C/C++ Kurzübersicht

überarbeitet von Dr. G.Wöste

Ein Programm

Das erste Programm gibt eine Meldung auf dem Bildschirm aus, fordert eine Zahleneingabe, errechnet das Quadrat hieraus und gibt das Ergebnis auf dem Bildschirm aus.

```
#include <iostream.h>
main()
{
   int Eingabe;
   int Quadrat;
   cout << "Geben Sie eine Zahl ein: ";
   cin >> Eingabe;
   Quadrat = Eingabe * Eingabe;
   cout << "Die Quadratzahl lautet " << Quadrat <<
endl;
}
```

Frage: Was müssen Sie noch ergänzen, um das Programm lauffähig zu gestalten?

Gehen wir das Programm Zeile für Zeile durch.

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

Dieser Befehl liest die Datei **iostream.h** an dieser Stelle in den Quelltext ein. Er wird gebraucht für die Ein- und Ausgabe (iostream).

```
main()
{
```

main(){ }

Der Name `main()` leitet die Hauptfunktion des Programms ein. Jedes C- oder C++-Programm hat genau eine Funktion `main()`. Jede Funktion, also auch `main()`, enthält eine Reihe von Anweisungen, die in geschweiften Klammern stehen.

```
   int Eingabe;
   int Quadrat;
```

int

Die erste Zeile im Funktionsrumpf enthält die erste Anweisung. Jede Anweisung wird durch ein Semikolon abgeschlossen. Diese Anweisung ist eine Variablendefinition. Dass die Variable ganze Zahlen aufnehmen kann, wird durch den Variablentyp `int` signalisiert, der vor dem Variablennamen steht. Das Schlüsselwort `int` steht für Integer, das ist der englische Begriff für ganze Zahlen, also Zahlen ohne Nachkommastellen.

Typen

Neben dem Typ `int` gibt es die Ganzzahlentypen `short` für kleinere Zahlen und `long` für größere.
Weitere Typen:

Typ	Daten
int	Ganze Zahlen
short	Ganze Zahlen (typischerweise --32.768 bis 32.767)
long	Ganze Zahlen (typischerweise --2.147.483.648 bis 2.147.483.647)
float	Fließkommazahlen (beispielsweise --2,5 oder 3,14159)

double	Fließkommazahlen höherer Genauigkeit
long double	Fließkommazahlen mit besonder großer Genauigkeit
char	Buchstaben oder ganze Zahlen von --128 bis 127

```
cout << "Geben Sie eine Zahl ein: ";
```

Frage: Warum gibt es verschiedene Typen für die Variablen?

cout

Bildschirmausgaben werden in C++ auf das Objekt **cout** gelenkt. Das Ausgabeobjekt **cout** steht immer links, es folgen zwei Kleiner-Zeichen, die man als Umleitungsoperator bezeichnet, und dann das, was auf dem Bildschirm erscheinen soll.

```
cin >> Eingabe;
```

cin

Das Gegenstück ist das Eingabeobjekt **cin**. Mit den zwei Größer-Zeichen werden die Daten von der Eingabe in eine Variable umgeleitet.

```
Quadrat = Eingabe * Eingabe;
```

Berechnung

Die Anweisung in der folgenden Zeile berechnet die Quadratzahl. Hier wird das Ergebnis in der Variablen **Quadrat** abgelegt.

Die Tabelle zeigt eine Übersicht der grundlegenden Operatoren.

Operator	Bedeutung
+	Addition
-	Subtraktion
*	Multiplikation
/	Division
=	Zuweisung
<<	Umleitung (typischerweise nach **cout**)
>>	Umleitung (typischerweise von **cin**)

Abkürzungen

Für bestimmte Berechnungen gibt es in C++ Abkürzungen. So können Sie durch die Folge `a+=5` den Inhalt der Variablen **a** um 5 erhöhen. Das ist also identisch zu `a=a+5`. Das funktioniert übrigens auch mit `-=`, `*=` und `/=`. `a++` zählt die Variable **a** um eins herauf.

```
cout << "Die Quadratzahl lautet " << Quadrat << endl;
```

nochmal cout

Nun muss das Ergebnis noch auf dem Bildschirm ausgegeben werden.

Abfrage und Schleifen

Programme müssen nicht nur Zeile für Zeile ablaufen. Es ist möglich, Programmteile aufgrund des Zustands von Variablen zu übergehen oder zu wiederholen.

Abfrage und boolesche Ausdrücke

Mit Hilfe der Abfrage können Sie einen Anweisungsblock unter eine angegebene Bedingung stellen. Beispielsweise können Sie prüfen, ob die Zahl, durch die Sie gerade teilen wollen, vielleicht 0 ist.

```
[Sichern gegen eine Division durch null]
if (divisor==0)
{
   cout << "Division durch 0" << endl;
}
else
{
   quotient = dividend / divisor;
}
```

Frage: Was ändert sich, wenn „**else**“ weggelassen wird?

if else

Die Abfrage beginnt mit dem Schlüsselwort `if`, und in Klammern folgt die Bedingung, unter der die nächste Anweisung bzw. der nachfolgende Block ausgeführt wird. Der nachfolgende Befehl `else` leitet die Anweisung bzw. den Block ein, der ausgeführt wird, wenn die Bedingung nicht zutrifft. Die Verwendung von `else` ist optional, **kann also weggelassen werden**.

Aufgabe: Ergänzen Sie obiges Programm, damit es lauffähig wird.

Vergleiche

Das Symbol für die Gleichheit zweier Werte ist in C++ ein doppeltes Gleichheitszeichen (==), um es von der Zuweisung zu unterscheiden. Das Zeichen für Ungleichheit ist (!=). Sie können zwei Werte mit dem Kleiner-Zeichen (<) und dem Größer-Zeichen (>) prüfen, ob der erste Wert kleiner/größer als der zweite Wert ist. Durch Anhängen eines Gleichheitszeichens kann auf kleiner oder gleich (<=) bzw. größer oder gleich (>=) getestet werden.

```
[Programmausschnitt]
if (Eingabe>=5 && Eingabe<=10)
```

Frage: Welche Zahlen müssen für die Eingabe gelten, damit die Bedingung wahr ist?

Und-Verknüpfung

Zwei kaufmännische Und-Zeichen (&&) bewirken die Und-Verknüpfung zweier logischer Ausdrücke. Mit der Abfrage wird also geprüft, ob die Eingabe zwischen fünf und zehn liegt.

Der Gesamtausdruck einer Und-Verknüpfung wird wahr, wenn beide Teilausdrücke wahr sind.

Aufgabe: Nennen Sie ein sinnvolles Beispiel für die UND-Verknüpfung.

Oder-Verknüpfung

Die Oder-Verknüpfung wird in C++ durch zwei senkrechte Striche (||) dargestellt. Es dürfen allerdings auch beide wahr sein. Das Oder ist also **kein** »Entweder-Oder«.

Der Gesamtausdruck einer Oder-Verknüpfung wird wahr, wenn mindestens einer der Teilausdrücke wahr ist.

Aufgabe: Nennen Sie ein sinnvolles Beispiel für die ODER-Verknüpfung.

Negation

Die Negation eines logischen Ausdrucks wird in C++ durch das Ausrufezeichen dargestellt. Wollen Sie also ausdrücken, dass die Eingabe nicht zwischen 5 und 10 liegen soll, schreiben Sie dies einfach so:

```
[Programmausschnitt]
if (!(Eingabe>=5 && Eingabe<=10))
```

Die Klammer hinter dem Ausrufezeichen bewirkt, dass der gesamte Ausdruck negiert wird!

Zusammenfassend zeigt die Tabelle die logischen Operatoren.

Operator	Bedeutung
==	Gleichheit
!=	Ungleichheit
<	Kleiner
<=	Kleiner oder gleich
>	Größer
>=	Größer oder gleich
!	Negation eines booleschen Ausdrucks (NOT)
&&	Und-Verknüpfung (AND)
||	Oder-Verknüpfung (OR)

Die while-Schleife

In Schleifen kann eine Anweisung oder ein Block von Anweisungen so lange wiederholt werden, wie die Schleifenbedingung zutrifft.

```
#include <iostream.h>

main()
{
   int InZahl = 0;
   while (InZahl<1 || InZahl>6)
   {
      cout << "Geben Sie eine Zahl ein!";
      cin >> InZahl;
   }
}
```

Frage: Was müssen Sie noch ergänzen, um das Programm lauffähig zu gestalten?

Frage: Was macht das Programm?

Schleifenbedingung

Das Programm wiederholt so lange die `while`-Schleife, wie die Variable **InZahl** kleiner als 1 oder größer als 6 ist. Anders ausgedrückt: Das Programm verlässt die Schleife, wenn der Wert von **InZahl** zwischen 1 und 6 liegt.

Schleifenkörper

Innerhalb der Schleife wird die Anweisung gegeben, eine Zahl zwischen 1 und 6 einzugeben. Daraufhin kann der Benutzer eine Zahl eingeben, die in der Variablen **InZahl** gespeichert wird.

Initialisierung

Wichtig ist auch, was vor der Schleife passiert. Durch die Zuweisung von 0 an die Variable **InZahl** ist gewährleistet, dass die Schleife überhaupt betreten wird. Sollte die Variable **InZahl** bereits vor der Schleife einen Wert zwischen 1 und 6 haben, dann würde die Schleife gar nicht erst betreten.

Die for-Schleife

Die `for`-Schleife wird besonders dann verwendet, wenn Dinge abgezählt werden.

Das folgende Programm erstellt eine Liste der ersten zehn Quadratzahlen auf dem Bildschirm.

```
#include <iostream.h>
main()
{
   int i;
   for (i=1; i<=10; i++)
   {
      cout << i << ": " << i*i << endl;
   }
}
```

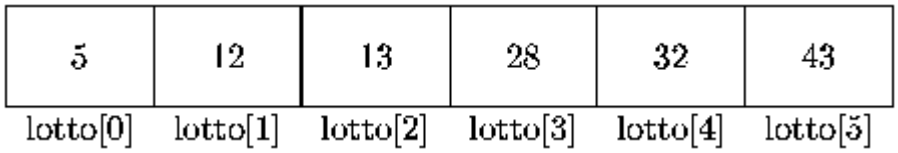
Klammerinhalt

In der `for`-Klammer stehen drei Elemente, jeweils durch ein Semikolon getrennt. Das erste Element ist die Initialisierungsanweisung. Sie wird **genau einmal** vor dem Betreten der Schleife ausgeführt. Das zweite Element ist die Schleifenbedingung. Sie überprüft, ob die Zählvariable noch unter dem Limit liegt. Das dritte Element ist die Schlussanweisung. Sie wird in **jedem** Durchgang nach Ausführung des Schleifenrumpfes ausgeführt.

Arrays

Ein Array ist ein Datenverbund mehrerer gleichartiger Variablen. Dabei stehen die Variablen in einer Reihe direkt nebeneinander. Um ein einzelnes Element eines Arrays anzusprechen, wird die Positionsnummer verwendet.

Ein typisches Beispiel für ein Array sind die Lottozahlen. Damit ist **lotto[0]** die erste Lottozahl und **lotto[3]** die vierte.



Im folgenden Listing wird eine Array-Variable für die Lottozahlen definiert und anschließend werden die Zahlen in einer `for`-Schleife ausgegeben.

```
[Lottozahlen]
#include <iostream.h>

main()
{
   int lotto[6];
   lotto[0] = 5;
   lotto[1] = 12;
   lotto[2] = 13;
   lotto[3] = 28;
   lotto[4] = 32;
   lotto[5] = 43;
   int i;
   for (i=0; i<6; i++)
   {
      cout << lotto[i] << "  ";
   }
   cout << endl;
}
```

Definition

Die Definition eines Arrays ist der Definition einer Variablen recht ähnlich.

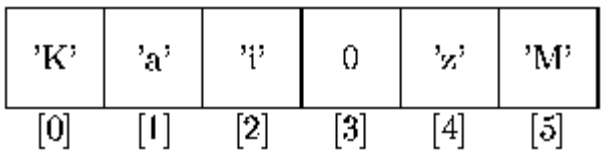
Frage: Wie wird ein Array definiert?

Zeichenketten

Das geläufigste Array besteht aus Zeichen des Datentyps `char`. Ein einzelner Buchstabe belegt ein Byte. Nebeneinander gestellt wird ein Text daraus. Das Ende des Textes wird dadurch signalisiert, dass das **Byte 0,** nicht der **Buchstabe '0'** verwechselt werden.

Frage: Warum wird das Ende des Textes überhaupt markiert? Warum nicht durch den Buchstaben 0?

Die Grafik in der folgenden Abbildung zeigt ein Array von `char`, in dessen ersten drei Elementen sich die Buchstaben 'K', 'a' und 'i' befinden. Im vierten Element mit dem Index 3 befindet sich eine Null. Der Zustand der restlichen zwei Elemente ist unbestimmt.



[Gefülltes char-Array]

Das nachfolgende Programm verdeutlicht, wie Sie Texte eingeben können. Dazu wird die Funktion `cin.getline()` verwendet.

```
[Begrüßung (gruss.cpp)]
#include <iostream.h>

main()
{
   char Name[80];
   cout << "Wie heißen Sie?" << endl;
   cin.getline(Name, 80);
   cout << "Guten Tag, " << Name << "!" << endl;
   char Kopie[80];
   int i;
   for (i=0; Name[i] && i<79; i++)
   {
      Kopie[i] = Name[i];
   }
```

```
   Kopie[i] = 0;
   cout << "Das ist Ihre Kopie: " << Kopie << endl;
}
```

Frage: Was müssen Sie noch ergänzen, um das Programm lauffähig zu gestalten?
Frage: Was macht dieses Programm?

Zu Anfang wird wieder die Datei **iostream.h** eingebunden. Dann beginnt die Funktion `main()`.

```
char Name[80];
```

Für den einzugebenden Namen wird eine Array-Variable namens **Name** angelegt, die bis zu 80 Zeichen aufnehmen kann. Da aber jede Zeichenkette ihre Abschluss-Null benötigt, stehen netto 79 Buchstaben zur Verfügung.

```
cout << "Wie heißen Sie?" << endl;
cin.getline(Name, 80);
```

Zeileneingabe

Zunächst gibt das Programm eine Meldung auf dem Bildschirm aus, in der es den Benutzer auffordert, seinen Namen einzugeben. In der folgenden Zeile wird eine Eingabezeile entgegengenommen und in der Variablen **Name** abgelegt. Es wird wieder das Eingabeobjekt **cin** verwendet.

Frage: Warum wird die Methode getline benutzt?

```
cout << "Guten Tag, " << Name << "!" << endl;
```

Nun wird die Begrüßung ausgegeben. Der eben eingegebene Name wird in diese Zeile eingefügt.

Frage: Woran sieht man, dass „Name“ eine Variable ist?

```
char Kopie[80];
int i;
```

Hier werden zwei Variablen definiert. Die erste ist ein Array, das exakt so definiert wurde wie der Name. Als Zweites wird die Integer-Variable **i** definiert, die als Zähler und Index benutzt wird.

```
for (i=0; Name[i] && i<79; i++)
```

Aufgabe: Erklären Sie den Kopf dieser For-Schleife!

Da jede Zeichenkette aber mit einer 0 endet und 0 in C++ als falsch interpretiert wird, endet die Schleife sofort, wenn im Original das Ende der Zeichenkette erkannt wird. Durch die Und-Verknüpfung reicht es aus, dass ein Teil falsch ist und die Gesamtbedingung für das Verbleiben in der Schleife nicht mehr erfüllt ist.

```
{
   Kopie[i] = Name[i];
}
```

Die geschweiften Klammern gehören zur `for`-Schleife und umschließen die Anweisungen. Hier wird der **i**-te Buchstabe des Arrays **Name** in das entsprechende Element des Arrays **Kopie** kopiert.

```
Kopie[i] = '\0';
```

Abschluss setzen

Frage: Warum muss dieser Abschluss gesetzt werden?

In dieser Zeile wird die Abschluss-Null in der Kopie gesetzt. Die Schleife kopiert nur den Inhalt. Wird die Null des Originals erreicht, bricht die Schleife ab, und die Kopie hat die Null nicht übernommen. Dies wird an dieser Stelle nach der Schleife nachgeholt. Der Index **i** ist nach der Schleife um eins höher als die letzte Kopie.

Funktionen

Mit einer Funktion können Sie mehrere Anweisungen so zusammenfassen, dass sie von beliebiger Stelle im Programm durch einen einzigen Befehl aufzurufen sind. Dadurch werden größere Programme übersichtlicher. Befindet sich in der Funktion eine Sequenz von Anweisungen, die öfter benötigt wird, wird das Programm darüber hinaus auch kürzer.

Programmaufteilung

Als Beispiel wird das vorige Programm in drei Funktionen zerlegt. Die erste Funktion nimmt die Eingabe des Benutzers entgegen, die zweite kopiert den Namen, und die dritte gibt die Meldung aus.

```
[Begrüßung in Funktionen aufgeteilt (fgruss.cpp)]
#include <iostream.h>

char Name[80];
char Kopie[80];

void EingabeName()
{
   cout << "Wie heißen Sie?" << endl;
   cin.getline(Name, 80);
   cout << "Guten Tag, " << Name << "!" << endl;
}

void Kopiere()
{
   int i;
   for (i=0; Name[i] && i<79; i++)
   {
      Kopie[i] = Name[i];
   }
   Kopie[i] = 0;
}

void ZeigeKopie()
{
   cout << "Das ist Ihre Kopie: " << Kopie << endl;
}

main()
{
   EingabeName();
   Kopiere();
   ZeigeKopie();
}
```

Aufgabe: Erklären Sie den den Programmablauf!

Die Anweisungen haben sich nicht verändert. Der Inhalt der Funktion `main()` ist wesentlich übersichtlicher geworden. Betrachten wir wieder die Anweisungen im Einzelnen:

```
char Name[80];
char Kopie[80];
```

Globale Arrays

Die beiden Array-Variablen sind nun global. Da sie zu Anfang des Programms **außerhalb** jeder Funktion definiert werden, können Sie von **jeder** Stelle des Programms aus auf sie zugreifen.

```
void EingabeName()
{
   cout << "Wie heißen Sie?" << endl;
   cin.getline(Name, 80);
   cout << "Guten Tag, " << Name << "!" << endl;
}
```

Das ist die Funktion `EingabeName()`. Der Funktionsname folgt den gleichen Regeln wie der Name einer Variablen. Vor dem Funktionsnamen steht immer der Typ des Rückgabewerts. Da diese Funktion gar keinen Rückgabewert hat, ist dieser Typ `void` (englisch: »leer« oder »unbesetzt«). Das Klammerpaar enthält die Parameter. Diese Funktion hat keine Parameter, also ist die Klammer leer.

```
void Kopiere()
{
   int i;
   for (i=0; Name[i] && i<79; i++)
   {
      Kopie[i] = Name[i];
   }
   Kopie[i] = 0;
}
```

Die Funktion `Kopiere()` ist ebenfalls eine Funktion ohne Rückgabewert und Parameter.

Die letzte Funktion bringt nicht viel Neues, darum wenden wir uns gleich der Funktion `main()` zu.

```
main()
{
   EingabeName();
   Kopiere();
   ZeigeKopie();
}
```

Der Aufruf der Funktionen kann beliebig oft an beliebigen Stellen des Programms erfolgen.

Frage: Warum werden Funktionen verwendet?

Ist es möglich, Funktionen zu bilden, die im Programm mehrfach anwendbar ist, ersparen Sie sich viel Tipparbeit. Hinzu kommt, dass Sie die Funktion beim zweiten Mal nicht noch einmal testen müssen, da sie ja bereits getestet ist.

Rückgabewert

Im Beispiel benötigten die Funktionen keinen Rückgabewert. In vielen Fällen soll aber die Funktion Informationen an den Aufrufer zurückliefern. Das kann eine Berechnung, aber auch eine Fehlermeldung sein.

```
float zaehler, nenner;

float Teilen()
{
   return zaehler/nenner;
}

main()
{
   float quotient;
   zaehler = 26;
   nenner = 4;
   quotient = Teilen();
}
```

Aufgabe: Erklären Sie die Funktionsweise des Programms.

```
float Teilen()
```

Der Rückgabewert der Funktion `Teilen()` ist vom Typ `float`. Dieser Rückgabetyp einer Funktion steht immer am Anfang einer Funktionsdefinition. Es folgt der Funktionsname und die bereits bekannten Klammern. Der Rückgabewert wird durch den Befehl `return` an den Aufrufer zurückgegeben.

```
return zaehler/nenner;
```

Der Befehl `return` beendet eine Funktion sofort. Besitzt sie einen Rückgabewert, muss hinter dem `return` stehen, was die Funktion an den Aufrufer als Ergebnis liefert. Im Beispiel ist der Rückgabewert die Division der Variablen **zaehler** und **nenner**.

```
main()
{
   ...
   zaehler = 26;
   nenner = 4;

}
```

Frage: Warum können Sie zaehler und nenner im Hauptprogramm verwenden, obwohl diese Variablen dort nicht deklariert wurden?

```
quotient = Teilen();
```

In dieser Zeile enthält die Variable **quotient** den Wert, den der Befehl `return` in der Funktion zurückgibt.

Frage: Warum müssen Klammern hinter dem Funktionsaufruf **Teilen** gesetzt werden?

Parameter

Über die Parameter werden Werte an die Funktion übermittelt.

```
[Parameterübergabe]
float Teilen(float zaehler, float nenner)
{
   return zaehler/nenner;
}

main()
{
   float quotient;
   myfloat=26;
   quotient = Teilen(myfloat, 4);
}
```

Frage: Warum werden keine globalen Variablen benötigt?

```
float Teilen(float zaehler, float nenner)
```

Die Parameterdeklarationen zwischen den Klammern sind lokale Variablendefinitionen, die durch Kommata getrennt werden. Die Werte, die beim Funktionsaufruf zwischen den Funktionsklammern stehen, werden in diese Variablen kopiert.

```
quotient = Teilen(myfloat, 4);
```

Die Funktion `Teilen()` kann auf den Wert der Variablen **myfloat** über die lokale Variable **zaehler** zugreifen.

Ende