführt.

Funktionen mit void

Im letzten Kapitel haben wir die Nützlichkeit von Funktionen kennen gelernt. Dabei wurde auch erwähnt, dass bei einer Funktionsdefinition der Rückgabewert angegeben werden muss. In der Funktion selber geschieht das mit dem Befehl **return** gefolgt von einem Wert.

Willst du allerdings eine Funktion schreiben, die keinen Wert zurückgeben muss – oder darf –, muss das mit dem Schlüsselwort **void** kenntlich gemacht werden. Anstelle des Datentyps (wie **int**), wird **void** geschrieben und damit auf eine Rückgabe eines Wertes verzichtet:

```
void funktion()
{
Befehle...
}
```

> Funktionen, die als void deklariert sind, brauchen keinen **return**-Befehl!

Datentypen

Es gibt verschiedene Datentypen innerhalb eines Rechners. Um dem gerecht zu werden, hat C++ eine Datensystematik, in der festgelegt ist, welcher Typ welche Werte annehmen kann. Einen dieser Typen kennen wir bereits: INT !

Willst du also in einem Programm mit Ganzzahlen arbeiten, musst du das dem Rechner mitteilen. Normalerweise erfolgt das bei der Definition von Variablen oder Funktionen. In unserem Additionsprogramm haben wir zwei Variablen als integre Werte definiert, nämlich **`zahl1`** und **`zahl2`**. Also wusste der Rechner, dass wir mit Ganzzahlen rechnen wollen. Wollten wir das Programm auf Kommazahlen erweitern, benötigten wir einen anderen Typ, nämlich **`float`** oder **`double`**. Ersetzt du alle **`int`**-Werte des Programms – natürlich mit Ausnahme von **`main()`** – durch **`float`**, kannst du auch Kommazahlen addieren.

C++ unterstützt die amerikanische Schreibweise von Kommazahlen, d.h. es wird ein Punkt (`.`) anstelle eines Kommas (`,`) zwischen die Ganzzahl und die Nachkommastellen gesetzt. Eingaben wie **`3,45`** bewirken Abstürze oder seltsame Ergebnisse – also: **`3.45`**

Neben der Charakteristik des *Wertes* bestimmt der *Typ* einer Variable, wie viel Speicher für diese Daten im Arbeitsspeicher reserviert werden müssen. Auch wenn es heute lächerlich scheint um ein paar Byte zu feilschen, sollte man den Speicherbedarf des Programms so gering wie möglich halten. Weißt du, dass du nur mit kleinen Ganzzahlen rechnest, solltest du auf eine Deklaration mit **`float`** oder **`double`** verzichten, weil dadurch doppelt soviel Speicher belegt würde.

Tabelle der Datentypen

Der Speicherbedarf von Variablen richtet sich in erster Linie nach dem Wertebereich, den dieser Datentyp abdecken muss. C++ ist bei der Verwaltung dieser Wertebereiche flexibel und erlaubt es, diese nach den jeweiligen Bedürfnissen auszurichten. So ist es möglich, negative Zahlen auszuschließen oder den Wertebereich für besonders große Zahlen anzulegen. Diese Anpassung erfolgt über die spezielle Variationen der Typen. Die Tabelle zeigt, welche Möglichkeiten man bei der Definition von Daten hat. Der Typ **`char`** hat hierbei eine Sonderrolle, auf die wir später genauer eingehen werden.

Typ	Speicherbedarf	Wertebereich
`short int` (kleine **`int`**-Werte)	2 Byte	-32768 bis +32767
`long int` (große **`int`**-Werte)	4 Byte	-2147483648 bis + 2147483647
`unsigned short int` (kleine positive **`int`**-Werte)	2 Byte	0 bis 65535
`unsigned long int` (große positive **`int`**-Werte)	4 Byte	0 bis 4294967295
`char` (ein Buchstabe oder Zeichen)	1 Byte	256 Zeichen
`float` (kleine Kommazahlen)	4 Byte	1.2e-38 bis 3.4e38
`double` (große Kommazahlen)	8 Byte	2.2e-308 bis 1.8e308

Variablen in C++

Wie in jeder Programmiersprache dienen Variablen dazu, Werte im Programm zu speichern. So sind wir in der Lage jederzeit auf die Information zuzugreifen und damit zu arbeiten. Wollen wir eine Variable erzeugen, müssen wir zuvor ihren Typ festlegen. Wollen wir etwa kleine positive Ganzzahlen speichern, ist **`unsigned short int`** das richtige.

Jetzt benötigt die Variable noch einen Namen, anhand dessen wir sie im Programm identifizieren können. Zulässig ist jede Art von Bezeichnung innerhalb von C++, die keine Sonderzeichen enthält und kein C++-Schlüsselwort ist.

Es empfiehlt sich, einen sinnvollen Namen für ein Programm zu wählen, der auf den Inhalt schließen lässt, damit keine Verwirrung entsteht. Namen wie C25FK sind zwar exotisch aber wenig hilfreich, wenn du ein Programm später wieder anschaust.

Wichtig ist zu erwähnen, dass C++ zwischen Groß- und Kleinschreibung unterscheidet. Die Variable **`Umfang`** ist also nicht gleich der Variablen **`umfang`**, wie das mancher vielleicht von Basic gewöhnt sein mag.

Um unsere Variable zu definieren, genügt der Aufruf des Typs gefolgt vom Namen:

```
unsigned short int Umfang;
      // das Semikolon nicht
vergessen!
```

Wollen wir der Variablen einen Wert zuweisen, benutzen wir den **`=`**-Operator:

```
Umfang = 5;
```

Diese beiden Schritte können auch in einem zusammengefasst werden:

```
unsigned short int Umfang = 5;
```

Konstanten definieren

Werte, die sich im Laufe des Programms nicht verändern, sollten als *Konstante* definiert werden, und zwar mit dem Schlüsselwort **`const`**, das vor die Definition gestellt wird:

```
const unsigned short int Umfang=5;
```

Sollte die Variable im Programm nun geändert werden, produziert der Compiler eine Fehlermeldung. Das ist nützlich, wenn du Fehlern auf die Schliche kommen und dein Programm sicherer machen möchtest.

Aufzählungskonstanten mit enum

Mit *Aufzählungskonstanten* kann man eine ganze Reihe von Konstanten erzeugen, die einen Bereich von Werten wiederspiegeln. Das geschieht mit dem Ausdruck **`enum`**:

```
enum Ganzzahlen {a,b,c,d,e,f,g,h,i};
```

Der Name der Aufzählung ist in diesem Fall **`Ganzzahlen`**. Danach folgt in geschweiften Klammern eine Reihe von Konstanten, die automatisch mit Werten belegt werden. Ist nichts weiteres angegeben, hat **`a`** jetzt den Wert **`0`**, **`b`** den Wert **`1`**, **`c`** den Wert **`2`** usw. Willst du die Konstanten mit anderen Werten belegen, tust du das innerhalb der Schweifklammern. Die darauf folgenden Konstanten werden dann jeweils um 1 erhöht:

```
enum Ganzzahlen {a=100, b, c=200 ,d
,e ,f =500, g, h, i =1000};
```

Mit dieser Anweisung hat **`a`** den Wert **`100`**, **`b`** hat den Wert **`101`**, **`c`** den Wert **`200`**, **`d`** gibt **`201`** zurück, **`e`** den Wert **`202`** usw. Jede festgelegte Konstante ist die Basis für die nachfolgenden Konstanten, die dann jeweils um einen Wert inkrementiert werden.

Schlüsselwörter

C++ hat bestimmte Wörter für den Compiler reserviert. Diese Ausdrücke dürfen nicht als Namen für Variablen oder Funktionen benutzt werden. Es handelt sich hier um *Schlüsselwörter*, mit denen das Programm gesteuert wird. Dazu gehören z.B. **`-main`**, **`-if`** oder **`class`**. In der Dokumentation des Compilers findest du eine komplette Liste der reservierten Wörter.

Als Faustregel gilt sicher, dass fast kein sinnvoller Variablen- oder Funktionsname ein Schlüsselwort ist. Allerdings bestätigen Ausnahmen immer wieder die Regel. In einem meiner Programme für Klassenfunktionen wollte ich ein Auto simulieren. Dabei hieß die Funktion fürs Bremsen anfänglich **`break()`**, was zu merkwürdigen Fehlermeldungen führte. Es dauerte etwas, bis ich merkte, dass **`break`** ein Schlüsselwort ist. Ein kleines **`s`** löste meine Probleme: aus **`break()`** wurde **`breaks()`** ...

Anweisungen

Wie wir bereits sahen, enden alle Anweisungen in C++ mit einem Semikolon. Doch was sind Anweisungen eigentlich? Allgemein gesprochen sagt eine Anweisung dem Rechner, was er tun soll. Anweisungen steuern also die letztendliche Ausführung eines Programms und sichern, dass etwas mehr oder weniger sinnvolles geschieht. Eine typische und häufige Anweisung ist diese:

```
x = a + b;
```

Im Gegensatz zur Mathematik bedeutet das nicht etwa, dass **`x`** gleich der Summe von **`a`** und **`b`** ist, sondern dass der Variablen **`x`** der Wert der Summe der Variablen **`a`** und **`b`** zugewiesen wird. Diese Zuweisung wird durch das Gleichheitszeichen ermöglicht, das besagt: der linken Seite wird der Wert der rechten zugewiesen. Diese Anweisung muss mit einem Semikolon beendet werden.

C++ beachtet Leerzeichen in einer Anweisung *nicht*. D.h. die Zeilen

```
x=a+b;
```

und

```
x     =     a          +          b;
```

werden exakt gleich interpretiert.

Leerzeichen solltest du immer dann einsetzen, wenn damit der Sourcecode lesbarer wird.

In diesem Beispiel ist allerdings die erste Version vorzuziehen, da die zweite eher verwirrt als hilft.

Operatoren

Im letzten Kapitel haben wir bereits den ersten Operator kennen gelernt. Das Gleichheitszeichen (=) weist den Compiler an, einen Wert einem anderen zuzuweisen. Aber C++ benutzt eine noch ganze Reihe weiterer Operatoren, ohne die wir nicht auskommen würden. Die wichtigsten sind die vier Grundrechenarten, die es uns erlauben elegant Rechnungen zu lösen.

Mathematische Operatoren

Mathematischer Ausdruck	Operator	Beispiel
Addition	`+`	`7 = 2 + 5`
Subtraktion	`-`	`3 = 5 - 2`
Division	`/`	`3 = 15 / 5`
Multiplikation	`*`	`15 = 3 * 5`
Modulo-Operation	`%`	`1 = 26 % 5`

Die ersten vier Operatoren dürften einleuchten. Die *Modulo*-Operation ist allerdings nicht jedem bekannt. Diese 'Rechenart' liefert bei einer Ganzzahlen-Division den Rest zurück. Der Ausdruck `26 % 5` ergibt also `1`, weil `5` in `26` genau 5 mal drinsteckt und danach `1` übrig bleibt (`5 * 5 + 1 = 26`). Dieser Operator eignet sich z.B. gut, um Zahlen auf gerade oder ungerade zu überprüfen. Gibt er bei einer `Modulo 2`-Rechnung `0` zurück, ist die Zahl mit Sicherheit gerade ☺

In unserem Additionsbeispiel haben wir diese Operatoren schon benutzt, ohne auf ihre Bedeutung einzugehen. Das macht klar, wie intuitiv C++ mit ihnen umgehen lässt. Operatoren können beliebig kombiniert werden – so ist z.B. der Ausdruck `erg = 2 + 8 * 5` zulässig, der dem Compiler sagt, dass er das Ergebnis der Rechung `2 + 8 * 5` der Variablen `erg` zuzuweisen soll.

In diesem Zusammenhang stellt sich auch gleich die Frage nach der Rechenreihenfolge von C++. C++ beachtet alle gültigen Rechenregeln, d.h. auch die *Punkt-vor-Strich*-Rechnung. Das obige Beispiel würde `erg` also den Wert `42` zuweisen, da zuerst die Multiplikation ausgeführt wird.

Willst du die Reihenfolge ändern, musst du sie mit Hilfe von Klammern definieren – ganz nach den üblichen Rechenregeln:

```
erg = (2 + 8) * 5
```

Hier wird `erg` auf den Wert `50` gesetzt, da C++ die Klammern vorrangig behandelt.

> Ich empfehle großzügigen Umgang mit Klammern, da das keinen Einfluss auf ein Programm hat, aber alle Rechnungen und Formeln einfacher wie auch übersichtlicher gestaltet.

Inkrementieren und Dekrementieren

Für häufig verwendete Rechnungen hat C++ eine Abkürzungsmöglichkeit, die die Tipparbeit ein wenig erleichtert. In Programmen werden Variablen oft hoch- oder heruntergezählt, um bestimmte Vorgänge zu systematisieren. Besonders bei Schleifen oder bedingten Wiederholungen begegnet man oft solchen Situationen. Anstatt also jedes mal, wenn die Variable `c` um `1` erhöht werden soll, zu schreiben:

```
c = c + 1;
```

kann auch folgende Kurzform benutzt werden:

```
c++;
```

Beide Möglichkeiten erhöhen `c` um den Wert `1`. Wie ich bereits in der Einleitung erwähnte, ist der Name C++ auf eben diese Weise entstanden; er bedeutet schlicht, dass C++ eine – verbesserte oder erhöhte – Weiterentwicklung von C ist.

Natürlich funktioniert das ganze auch umgekehrt: `c--;` erniedrigt den Wert `c` um `1`, hat also die selbe Bedeutung wie `c = c - 1` !

Bedingungen mit if

Eine der wichtigsten Aufgaben von Programmen ist die Bewertung von Daten und eine daraus sich ergebende Entscheidung über das weitere Vorgehen. Dafür sind in einem Programm Bedingungen zu setzen, anhand derer der Rechner dann entscheidet, was zu tun ist.

Diese Bedingungen werden Hilfe von schlichten *wenn...dann...*-Konstruktionen realisiert, die in ihrer einfachsten Form so aussehen:

```
if (Bedingung) {Anweisung;}
```

Ist die Bedingung wahr, sollen die nachfolgenden Anweisungen ausgeführt werden. In den meisten Fällen werden die Bedingungen anhand von Vergleichsoperatoren gesetzt, dank derer der Rechner eindeutig auf wahr oder falsch entscheiden kann.

Vergleichsoperatoren

Vergleich	Operator	Beispiel	Rückgabe
Gleich	`= =`	`10 == 10` `10 == 9`	`TRUE` `FALSE`
Ungleich	`!=`	`10 != 9` `10 != 10`	`TRUE` `FALSE`
Größer als	`>`	`10 > 9` `9 > 10`	`TRUE` `FALSE`
Größer oder gleich	`>=`	`50 >= 50` `49 >= 50`	`TRUE` `FALSE`
Kleiner als	`<`	`10 < 11` `11 < 10`	`TRUE` `FALSE`
Kleiner oder gleich	`<=`	`50 <= 50` `50 <= 49`	`TRUE` `FALSE`

Der Rückgabewert dieser Vergleiche ist entweder **`TRUE`** (wahr) oder **`FALSE`** (falsch). Bei **`TRUE`** wird die folgende Klammer der **`if`**-Bedingung ausgeführt, bei **`FALSE`** nicht. In der Regel ist es so, dass C++ alle Werte außer **`0`** als **`TRUE`** betrachtet und den Wert 0 als **`FALSE`**. Die Bedingung **`if (0) {Anweisungen;}`** ist also immer falsch und die nach ihr folgenden Bedingungen werden niemals ausgeführt. Ein kluger Compiler wird das auch sofort bemängeln und eine entsprechende Warnung ausspucken.

Im folgenden Beispielprogramm vergleichen wir zwei **`int`**-Werte miteinander und geben den größeren wieder aus:

```
#include <iostream.h>
//Zahlenvergleich
int x, y, bigger;

int main() {

cout << „Geben Sie zwei Werte
ein!\n";
cin >> x;
cin >> y;

if (x > y) {cout << „Die größere
Zahl ist: - << x << -\n";}
if (x < y) {cout << „Die größere
Zahl ist: „ << y << „\n";}
if (x == y) {cout << „Beide Zahlen
sind gleich groß!\n";}
return 0;
}
```

Das Programm gibt entweder die größere Zahl zurück oder aber stellt fest, dass beide Zahlen gleich groß sind.

Erweiterte if-Anweisungen mit else

Jetzt gehen wir einen Schritt weiter und bauen einen weiteren Anweisungsblock an unsere **`if`**-Konstruktion an. Dieser Block steht ebenfalls in geschweiften Klammern und soll dann ausgeführt werden, wenn die Bedingung *nicht* wahr ist. Also eine *wenn... dann... wenn nicht... dann...*-Konstruktion. Das passiert mit der Anweisung **`else`**:

```
if(Bedingung){Anweisungen;}
else{Anweisungen;}
```

Ist die Bedingung nicht erfüllt, werden die Anweisungen in der zweiten geschweiften Klammer ausgeführt. Bedingungen können beliebig ineinander verschachtelt werden, d.h. innerhalb einer Anweisungsklammer können weitere Anweisungen stehen.

Mit diesen Voraussetzungen können wir unser obiges Beispiel zum Zahlenvergleich auch anders schreiben:

```
#include <iostream.h>

int x, y, bigger;

int main() {

cout << „Geben Sie zwei Werte
ein!\n";
cin >> x;
cin >> y;

if (x > y) {cout << „Die größere
Zahl ist: - << x << -\n";}
else
{if (x < y) {cout << „Die größere
Zahl ist: - << y << -\n";}
            else { cout << „Beide
Zahlen sind gleich groß!\n";}
}
return 0;
}
```

Beide Programme bewirken dasselbe. Welches das bessere ist, mag jeder selber entscheiden. Bei verschachtelten Bedingungen musst du allerdings die geschweiften Klammern im Auge behalten. Hier kann sich schnell ein Fehler einschleichen.

Ein weiterer häufiger Fehler, der viele Anfängern passiert, ist das Verwechseln des Vergleichsoperators **==** mit der Zuweisung **=** . Die Bedingung **`if ( x = 3) {...}`** ist z.B. immer **TRUE**, da eine Zuweisung grundsätzlich **TRUE** zurück gibt. Dies setzt **`x gleich 3`** und führt dann die geschweiften Klammern aus.

Der Fehler ist schwer festzustellen und kann einen tatsächlich in Verzweiflung stürzen - es muss natürlich heißen **`if ( x == 3) {...}`**

Logische Operatoren (AND, OR, NOT)

Oft muss man gleichzeitig mehrere Vergleiche anstellen. Dazu werden verschiedene Operatoren benötigt, die die Verknüpfung von Bedingungen erlauben. Je nach Bedarf verwendet man hier **AND**, **OR** oder **NOT**. Diese Arten der Verknüpfung, die Grundrechenarten der elementaren Logik, gehorchen einfachen Gesetzen.

Werden zwei Bedingungen mit **AND** (und) verknüpft, so ist der gesamte Ausdruck nur dann **TRUE**, wenn beide Bedingungen **TRUE** sind – ganz im Gegensatz zur **OR**-(oder)-Verknüpfung. Um die Verbindung zweier Bedingungen **TRUE** werden zu lassen, reicht es, wenn eine der beiden Bedingungen TRUE ist. Der **NOT**-(nicht)-Operator negiert eine einzelne Bedingung ins Gegenteil. Ist eine Bedingung also **TRUE**, wird sie durch den **NOT**-Operator **FALSE**, und umgekehrt.

Die Operatoren werden im Programm durch folgende Zeichen abgekürzt:

Logische Operatoren

Operator	Zeichen	Beispiel
AND	&&	Bedingung && Bedingung
OR	||	Bedingung || Bedingung
NOT	!	!Bedingung

Im folgenden Beispiel wollen wir die Operatoren genauer unter die Lupe nehmen.

Es sollen die Personalnummer und der Zugangscode eines Users überprüft werden. Nur wenn beide korrekt sind, wird der Zugang gewährt. Der Einfachheit halber sind beide Werte normale **int**-Zahlen:

```
#include <iostream.h>

int pnummer, code;
int personalnummer = 98765432;
int geheimcode = 12345678;

int main() {
cout << „Geben Sie Ihre Personalnummer ein: „;
cin >> pnummer;
cout << „\nGeben Sie Ihren Zugangscode ein: „;
cin >> code;

if ((pnummer == personalnummer) && (code == geheimcode))
      {
      cout << „\nZutritt gewährt!\n“;
      return 0;
      }

if ((pnummer == personalnummer) && !(code == geheimcode))
      {
      cout << „\nFalscher Zugangscode!\n“;
      return 0;
      }
if (!(pnummer == personalnummer) && (code == geheimcode))
      {
      cout << „\nFalsche Personalnummer!\n“;
      return 0;
      }
if (!(pnummer == personalnummer) && !(code == geheimcode))
      {
      cout << „\nFalsche Zugangsdaten!\n“;
      return 0;
      }
}
```

Das Programm ist umständlich programmiert, verdeutlicht aber den Gebrauch von logischen Operatoren. Wichtig sind die Klammern um die **if**-Bedingung. C++ verlangt, dass die gesamte Bedingung mit allen Operatoren von einer Klammer umfasst wird. Ansonsten gibt es eine Fehlermeldung.

Das Programm gibt je nach Eingabe der Userdaten eine Meldung aus, die dem User sagt, wo der Fehler seiner Eingabe liegt. Das wird ermöglicht durch dein Einsatz von verknüpften Bedingungen.

Lokale Variablen in Funktionen

Den Einsatz von Funktionen lernten wir bereits in einem der vorhergehenden Kapitel kennen. Allerdings habe ich dabei ein paar wesentliche Punkte im Bezug auf die Variablen in Funktionen verschwiegen. Wie du, weißt kann man einer Funktion innerhalb des Kopfes Variablen übergeben, die weiter verarbeitet werden können. Außerdem sich weitere Variablen innerhalb der Funktion deklarieren, wie wir das in unserem **addiere**-Programm gezeigt haben. Das besondere an diesen Variablen ist, dass sie nur innerhalb der aktuellen Funktion sichtbar sind. Wird die Funktion verlassen, werden die Variablen zerstört und sind nicht mehr zugänglich. Dazu ein Beispiel:

```
#include <iostream.h>
//Beispielprogramm für lokale Variablen!
void funktion() {
int var = 10;
cout << "Bin in funktion()! var hat den Wert: " << var << " !\n";
}
int main() {

int var = 5;
cout << "Bin in main()! var hat den Wert: " << var << " !\n";
funktion();
cout << "Bin wieder in main()! var hat den Wert: " << var << " !\n";
return 0;
}
```

Nach dem Start deklariert das Programm die Variable **var** mit dem Wert **5**, der ausgegeben wird. Dann wird die Funktion **funktion()** gestartet, die ebenfalls eine Variable mit dem Namen **var** deklariert. Sie bekommt allerdings den Wert **10** zugewiesen, der auch ausgegeben wird. Nach der Rückkehr in die Hauptfunktion **main()** wird ein weiteres mal **var** ausgegeben, und es zeigt sich dass der Wert immer noch **5** ist.

Überraschung? – sicher nicht: die Variable **var** in der Unterfunktion hat nichts mit der Variablen **var** in der Funktion **main()** zu tun. Beide sind lokale, voneinander völlig unabhängige Variablen. Darum werden sie auch getrennt behandelt.

> Die Parameter einer Funktion sind ebenfalls lokale Variable der aufgerufenen Funktion. Die 'Original-Werte' der aufrufenden Funktion werden nicht berührt. Genau das macht den Befehl **return** so wichtig, da er z.Zt. die einzige Möglichkeit ist, manipulierte Werte wieder zurückzugeben.

Diese Sicherheitspolitik von C++ mag auf den ersten Blick ein wenig umständlich erscheinen, ist aber bei genauerer Betrachtung sehr wichtig. Vor allem bei größeren, von mehreren Personen bearbeiteten Projekten ist unabdingbar, dass alle Variablen innerhalb von Funktionen geschützt sind. Es wäre eine tödliche Fehlerquelle, wenn alle Variablen global manipulierbar wären. Wahrscheinlich kämen sich die Entwickler bei der Namensgebung in die Quere, wodurch der Arbeitsaufwand immens anwüchse.

Globale Variablen

Eben habe ich sie verflucht – manchmal braucht man sie aber doch: die globalen Variablen. Diese Variablen können von allen Funktion bearbeitet und beschrieben werden, ohne dass das zu Problemen führt. Diese Freiheit muss manchmal für bestimmte Daten gegeben sein, damit ein Programm realisiert werden kann. Ein Beispiel sind Userdaten, die von jeder Funktion innerhalb eines Programms ausgelesen werden müssen, damit der User identifiziert werden kann.

Willst du globale Variablen schaffen, reicht es, diese außerhalb einer Funktion zu deklarieren. Unser Programm für die Zugangsdaten auf Seite 16 arbeitet beispielsweise mit globalen Variablen. Es empfiehlt sich globale Variablen gleich am Anfang des Programms zu deklarieren, damit man sie im Auge behält:

```
#include <iostream.h>
//Beispielprogramm für globale
Variablen!
int var;
void funktion() {
cout << "Bin in funktion()! var hat
den Wert: " << var << " !\n";
}
int main() {
var = 10;
cout << "Bin in main()! var hat den
Wert: " << var << " !\n";
funktion();
cout << "Bin wieder in main()! var
hat den Wert: " << var << " !\n";
return 0;
}
```

Hier haben wir unser letztes Beispielprogramm ein wenig verändert und **var** global definiert. Jetzt kann der Rechner sowohl in **main()** als auch in allen Unterfunktionen auf die Variable **var** zugreifen, wie die Ausgabe beweist: **var** hat nun grundsätzlich den Wert **10** !

Standardparameter in Funktionen

Wie wir wissen, kann man Funktionen mit Parametern definieren, die dann innerhalb der Funktion verarbeitet werden. Diese Möglichkeit der Datenübergabe ist zwar nützlich, schränkt aber auch ein, weil der Funktionsaufruf nur gültig ist, wenn die vorher festgelegten Parameter auch besetzt wurden. Ist das nicht der Fall, führt das zu einem Compilerfehler. Diese Einschränkung kann man relativ einfach umgehen. C++ erlaubt die Definierung von Standardparametern für eine FunktionStandard, die beim Funktionsaufruf im Falle von fehlenden Parametern automatisch an deren Stelle treten. Dadurch lassen sich Funktionen viel flexibler nutzen – und sie lassen sich bis zu einem gewissen Grad standardisieren.

```
#include <iostream.h>
//Berechnung eines Rechtecks
int rechteck (int x =10, int y=10)
{
int inhalt = x * y;
return inhalt;
}
int main()
{
cout << –Fläche eines 25 * 15 großen
Rechtecks: – << rechteck(25, 15) <<
„\n";
cout << „Fläche des Standardrechecks
(10 * 10): „ << rechteck() << „\n";
return 0;
}
```

Das Programm berechnet zweimal die Größe eines Rechtecks. Im ersten Fall ist es die Größe eines definierten Rechteckes mit dem Ergebnis 375 – hier werden die Parameter übergeben. Im zweiten Fall werden keine Parameter übergeben – hier setzt der Rechner automatisch die Werte der Standardparameter ein, die in diesem Fall zum Ergebnis 100 führen.

Funktionen überladen, Polymorphie

In diesem Kapitel gehen wir einen Schritt weiter, indem wir die Funktionen noch mehr strapazieren. C++ erlaubt die Erzeugung mehrerer Funktionen mit demselben Namen. Der Unterschied liegt dann einzig und allein bei den Parametern, die über die auszuführende Funktion entscheiden.

```
int rechteck(int, int)
int rechteck (int)
float rechteck(float, float)
```

Der Rückgabewert der überladenen Funktion kann der selbe sein oder auch ein anderer. Wesentlich ist, dass die Parameter sich ändern.

Wird die Funktion aufgerufen, entscheidet der Compiler anhand der Art und Menge der Parameter welche Funktion aufgerufen wird. Diese Technik nennt sich Funktionspolymorphie, d.h. Vielgestaltigkeit der Funktion.

```
#include <iostream.h>
//Berechnung eines Rechtecks
int rechteck (int x)
{
int inhalt = x * 10;
return inhalt;
}
float rechteck (float x, float y)
{
float inhalt = x * y;
return inhalt;
}
int main()
{
cout << "Fläche mit einem int-Wert
(12):   " << rechteck(12) << "\n";
cout << "Fläche mit float-Werten
(12.34 17.28):   " <<
rechteck(12.34,17.28) << "\n";
return 0;
}
```

In diesem Beispielprogramm wurde die Funktion **`rechteck()`** überladen. Im ersten Fall wurde nur ein **`int`**-Wert übergeben, im zweiten Fall zwei **`float`**-Werte. Welche Funktion der Rechner benutzen soll, entscheidet er anhand der Parameter in der **`main()`**-Funktion.

Diese Vorgangsweise ist äußerst praktisch, da man sonst für jede Funktion einen eigenen Namen benutzen müsste. So ist man als Programmierer unabhängig und muss nur diese eine Funktion benutzen – alles andere erledigt sich automatisch.