C++ Grundlagen

Generiert als „Best of“, siehe z.B. hier: **http://www.onlinetutorials.de/cpp-index.htm**

Der Inhalt gibt einen grundlegenden Überblick von C++ mit vielen Beispielen.

10.

Kapitel 1

Die Hauptfunktion

Das erste Programm

```
int main(void)
{
   return 0;
}
```

Das ist die main-Funktion - der Code darin wird bei einem DOS-Programm zuerst ausgeführt. Es öffnet sich ein Fenster - unter Windows also eine MS-DOS Eingabeaufforderung - und schließt sich auch gleich wieder. Es fehlt an Inhalt!

Die Wörter

int und **void** sind Schlüsselwörter von C++, das heißt, dass es zum Sprachumfang von C++ gehört. Schlüsselwörter kann man in jedem Programm benutzen. **void** heißt ins Deutsche übersetzt "nichts". **int** ist ein Datentyp, der Ganzzahlen speichert.

main ist kein Schlüsselwort - es stellt die Hauptfunktion eines jeden C++Programms dar. Schreibst du diesen Namen falsch, gibt C++ eine Fehlermeldung aus. Merke dir:

C++ (und auch C) sind Case-sensitive, sie unterscheiden Groß- und Kleinschreibung, beispielsweise wäre **void** nicht Void oder VoID und main nicht MAIN oder mAIn.

Falsch wäre also: Int Main(Void) und INT MAIN(VOID).

return 0 sollte in jede main-Funktion rein, die dem C++ Standard folgt. **return** ist ein weiteres Schlüsselwort und gibt einen Wert von der Funktion zurück - bei der main-Funktion normalerweise 0. Wenn ein **return** vor ein paar anderen Anweisungen im Code steht, werden diese nicht mehr ausgeführt, weil es sozusagen die Kontrolle an die vorherige Funktion übergibt. (Im Programm ganz oben ist **return** 0; die erste und einzige Anweisung, die die main-Funktion ausführt.)

Danach steht ein Semikolon - es drückt aus, dass ein Befehl zuende ist. Wie in einer realen Sprache musst du Satzzeichen setzen, sonst kann der Sinn des Satzes verfälscht werden. Bei **return** 0; beendet es also den **return**-Befehl.

Syntax der main-Funktion

Wie schon erwähnt, ist main() die Hauptfunktion. Dabei handelt es sich um den für Funktionen typischer Syntax:

```
Rückgabetyp Funktionsname ( Parameter )
```

Doch dazu berichte ich in einem späteren Kapitel detaillierter. Der C++ Standard schreibt für die main-Funktion entweder den Syntax von oben oder einen Syntax, den ich später auch noch zeigen werde, vor.

Die geschweiften Klammern deuten an, dass darin der Code für die `main`-Funktion steht - derzeit nur die Rückgabe mit **`return`**. Jedes C++ Programm muss eine `main`-Funktion enthalten, also darfst du die Hauptfunktion auch nicht umbenennen (bzw. falsch schreiben).

Im Kapitel über Funktionen wirst du noch eine zweite Variante sehen, der Vollständigkeit halber kurz mal gezeigt:

```
int main(int argc, char *argv[])
{
   return 0;
}
```

Kommentare

Selbe Funktion, nur mit etwas mehr Inhalt. Der ist allerdings nicht für den Compiler interessant:

Kommentare

In Zeile 4 wird ein einzeiliger Kommentar verwendet. Ein Kommentar kannst du nach jedem Befehl gebrauchen, es geht jedoch auch in einer Extrazeile. Der Compiler ignoriert es, denn es dient dem Programmierer als Notiz.

`//` - einzeilig
`/*` - Anfang eines mehrzeiligen
`*/` - Ende eines mehrzeiligen

Kommentare kannst du fast überall im Code ablassen.

Verwendungszwecke:

1.)

Kommentare dienen natürlich der Verständlichkeit und Lesbarkeit des Codes. Allerdings immer nur in dem Maße, dass sie entweder ans Ende der Zeile oder in einen eigenen Absatz kommen. Du kannst zwar davon ausgehen, dass man deinen Code auch ohne Kommentare lesen und verstehen kann, dann ist aber deine Absicht für den Leser wahrscheinlich nicht klar und er wird wesentlich länger brauchen als mit angebrachten Kommentaren.

Was zu kommentieren ist und was nicht, wird beim Codieren meistens erkenntlich. Korrektes Kommentieren lässt sich recht einfach lernen (wenn das überhaupt nötig ist). Hier ein paar Regeln für "gutes" kommentieren (du solltest später noch einmal hierher kommen, damit du weißt, was die Begriffe bedeuten):

- selten oder temporär gebrauchte Variablen mit einem Kommentar erkenntlich machen
- bei Funktionsprototypen die Verwendung der Rückgabewerte und Parameter nennen
- größere Sinnabschnitte gliedern und mit Kommentaren beginnen lassen
- Module einen Anfangskommentar verpassen ("Comment Header")
- Fehler mit einem langen `/////////` markieren
- bei Klassen und Funktionen den Programmierer und den Programmierstatus angeben
- Kommentare immer auf dem aktuellsten Stand bringen

Nicht zu empfehlen ist übermäßiges Kommentieren - einfache Anweisungen sind unkommentiert zu lassen! Die Übersichtlichkeit leidet enorm, wenn du jeder Zeile (oder jeder zweiten oder dritten) ein Kommentar anhängst.

2.)

Nicht zu vergessen und sehr nützlich ist, Befehle von Compilern auszuschließen und somit Code-Fragmente zu "konservieren". Ein Stückchen alter Code kann also weiterhin neben einem neuen alternativen Code stehen. Alte Ideen bleiben erhalten und können einem neuen Lösungsweg möglicherweise helfen.

Ein Fallstrick bei Kommentaren ist allerdings, mehrzeilige Kommentare zu schachteln:

```
int main(void)
{
   /*außen /*innen*/ (wieder außen - Fehler)*/
   //...
}
```

Der Kommentar endet schon nach "innen"! Auch ein simples */ erzeugt einen Fehler! Du musst also auf sowas achten, vor allem bei auskommentiertem Code.

Das Ausgabe-Objekt cout

Ausgabe mit cout - cout1

Beim Ausführen siehst du kurz das DOS-Fenster, und kannst vielleicht sogar erkennen, dass da was geschrieben steht, nämlich "Ausgabe mit cout". Du solltest jetzt mal die MS DOS Eingabeaufforderung verwenden, damit du den Inhalt nachlesen kannst (falls dieser nicht klar sein sollte). Der cout-Befehl gibt Text auf den Bildschirm aus - hier Strings (Zeichenketten), später noch anderes. Diese müssen in 2 Anführungszeichen stehen:

```
cout << "Text, Zahlen und Kram";
```

Du kannst auch schreiben:

```
cout << "Aus" << "gabe" << "mit c"<< "out";
```

Du kannst zwischen einem "" und einem << auch eine neue Zeile beginnen:

```
cout << "Te"
     << "xt";
```

Dann musst du das Semikolon hinter den letzten Stringfragment setzen.

Genauer erklärt:

cout ist kein Schlüsselwort, sondern ein Ausgabeobjekt - ein Objekt der objektorientierten Programmierung zur Ausgabe von Daten auf den Bildschirm. Du wirst später was darüber erfahren - wichtig ist, dass du damit komfortabel Daten ausgeben kannst.

Noch zwei Neuerungen gegenüber dem vorigem Programm (welches **main2** war):

Die Anweisung `#include <iostream>` :

`#` - Eine Raute veranlasst den Präprozessor, die nachfolgende Anweisung auszuführen. Der Präprozessor ist ein Programm, das noch vor dem Kompilieren abläuft.

`include` - zu Deutsch "Einbeziehen, inkludieren". Der Präprozessor ersetzt die gesamte Anweisung durch den Inhalt der einbezogenen Datei. Dateien in 2 eckigen Klammern `<>` sucht der Compiler im Standard-Include-Verzeichniss - **`C:\MinGW\include\`** . Dateien in Anführungszeichen `""` sucht der Compiler in dem Verzeichnis, wo das Projekt gespeichert ist.

Kleine Datenkunde

`*.c` - C Quellcodedatei
`*.h` - C und C++ Headerdatei
`*.cpp` - C++ Quellcodedatei
`*.hpp` - C++ Headerdatei
`*` - ohne Endung - C++ Standard-Headerdatei (nach neuem C++ Standard)

Typischerweise verwendet man aber, obwohl C++ eigene **`*.hpp`** Dateien hat, meist C-typische **`*.h`** Header. Headerdateien sollen uns aber erst später interessieren.

`#include <iostream>` heißt also: **`iostream`** in dem Standard-Include-Verzeichniss suchen und dann die Anweisung `#include <iostream>` durch den Code, der in der **`iostream`**-Datei steht, ersetzen. **`iostream`** beinhaltet wichtige Funktionen (bzw. Objekte) zur Daten-Aus- und Eingabe, darunter `cout`.

Die Anweisung **`using namespace`** `std;`:

In den C++ Headerdateien gehören sämtliche Funktionen und Objekte dem sogenannten Namensraum (**`namespace`**) `std` an. Würde die Anweisung nicht dastehen, müsstest du alle Objekte und Funktionen des jeweiligen Headers mit `std::FunktionOderObjekt` ansprechen. Die Anweisung gibt also bekannt, dass dieser spezielle Namensraum allgemeingültig ist.

Du wirst in fast jedem Beispielprojekt diese Anweisung finden, weil viele der Standardheader ihre Funktionen und Objekte in diesen Namensraum gepackt haben und dieses `std::` auf Dauer stört.

Ein Wort zur Formatierung

Um Code übersichtlich zu halten, nutzt du Leerzeichen, Tabulator und Enter. Je besser Quellcodedateien formatiert sind, desto besser kannst du sie später lesen. Bei sehr langen Ausdrücken in einer Zeile solltest du versuchen, diese in mehrere Zeilen aufzuspalten. Hier einige Tipps:

- jede Anweisung in eine eigene Zeile
- nach jeder geschweiften Klammer mit 3 Leerzeichen einrücken
- den Code klar in Abschnitte gliedern (Sinnabschnitte)
- lange Ausdrücke möglichst so kurz halten, dass man nicht quer scrollen

Kapitel 2

Variable

Eine Variable ist ein Platzhalter für einen Wert, den man ändern kann. Erstellt man in C++ eine Variable, so wird der dafür benötigte Speicherplatz im Arbeitsspeicher reserviert. Wenn man dieser Variable dann einen Wert zuweisen möchte, schreibt das Programm den Wert an die reservierte Stelle im Speicher. Das Erstellen bzw. Bekanntgeben einer Variablen heißt "Deklarieren". Wenn man gleich bei der Deklaration der Variablen einen Wert zuweist, nennt man das "Initialisieren" (weist man später einen Wert zu, so heißt es einfach "Zuweisen").

Variablen deklarieren

Der Syntax einer Variablendeklaration sieht jedes Mal so aus:

```
Variablentyp Name ;
```

Beides mal gibt es viele Möglichkeiten, nun erstmal zu den:

Variablentypen

C++ stellt dir verschiedene Grunddatentypen zur Verfügung. Eine kurze Übersicht siehst du hier:

Datentyp	Größe in Byte	Wertebereich
bool	1	0 oder 1 bzw. **true** oder **false**
char	1	-128 bis 127 - jeder Wert steht für das entsprechende Zeichen im ASCII-Code (-128 wird mit dem ASCII-Zeichen 0 verknüpft; 127 entspricht ASCII 255)
short	2	-32768 bis 32767
wchar_t	2	-32768 bis 32767 - ein 16-Bit **char**
int	2 oder 4	-32768 bis 32767 oder -2147483648 bis 2147483647
long	4	-2147483648 bis 2147483647
float	4	3.4E +/- 38 (7 Ziffern nach Komma)
double	8	1.7E +/- 308 (15 Ziffern nach Komma)
long double	10	1.2E +/- 4932

1 Byte ist eine Zusammensetzung aus 8 Bit - ein Bit kann den Wert `0` (**false**) und `1` (**true**) annehmen. Der Computer interpretiert ein Byte (Ziffernfolge aus 8 Nullen oder Einsen) nach dem Zweiersystem.

bool ist ein Typ, der nur 2 Werte speichern kann - `0` (**false**) und `1` (**true**). Die Schlüsselwörter **true** und **false** sind extra für diesen Datentyp geschaffen.

char kann 256 (2^8) verschiedene Werte annehmen, die den im ASCII-Code definierten Zeichen entsprechen. **ASCII-Tabelle**.

wchar_t kann 65536 verschiedene Werte haben, die den im ANSI-Code definierten Zeichen entsprechen. Nur einige Windows-Schriftarten haben für jeden Wert ein Zeichen. Darin sind viele nicht-arabische Alphabete zu finden.

short kann 65536 verschiedene Zahlen annehmen - 32768 Negative und 32767 Positive und natürlich noch Null. **short** ist dasselbe wie **short int**.

int - ebenfalls für Ganzzahlen - kann neben Null auch 2147483648 negative und 2147483647 positive Werte annehmen, wenn die Variable in einer 32-bit Umgebung ist - z.B. Windows ab Version 95. Unter Dos und Windows 3.11 ist ein **int** 2 Byte groß, weil diese 16-bit-Umgebungen sind. Da Dev-C++ mit MinGW nur unter Windows ab Version 95 verfügbar ist, kannst du sicher sein, das sich **int** wie **long** verhält.

long ist auch für Ganzzahlen, und zwar im Bereich von -2147483648 bis 2147483647. Statt **long** kannst du auch **long int** schreiben.

Bevor es mit Beispielen losgehen kann, muss ich noch etwas erklären, und zwar die

Operatoren

Alle der folgenden Operatoren bis auf den Modulo-Operator kannst du auf Variablen der eben vorgestellten Datentypen anwenden. C++ hat sich nicht pingelig, auch wenn du mal **char** mit **float** multiplizierst :)

Für das Dezimalsystem:

Variablen initialisieren und Werte zuweisen

Dazu braucht man den Zuweisungsoperator `=`. Er bewirkt, wie du dir das denken kannst, dass der Ausdruck auf der rechten Seite der Variablen auf der linken Seite zugewiesen wird. Nehmen wir als Beispiel:

```
int VariableEins = 5; //Initialisierung

VariableEins = 128;   //Zuweisung
```

ist gültig, der Wert von `VariableEins` ist nun `128` geworden, nachdem er den Anfangswert `5` hatte. Komplett falsch ist aber folgender Ausdruck:

```
128 = VariableEins;
```

weil `128` kein gültiger Name für eine Variable ist; dazu kannst du später etwas lesen.

Du kannst auch mehreren Variablen ein und denselben Wert auf einmal zuweisen.

```
VariableEins = VariableZwei = VariableDrei = VariableVier;
```

Der Wert von `VariableVier` wird den Variablen `VariableEins` bis `-Drei` zugewiesen.

Arithmetische Operatoren

Es gibt neben den vier Grundrechenarten `+-*/` auch noch den Rest von einer ganzzahligen Division - der Modulo-Operator.

```
+ für Addition
- für Subtraktion
* für Multiplikation
/ für Division

% als Modulo-Operator
```

Beispiele zu den Grundrechenarten:

```
VariableEins = 33+  3; // ergibt 36
VariableEins = 33 - 3; // 30
VariableEins = 33  *3; // 99
VariableEins = 33 / 3; // 11
```

Der rechte Ausdruck wird zuerst vollkommen ausgerechnet, dann dem linken Ausdruck zugewiesen. Du kannst Leerzeichen hinpacken wie du willst.

Den Modulo-Operator kann man nur auf Ganzzahlen - **`short`**, **`int`**, **`long`** - anwenden. Er liefert den Rest einer ganzzahligen Division. `41 / 4` wäre als Fließkommazahl `10.25` und als Ganzzahl `10` Rest `1`. Der Modulo-Operator gibt also `1`.

```
VariableEins = 41 % 4;  // 1
VariableEins = 99 % 10; // 9
```

Erweiterte Zuweisungsoperatoren

Wenn eine Variable verändert wird und das Ergebnis der Variable wieder zugewiesen wird, kannst du die erweiterten Zuweisungsoperatoren verwenden. Es gibt:

```
VariableEins += 10; // VariableEins = VariableEins + 10;
VariableEins -= 10; // VariableEins = VariableEins - 10;
VariableEins *= 10; // VariableEins = VariableEins * 10;
VariableEins /= 10; // VariableEins = VariableEins / 10;
```

Diese machen den Code natürlich ein wenig übersichtlicher.

Inkrement- und Dekrementoperatoren

Die stellen eine weitere Vereinfachung dar. Später wird es oft vorkommen, dass eine Variable um den Wert 1 erhöht oder verringert werden soll. Und genau das tun diese Operatoren:

```
VariableEins++; // Inkrementoperator
VariableEins--; // Dekrementoperator
```

Der Anhang `++` bedeutet dasselbe wie

```
VariableEins = VariableEins + 1;
```

Der Anhang `--` bedeutet dasselbe wie

```
VariableEins = VariableEins – 1;
```

Präfix und Postfix bei diesen Operatoren

Das war jetzt der Postfix. Es gibt noch das Präfix - beide spielen bei Zuweisungen eine Rolle:

```
VariableEins = VariableZwei++;
```
Postfix, VariableEins wird der Wert von VariableZwei zugewiesen, die dann um eins erhöht wird

```
VariableEins = VariableZwei--;
```
Postfix, VariableEins wird der Wert von VariableZwei zugewiesen, die dann um eins verringert wird

```
VariableEins = ++VariableZwei;
```
Präfix, VariableZwei wird um eins erhöht, und dann VariableEins zugewiesen

```
VariableEins = --VariableZwei;
```
Präfix, VariableZwei wird um eins erniedrigt, und dann VariableEins zugewiesen

Besserer Programmierstil ist aber folgender:

```
//Postfix

VariableEins = VariableZwei;
VariableZwei++;

VariableEins = VariableZwei;
VariableZwei--;
```

bzw.

```
//Präfix

VariableZwei++;
VariableEins = VariableZwei;

VariableZwei--;
VariableEins = VariableZwei;
```

Für das Binärsystem:

Variablen kannst du auch auf Binärbasis ändern. Computer können Bitoperatoren schneller verarbeiten als die arithmetischen Kollegen. Hier die vier Operatoren:

```
& als bitweises UND
| als bitweises ODER
^ als bitweises EXKLUSIV-ODER
~ als bitweises NICHT
```

Mit dem bitweisen UND-Operator kann man zwei Werte so verknüpfen, dass deren Ergebnis nur die Bits auf Eins lässt, die auch in den Ausgangswerten auf Eins standen:

Dezimal	sieht binär so aus:
5 & 4 = 4	00000101 & 00000100 = 00000100
101 & 31 = 100	01100101 & 00011111 = 00000101

Der ODER-Operator lässt nur da Einsen stehen, wo mindestens eine Eins in den Ausgangswerten stand.

Dezimal	sieht binär so aus:
5 | 4 = 5	00000101 | 00000100 = 00000101
101 | 31 = 127	01100101 | 00011111 = 01111111

Beim EXKLUSIV-ODER-Operator ist nur dann die Ergebnisstelle eine Eins, wenn sich die beiden Binärzahlen unterscheiden:

Dezimal	sieht binär so aus:
5 ^ 4 = 1	00000101 ^ 00000100 = 00000001
101 ^ 31 = 122	01100101 ^ 00011111 = 01111010

Beim NICHT-Operator wird nur eine Variable verändert, und zwar so, dass Nullen zu Einsen und Einsen zu Nullen werden:

Dezimal	sieht binär so aus:
~5 = 250	~00000101 = 11111010
~101 = 154	~01100101 = 10011010

bzw.

Dezimal	sieht binär so aus:
~4 = 251	~00000100 = 11111011
~31 = 224	~00011111 = 11100000

Des Weiteren gibt es Schiebeoperatoren, mit ihnen kann man alle Einsen im Binärwert der Zahl um einen beliebigen Betrag verschieben:

<< für das Verschieben der Einsen nach links >> für das Verschieben der Einsen nach rechts

Dezimal	sieht binär so aus:
5 << 4 = 80	00000101 << 4 = 01010000
5 >> 2 = 1	00000101 >> 2 = 00000001
101 >> 2 = 25	01100101 >> 2 = 00011001
60 << 3 = 224	00111100 << 3 = 11100000

Es werden dabei jeweils Nullen aufgefüllt.

Anwenden kannst du das, indem du den veränderten Wert einer Variable zuweist:

```
VariableEins = VariableZwei | VariableDrei;
VariableEins = VariableZwei & VariableDrei;
VariableEins = VariableZwei ^ VariableDrei;
VariableEins = ~VariableDrei;
VariableEins = VariableZwei << VariableDrei;
VariableEins = VariableZwei >> VariableDrei;
```

Hier funktionieren auch die erweiterten Zuweisungsoperatoren:

```
VariableEins |= 33;
VariableEins &= 33;
VariableEins ^= 33;
VariableEins <<= 3;
VariableEins >>= 3;
```

Klammern (für beide Zahlensysteme)

Mit Klammern kannst du die Rangfolge in Ausdrücken gewollt verändern:

```
Variable = 25 * 2 + 2 –10; // 42
Variable = 25 * (2+2) -10; // 90
```

Operatoren für das Potenzieren oder Wurzeln gibt es in C++ nicht. Dafür werden wir später Funktionen kennenlernen.

Damit du deine Tastatur in deinen Programmen gebrauchen kannst:

Eingabe mit cin

Während du mit `cout <<` Informationen ausgeben lassen kannst, ist `cin >>` dafür gedacht, Variablen mit Daten von der Tastatur zu belegen. `cin` verlangt allerdings keine feststehende Zeichenkette, sondern eine Variable. Folgender Code würde eine Eingabe verlangen:

```
int Variable;
cin >> Variable;
```

Hier müsstest du eine ganz normale Dezimalzahl eingeben (nicht außerhalb des vorgesehenen Wertebereichs und auch kein Zeichen) und danach Enter drücken. Das wird uns in zukünftigen Programmen sehr nützlich sein, denn ohne Benutzereingabe ist fast jedes Programm armseelig. Zunächst soll eine solche Eingabe verhindern, dass das Programm sofort wieder schließt. Mit dieser sogenannten Dummy-Eingabe kannst du ja die Beispiele aus dem vorigen Kapitel bereichern, wenn du die Ausgaben mit `cout` sehen willst.

Ganzzahlen

Beispielprogramm variable1 zeigt, wie du Ganzzahlen deklarieren und initialisieren kannst.

Wenn du einer Variablen etwa des Typs **`short`** den Wert `600000` zuweist, ihn anschließend ausgeben lässt, erhältst du den Wert `10176`. Das lässt sich erklären: Du hast den Wertebereich überschritten. Ist der Wert 1 größer (bzw. kleiner) als der Wertebereich es zulässt, beginnt der Computer vom negativ (bzw. positiv) größten Wert weiterzuzählen, was bedeutet, dass zur Variable solange Eins dazuaddiert wird, bis 600000 Einsen hinzuaddiert wurden. Falls die Variable größer als `32767` wird, wird der Wert von der Variable zu `-32768`.

Fließkommazahlen

Beispielprogramm variable2 zeigt, was bei Fließkommawerten anders ist.

float und **double**-Werte musst du also mit einem Punkt als Komma angeben (Bsp: `23.695564`). Wenn du feststehende **float**-Werte einsetzen willst, solltest du ein '`f`' hinter den wert schreiben, damit du dem Compiler dies ausdrücklich klarmachst (wie das Beispiel gezeigt hat). Ansonsten verwendet der Compiler diesen Wert als **double**-Konstante.

Außerdem musst du beachten, dass bei der Division von Fließkommawerten durch Ganzzahlen der Punkt mit hingeschrieben werden muss:

```
float Variable = 3.3f;
float Variable2 = Variable / 2.0f;
```

Wenn du diese Angabe vergisst, würde für `Variable2` `1.15f` herauskommen und dann ganzzahlig gerundet werden (und `1` übrigbleiben).

Zeichen

Beispielprogramm variable3 zeigt den Einsatz von **char**-Werten.

Hier siehst du, wie verschiedenen **char**-Variablen konstante Zeichen zugewiesen werden. Einzelne Zeichen müssen dabei in einfachen Anführungszeichen '`x`' stehen. Dies hast du auch schon bei Ausgaben mit `cout << xyz << ' ';` gesehen.

Mit **char**-Variablen kannst du genauso rechnen wie mit Ganzzahlen. Das scheint zunächst etwas komisch, ist aber durchaus praktisch.

Dir wird aufgefallen sein, dass die Zeichen 0 bis 31 im ASCII-Code eine wesentlich andere Bedeutung haben als die Restlichen - sie sind Steuerzeichen. Jedes dieser besonderen Zeichen hat eine bestimmte Verwendung für das Computer-System.

Wenn du einer **char**-Variable einen solchen Wert Wert zuweisen willst, musst du das gewünschte Zeichen dementsprechend besonders eingeben. Die C++-Synonyme sind in der folgenden Tabelle zusammengefasst:

Steuerzeichen	Bedeutung	ASCII Code-Nummer
`\n`	Neue Zeile	10
`\b`	Linkes Zeichen löschen	8
`\f`	Neue Seite	12
`\r`	Kursor geht zum Anfang der Zeile zurück	13
`\a`	Signalton (beep)	7
`\"`	Doppeltes Anführungszeichen	34
`\'`	Einfaches Anführungszeichen	39
`\\`	Backslash	92
`\?`	Fragezeichen	63
`\t`	Tabulator(horizontal)	9
`\0nn`	nn als Oktalwert für ein ASCII-Zeichen	Oktal nn
`\xnn`	nn als Hexalwert für ein ASCII-Zeichen	Hexal nn

Besonders das `\n` Steuerzeichen wird von großer Bedeutung sein, da du damit auszugebende Texte viel besser formatieren kannst.

wchar_t verwendet einen größtenteils anderen Zeichensatz als **char**. Zeichenketten aus **wchar_t** müssen besonders gekennzeichnet werden:

```
L"Zeichenkette"
```

Da die Verwendung von **wchar_t**-Variablen etwas sonderbar ist, werde ich diese Art von Variablen nicht weiter beschreiben. Ich verwende im Folgenden nur noch **char**-Variablen für Zeichen.

Der **bool**-Datentyp ist jetzt noch nicht wichtig, er wird erst im Abschnitt Fallunterscheidung eine Rolle spielen.

Zahlensysteme mit cout und im Code

Du kannst Ganzzahlen auch anders als im Dezimalzahlensystem ausgeben lassen, indem du `cout` passende Informationen gibst:

- `dec` - dezimal
- `okt` - oktal
- `hex` - hexadezimal

Doch wohin genau??

```
int Variable;

cout << dec << Variable;    // Variable wird wie immer Dezimal
ausgegeben
cout << okt << Variable;    // Variable wird durch okt als Oktal-
Wert ausgegeben
cout << hex << Variable;    // Variable wird durch hex als
Hexadezimal-Wert ausgegeben

cout << hex << Variable << dec << Variable; // Variable wird zuerst
als Hex-Wert ausgegeben
// dec ist notwendig, weil sonst Variable ein Zweites mal
hexadezimal ausgegeben wird
```

Du kannst auch Variablen in unterschiedlichen Zahlensystemen in den Code eingeben, etwa oktal durch voranstellen einer `NULL` (`0`) oder hexadezimal durch voranstellen von `NULL X` (`0x`):

```
int Dezimal = 10;
int Oktal = 012;
int Hexadez = 0xa;
```

Bei der Ausgabe einer Hexadezimal- oder Oktalzahl wirst du feststellen, dass `cout 0x` bzw. `0` nicht mit ausgibt. Falls du das jedoch wünschst, musst du `showbase` als Manipulator für `cout` angeben, andernfalls `noshowbase` oder einfach die Voreinstellung nehmen. Das machst du genauso wie bei `dec`, `okt` und `hex`:

```
cout << showbase << hex << 255;
```

Bei Dezimalbrüchen kannst du zwischen den zwei Arten "festkomma" und "wissenschaftlich" unterscheiden. Dabei ist normalerweise die Festkommaschreibweise

gewählt, wenn es jedoch zu viele Ziffern nach dem Komma gibt, stellt `cout` die Zahl in der wissenschaftlichen Exponentialschreibweise dar. Für Exponentiell musst du `scientific` und für Festkomma `fixed` angeben.

Wenn du solche Extrawürste machst, solltest du immer ein Kommentar hinzufügen, damit deine Absicht klar wird. Eine Null kann schließlich schnell verloren gehen!

Kapitel 3

Hier noch einiges zum Thema Variablen:

- Konstanten
- Benennung von Variablen/Konstanten
- Gültigkeitsbereiche
- Variablenzusätze

Konstanten

Konstanten sind Werte, die während der Laufzeit des Programms nicht geändert werden können. Sie sind also voll und ganz vom Quellcode abhängig.

Es gibt zwei Unterarten, zwischen denen man unterscheiden muss: literale und symbolische Konstanten. Kurz gesagt sind literale Konstanten Werte, die direkt an der gewünschten Stelle stehen und deren Wert also sofort ersichtlich ist. Symbolische Konstanten sind Werte, die durch ein "Symbol", also einen Namen, repräsentiert werden. In gewisser Weise sind symbolische Konstanten wie Variablen.

Literale Konstanten

Diese haben wir in der Tat schon eine ganze Weile verwendet. Literale Konstanten sind Zahlen, Buchstaben oder Strings, die genau da stehen, wo sie gebraucht werden:

```
cout << "Hallo" << 2+3 << ' ' << 0xa2;
```

Ganz einfach zu erkennen - `"Hallo"`, `2+3`, `' '` und `0xa2` sind literale Konstanten.

Symbolische Konstanten

Hier versteckt sich ein Wert hinter einem Symbol, also einem Namen. Diese solltest du genauso wie Variablen benennen - also den Sinn kurz beschreiben. (Näheres zur Benennung von Variablen und Konstanten gleich nach den Konstanten!)

Beispielsweise wäre es durchaus sinnvoll, die Kosten für 100 m^2 Parkettboden anstatt mit

```
float Cost = 100 * 13.99f;
```

durch

```
float Cost = Area * PricePerM2;
```

zu errechnen. Denn dann weiß man gleich die Absicht und das macht durchaus ein Kommentar überflüssig.

Bei der Deklaration von solchen Symbolkonstanten gibt es zwei verschiedene Arten:

1.) Als Variable mit dem Zusatz **const**

Mit dem Schlüsselwort **const**, vor den Variablentyp gesetzt, kannst du eine Konstante erzeugen, die du bei der Deklaration sofort initialisieren musst:

```
const float PI = 3.14159f;
```

Konstanten, die du auf die Art definierst, sind sehr sicher, weil der Compiler zu große Werte an der richtigen Stelle als Fehler markiert.

2.) Als definierter Wert mit `#define`

Mit Hilfe der Präprozessor-Anweisung `#define` kannst du Konstanten definieren:

```
#define PI 3.14159;
```

Für mit `#define` definierte Konstanten wird kein Speicher im Arbeitsspeicher reserviert, weil der Präprozessor noch vor dem Compilieren alle Konstanten im Quellcode sucht und durch den Wert (bei `PI` ist der Wert `3.14159f;`) ersetzt. `#define` braucht kein Semikolon, es gehört also zu PI mit dazu! Das ermöglicht es dir, so etwas zu schreiben:

```
cout << PI
```

Mit `#define` kannst du jedoch nicht nur für Konstanten nutzen, sondern auch für komplette Ausdrücke:

```
#define NL cout << '\n';

int main(void)
{
   NL NL NL
   NL NL NL
   NL NL NL

   return 0;
}
```

Der erste Ausdruck hinter `#define` (also `NL`) ist der Name, alle folgenden Wörter sind die definierte Konstante, es ist also egal, ob und wie viele Leerzeichen in der definierten Konstanten stehen. Damit du komplexe Befehle auf mehrere Zeilen aufspalten kannst, musst du ein Backslash '`\`' hinter das letzte Zeichen der jeweiligen Zeile der Anweisung schreiben:

```
#define NL co\
ut\
<<\
'\n'\
;
```

Dieser Ausdruck ist eigentlich der gleiche wie oben, nur dass er zerlegt wurde. Das `\` als Steuerzeichen im Zeichen `'\n'` verhält sich allerdings nicht so wie das Zerlegungs-`\` eine Zeile darüber, weil nach dem Zeichen noch weitere andere Zeichen kommen.

Wenn du also Konstanten, die Zahlen oder Buchstaben beinhalten, verwenden willst, solltest du (entweder literale oder) symbolische **`const`**-Konstanten einsetzen. Obwohl eine **`const`**-Konstante Speicher verbraucht, solltest du die Sicherheit nutzen.

Benennung von Variablen und Konstanten

Bisher hab ich alle Variablen `VariableEins`, `VariableZwei` usw. und alle Konstanten `KONSTANTE1` usw. benannt. In folgenden Kapiteln werde ich das nicht mehr so oft tun (es sei denn, mir fällt kein gescheiter Name ein). Man kann sich natürlich alle Variablennamen selbst ausdenken, man muss dabei darauf achten, dass man die Variablen

- nicht mit einem Unterstrich beginnen lässt '`_`'
- nicht nach Schlüsselwörtern von C++ benennt (etwa **`void`**)
- nicht mit Nummern am Anfang des Namens versieht (etwa `12DiBromEthan`)
- nicht so benennt, wie es in einem Header schon getan wurde (doppelt definieren)
- dem Zweck entsprechend benennt (etwa `AnzahlRinge`)
- möglichst englische Namen nimmt (`NumberRings`)

Bei Variablennamen, die eigentlich aus mehreren Wörtern bestehen, solltest du neue Wörter immer groß schreiben (etwa `DerErsteBuchstabeDesAlphabets`).

Konstanten solltest du immer groß schreiben (etwa `KONSTANTE1` oder `PI`), damit man sie auf den ersten Blick erkennt.

Lokal und Global

Sämtliche bisher verwendeten Variablen und Konstanten waren global - sie wurden am Anfang des Programms deklariert, noch bevor die `main`-Funktion begonnen hat. Man kann sie aber auch in der `main`-Funktion deklarieren.

Beispiel global1.

In Zeile 8 wird eine **`int`**-Variable deklariert. Für die `main`-Funktion ist sie lokal, für den Block jedoch global, weil sie auch nach dem Block gültig ist. Die Variable `a1` ist total global, sie ist überall gültig. `c1` gilt jedoch nur im Block.

Beispiel global2.

Hier wird der Bereichsoperator `::xyz` gezeigt. Das Programm greift bei einer solchen Anweisung nicht auf das lokalste `xyz`, sondern auf ein globaleres `xyz`.

Du solltest möglichst andere Namen für Variablen mit unterschiedlichen Gültigkeitsbereichen wählen, da du sonst schnell durcheinander kommst.

Variablenzusätze

Neben **const** gibt es noch weitere Zusätze, die du vor den Variablentyp schreiben kannst. Jeweils gilt die Reihenfolge:

```
Zusatz Variablentyp Name;
```

unsigned

Dieses Schlüsselwort legt fest, dass bei Variablen der Typen **int**, **float**, **double**, **long**, **short**, **wchar_t** und **char** keine negativen Werte möglich sind. Dadurch verdoppelt sich der Wertebereich in positive Richtung. Eine **short**-Variable geht dadurch bis maximal 65535.

signed

Genau das Gegenteil von **unsigned** - Variablen können negativ sowie positiv sein. Generell gilt: selbst wenn man diesen Zusatz weglässt, ist die Variable vorzeichenbehaftet.

register

Mit diesem Zusatz wird die Variable im Prozessorregister gespeichert, was teilweise eine große Geschwindigkeitsverbesserung mit sich zieht.

volatile

bewirkt, dass die Variable vom Compiler NICHT optimiert wird. Grund ist, dass so Fehler verhindert werden, wenn Quellen außerhalb des Programms zugreifen. Soll zunächst noch keine Rolle spielen.

Variablentyp durch Variablenzusätze ersetzen

register und **volatile** müssen vor einem Typen stehen; **signed** und **unsigned** werden implizit als **int** angenommen, wenn der Typ nicht spezifiziert ist.

Beispiel implicit1.

Zum Überblick zeige ich hier eine komplette Tabelle mit **unsigned** und **signed**:

Datentyp	Größe in Byte	Wertebereich	Konstante
bool	1	0 oder 1 bzw. **true** oder **false**	`0 1 true false`
char	1	-128 bis 127 (0 ist ASCII 0; -1 ist ASCII 255)	`'a' 'b' 'c' '\n'`
signed char	1	wie **char**	`'a' 'b' 'c' '\n'`
unsigned char	1	0 bis 255 (0 entspricht ASCII 0 :)	`'a' 'b' 'c' '\n'`
short	2	-32768 bis 32767	`5 -10 0xff 084`
signed short	2	-32768 bis 32767	`5 -10 0xff 084`
unsigned short	2	0 bis 65335	`5u 0xffU 084u`
wchar_t	2	-32768 bis 32767 - ein 16-Bit char	`'a' 'b' 'c' '\n'`
int	2 oder 4	-32768 bis 32767 oder -2147483648 bis 2147483647	`5 -10 0xff 084`
signed int	2 oder 4	-32768 bis 32767 oder -2147483648 bis 2147483647	`5 -10 0xff 084`
unsigned int	2 oder 4	0 bis 65335 oder 0 bis 4294967295	`5U 0xffu 084U`
long	4	-2147483648 bis 2147483647	`5l -10l 0xffL 084L`
signed long	4	-2147483648 bis 2147483647	`5l -10L 0xffl 084l`
unsigned long	4	0 bis 4294967295	`5uL 0xffUL 084Ul`
float	4	3.4E +/- 38 (7 Ziffern nach Komma)	`3.5f -12.34f 2e-3f`
double	8	1.7E +/- 308 (15 Ziffern nach Komma)	`3.5 -12.34 2e-3`
long double	10	1.2E +/- 4932	`3.5L -12.34L 2e-3L`

Wie du siehst, gibt es auch zu Ganzzahltypen kleine Anhängsel. **long** sollte mit `l` oder `L` und **unsigned int** oder **unsigned short** sollten mit `u` enden. Diese Buchstaben verdeutlichen dem Compiler ausdrücklich, dass die Konstanten von einem bestimmten Typ sind. Du kannst die Buchstaben groß oder klein schreiben, es spielt im Gegensatz zum Rest von C++ keine Rolle. Ich empfehle, weil das kleine '`l`' fast aussieht wie die eins '`1`', die Buchstaben groß zu schreiben.

Type-cast

long, **int** und **short** sind Ganzzahlentypen. **float** und **double** sind Fließkommazahlentypen. **char** ist eigentlich auch eine Ganzzahl. Wenn du einen **float**- oder **double**-wert einem **long**, **int** oder **short**-Wert zuweisen würdest, würde der Compiler eine Warnung ausgeben:

```
int A1 = 10;
float B1 = 20.55f;
char C1 = 100;
short D1 = 128;

A1 = B1; //A1 = 21; der Compiler rundet 20.55 auf und gibt eine Warnung aus
C1 = A1; // das geht!!
A1 = D1; // das geht auch!!
```

Im Folgenden will ich dir 3 Typumwandlungsoperatoren vorstellen, mit denen du dem Compiler klarmachen kannst, dass du die Umwandlung beabsichtigst. Die ersten zwei sind eher für einen alten Stil der Programmierung, ich will sie dir nur zeigen, damit du von ihnen mal gehört hast.

1.) Traditioneller C-Cast

Dieser Castingoperator stammt vom ursprünglichen C. Dabei kommt der Typ, der erwünscht wird für einen Ausdruck, in Klammern:

```
(Typ) Ausdruck
```

Ein kleines Beispiel:

```
int A1 = 10;
float B1 = 20.55f;
A1 = (int)B1; // keine Compiler-Warnung
```

2.) Funktionaler Cast

Dieser ist eigentlich nur eine Abwandlung des traditionellen C-Casts. Hier ist es genau anderherum: der Ausdruck steht in Klammern und der gewünschte Typ steht davor:

```
Typ (Ausdruck)
```

Als Beispiel:

```
int A1 = 10;
float B1 = 20.55f;
A1 = int(B1); // keine Compiler-Warnung
```

Seinen Namen hat dieser Cast davon, dass er einem Funktionsaufruf gleicht. Er ist nur für die bereits vorgestellten Datentypen verfügbar.

3.) Der C++-Cast **static_cast**<>()

Dieser ist einer der 4 C++-Casts. Er soll die beiden obigen alten Casts ablösen, denn er tut eigentlich dasselbe. Allerdings gelten die neuen Casts als sicherer und auch als besser einsetzbar, wenn auch schreibaufwendiger.

static_cast<>() soll uns also dazu dienen, einen normalen Cast durchzuführen. Er hat folgende Syntax:

```
static_cast<Typ>(Ausdruck)
```

Wieder ein kleines Beispiel:

```
int A1 = 10;
float B1 = 20.55f;
A1 = static_cast<int>(B1); // keine Compiler-Warnung
```

Beispiel cast1.

Wenn also eine Typumwandlung geschehen soll, nimmst du natürlich den neuen glänzenden C++ Cast!

Typedef

Du kannst auch neue Typen definieren, diese basieren jedoch auf schon bekannten Typen. Das geht mit dem Schlüsselwort **typedef**:

```
typedef BekannterTyp NeuerTyp;
```

BekannterTyp: Hier muss entweder ein bekannter Typ (**float**, **int**, **char** ...) oder ein bereits von dir mit **typedef** definierter Typ hin.
NeuerTyp: Hier muss der Name für den neuen Typen hin.

Beispiel typedef1.

Kapitel 4

Bis jetzt waren alle Programme simple Folgen aus Befehlen, langweilig und monoton. Jetzt kommt etwas Abwechselung rein, hier lernst du:

- einfache Fallunterscheidung mit **if**
- verzweigte Fallunterscheidung mit **else** und **else if**
- Mini-Fallunterscheidung mit dem ?:-Operator
- Fallunterscheidung mit **switch**

Fallunterscheidung mittels if

Mit dem Schlüsselwort **if** kannst du flexibel Einzelfälle unterscheiden lassen. Dabei sagst du dem Compiler, unter welcher Bedingung der gewünschte Programmteil ausgeführt werden soll. Der Syntax von **if** ist:

```
if( Bedingung )
{
   //Anweisungen
}
```

WENN die Bedingung wahr ist, DANN werden die Anweisungen in den geschweiften Klammern ausgeführt. Um eine solche Bedingung zu schildern, gibt es die logischen und die vergleichenden Operatoren. Im Kapitel 2 hab ich Variablentypen erklärt - die Booleans kommen hier bei der Fallunterscheidung sinnvoll zum Einsatz.

bool ist ein Typ, der nur 2 Werte speichern kann – 0 (**false**) und 1(**true**). Genauer gesagt kann dieser 1-Byte-Typ genauso viele Werte speichern wie ein **char**. Es geht aber eigentlich nur um 0(**false**) und nicht-0(**true**).

Vergleichsoperatoren

Mit ihnen kann man prüfen, ob und auf welche Weise zwei Werte sich unterscheiden, dabei wird ein Wahrheitswert zurückgegeben, also ein **bool**. Folgende Tabelle zeigt diese Operatoren:

Operator	Name	Erklärung	Beispiel mit `int A = 4;`
==	gleich	Wahr, wenn linker Wert gleich dem rechtem ist	`A == 5` wird zu **false**
!=	ungleich	Wahr, wenn linker Wert sich vom rechtem Wert unterscheidet	`A != 5` wird zu **true**
<	kleiner als	Wahr, wenn linker Wert kleiner als der rechte ist	`A < 4` wird zu **false**
<=	kleiner gleich	Wahr, wenn linker Wert kleiner oder gleich dem rechtem ist	`A <= 4` wird zu **true**
>	größer als	Wahr, wenn linker Wert größer als der rechte ist	`A > 4` wird zu **false**
>=	größer gleich	Wahr, wenn linker Wert größer oder gleich dem rechtem ist	`A >= 4` wird zu **true**

In C++ gibt es mehrere Operatoren, die sich in ihren Symbolen ziemlich ähnlich sehen, so ist z.B. der Zuweisungsoperator (=) und der Vergleichsoperator (==) für viele Anfänger eine fatale Fehlerquelle. Also unbedingt ein bisschen üben, bis sich das eingefleischt hat!!

Logische Operatoren

Hier gibt es nur drei Operatoren:

Operator	Name	Allg. Beispiel	Erklärung	Beispiel mit `int A = 4` und `int B = 6`
&&	UND	`Ausdruck1 && Ausdruck2`	Wahr, wenn `Ausdruck1` und `Ausdruck2` zutreffen	`(A == 4) && (B == A + 2)` wird zu **true**
||	ODER	`Ausdruck1 || Ausdruck2`	Wahr, wenn `Ausdruck1` oder `Ausdruck2` zutreffen	`(A == 6) || (B == A - 2)` wird zu **false**
!	NICHT	`!Ausdruck`	Wahr, wenn `Ausdruck` nicht zutrifft	`!(B == A)` wird zu **true**

Ein bisschen praktisch angewandt:

```
bool a = false;
bool b = true;
short c = 0;

a = !c;     // a = nicht 0 , also a = 1
b = c && a; // b = 0 , weil a = 1 aber c = 0
c = 200;
a = (c==200) || (b!=1); // beide Bedingungen treffen zu, und deswegen ist a = 1
```

Beispiel bool1.

Wie du siehst, kann man auch mehr als zwei Werte vergleichen.

Ein Anfängerfehler

Zunächst könnte es dir Probleme bereiten, wenn du aus der Formulierung "wenn x 4 oder 9 ist" die falschen Bedingungen ableitest:

```
if(x == 4 || 9)
{
   //Anweisungen   //werden IMMER ausgeführt!!
}
```

x wird lediglich daraufhin geprüft, ob es den Wert 4 hat. 9 wird überhaupt nicht mit x verglichen, und da 9 nicht 0 ist, nimmt das Programm **true** an. Also immer schön Acht geben!

Doch nun wieder zu Fallunterscheidung mit if:

Beispiel if1.

Hier kam die **if**-Abfrage zum Einsatz. Wenn du ein Programm schreiben würdest, das eine **short**-Variable nach seinem Wert abfragt, müsstest du jedoch alle möglichen Fälle abfragen (also 65536 **if**-Abfragen), wenn du die Vergleiche mit == vornimmst. Wenn du hier jedoch nur 2 exakte Werte brauchst (**if**(a == 0) und **if**(a == 1), jedoch kein größer oder kleiner), kannst du restlich Fälle mit **else** und **else if** abfragen:

```
short a;
if(a==0)
{
   Anweisungen;
}
else if(a==1) // falls die vorige if-Abfrage nicht zutraf, wird
diese Bedingung geprüft
{
   Anweisungen;
}
else // falls keine der vorigen Abfragen zutraf, werden diese
Anweisungen ausgeführt
{
   Anweisungen;
}
```

Wenn a==0 ist, dann werden die Anweisungen in der ersten **if**-Abfrage ausgeführt, die anderen jedoch nicht. Falls a==1 ist wird die erste **if**-Abfrage übersprungen und die Anweisungen in der **else if**–abfrage ausgeführt. Wenn keiner der beiden Fälle zutrifft, werden die Anweisungen in dem **else**-Zweig ausgeführt. Die **else** und **else if**-Erweiterungen beziehen sich immer auf das letzte **if**, das am DIREKT davor stand. **else if**-Erweiterungen werden nur dann geprüft, wenn die letzte **if**- oder **else if**-Abfrage nicht zutraf. Die **else**-Erweiterung stellt den Schluss einer jeden **if**- bzw. **else if**-Abfrage dar, und kann nicht zweimal direkt hintereinander stehen.

Beispiel if2 zeigt einen solchen Zweig.

Wie du siehst, kann man auch mehrere **if**s ineinander verschachteln. Die inneren **if**s sind natürlich vollkommen unabhängig von den äußeren.

Kurze if-Anweisungen

Es ist auch möglich, EINE Anweisung direkt hinter den Vergleich zu schreiben (anstatt einen ganzen Block zu schreiben):

```
if(Variable == 22) cout << "Text";
```

Dann dürfen allerdings keine weiteren Anweisungen folgen, sonst würden diese auf jeden Fall ausgeführt werden, und nicht von der **if**-Abfrage betroffen sein.

Beachte dies, denn es ist noch so eine anfängliche Fehlerquelle, gleich nach der Bedingung das Semikolon zu setzen:

```
if(Variable == 22); cout << "Text"; //passiert eher selten

if(Variable == 11); //passiert deutlich öfter
{
   cout << "Variable";
   Variable++;
}
```

Beidesmal würden die Anweisungen ausgeführt werden, unabhängig davon, was `Variable` für einen Wert hat.

Der ?: -Operator - ein Vereinfachtes if-else

Ein weiterer Operator ist der ternäre Operator `?:`. Ternär heißt, dass er gleich 3 Operanden verlangt. Er prüft erst eine Bedingung und liefert bei wahrer Bedingung den einen Wert zurück, bei falscher Bedingung den anderen. Meist wird das angewandt, um einer Variablen einen Wert zuzuweisen:

```
int a = (b <= 5) ? b : 6;
```

`a` wird auf jeden Fall ein Wert kleiner oder gleich `6` zugewiesen. Nur wenn `b` größer als `5` ist, wird `a` zu `6`.

Zur besseren Lesbarkeit solltest du ausreichend Leerzeichen einsetzen. Wie gesagt findet dieser Operator fast ausschließlich Anwendung bei Zuweisungen. Extra zu diesem Zweck sind Makros programmiert wurden, die später behandelt werden!

Dieser Operator mag vielleicht etwas seltsam aussehen, stellt aber eine ernsthafte Alternative zum **`if`**-**`else`**-Geäst dar, wenn sie auch schlechter lesbar sind. Hast du schwierige Bedingungen, solltest du dann doch besser **`if`**-**`else`** nehmen!

Fallunterscheidung mit switch

Wenn du sehr viele Werte auf Gleichheit überprüfen willst, empfiehlt sich nicht, das mit **`if`** zu tun, denn es gibt noch einen weiteren Weg:

```
switch(Variable)
{
   case 'A' : Anweisungen1;
              break;
   case 'B' : Anweisungen2;
   case 67  : Anweisungen3;
              break;
   default  : Anweisungen4;
              break;
}
```

Hier wird keine Bedingung zur Überprüfung herangezogen, sondern eine Variable. Die zu überprüfende Variable steht dabei in den Klammern hinter dem Schlüsselwort **`switch`**. In den geschweiften Klammern wird die Variable mit möglichen Werten verglichen (auf Gleichheit, nicht kleiner oder größer als), wobei es sich hier um eine **`char`**-Variable handelt. Hinter das Schlüsselwort **`case`** kommt der mögliche Wert, als Buchstabe in einfachen Anführungszeichen oder als Zahl ohne Anführungszeichen.

Danach folgt ein Doppelpunkt, ab hier gehen die Anweisungen für den jeweiligen Fall los. Die Anweisungen werden bis zu dem nächsten Schlüsselwort **`break`** ausgeführt. Das

heißt, dass bei **case** A die Anweisungen1 ausgeführt werden, bei **case** B jedoch werden die Anweisungen2 und Anweisungen3 ausgeführt, weil zwischen den beiden Fällen B und 67 (also 'C') kein **break** steht. Das Schlüsselwort **default** übernimmt alle Fälle, die nicht von den **case**-Abfragen abgefragt werden (praktisch wie das **else** von der **if**-Fallunterscheidung). Auch hier musst du den Doppelpunkt und **break** hinschreiben; außerdem solltest du die **default**-Anweisung IMMER einsetzen.

Beispiel switch1.

Kapitel 5

Wie du sicherlich schon gemerkt hast sind mehrmals hintereinander ausgeführte Befehle oder Befehlsgruppen eine unschöne Tipparbeit. Moderne Programmiersprachen liefern einen recht einfachen Ansatz - die Schleifen. Mehrmaliges Ausführen derselben Befehle mit Schleifen nennt man Iteration. (Die Alternative Möglichkeit ist Rekursion, die du später noch lernen wirst)

Schleifen

Mit Hilfe von Schleifen kannst du einen Teil des Programms beliebig oft wiederholen lassen. Das wird jeweils solange gemacht, bis eine Bedingung nicht mehr zutrifft. Allgemein gesehen unterscheidet man zwischen 3 Schleifen in C++, die ich dir nun nacheinander vorstellen werde.

Die while-Schleife

Der Syntax dieser Schleife:

```
while(Bedingung)
{
   //Anweisungen
}
```

Zuerst wird die Bedingung geprüft; wenn sie wahr ist, dann werden die Anweisungen ausgeführt, andernfalls nicht. Die Bedingung kannst du wie im vorigen Kapitel mit Vergleichsoperatoren und mit logischen Operatoren ausgedrückt.

Stimmt nach einer Durchführung die Bedingung immer noch, so geht es in die zweite Runde. Das ganze wird solange getan, bis die Bedingung nicht mehr stimmt.

Im folgenden Programm würde solange ein Ausrufezeichen ausgegeben, bis 'x' eingegeben wird:

```
int main(void)
{
   char i = 0;

   while(i != 'x')
   {
      cout << '!';
      cin >> i;
   }

   return 0;
}
```

Das folgende Projekt gibt genau 1000 Ausrufezeichen aus:

Beispiel while1.

Eine potentielle Fehlerquelle (wie beim **if**) könnte sein, gleich nach der Bedingung ein Semikolon zu schreiben:

```
while(Bedingung);
{
   //Anweisungen
}
```

Hier könnte leicht eine Endlosschleife entstehen. Bei einer Endlosschleife ist es problematisch, wieder herauszukommen - als einzige Möglichkeit bleibt, das Programm zu beenden.

Die do-while-Schleife

Diese ist der **while**-Schleife sehr ähnlich, mit dem Unterschied, dass die Bedingung erst nach einem Durchlauf der Anweisungen überprüft wird. Ist diese wahr, so werden die Anweisungen solange ausgeführt, bis die Bedingung nicht mehr wahr ist.

```
do
{
   //Anweisungen
}while(Bedingung);
```

Hier muss der Bedingung ein Semikolon folgen. Fehlerquelle könnte sein, dieses zu vergessen oder eines nach dem **do** zu schreiben.

Mit dieser Schleife kannst du sicher sein, dass die Anweisungen mindestens einmal ausgeführt werden. So wird hier einmal '!' ausgegeben, weitere nach Bedarf:

```
int main(void)
{
   char i = 65;

   do
   {
      cout << '!';
      cin >> i;
   }while(i != 'x');

   return 0;
}
```

Beispiel dowhile1.

Die for-Schleife

Der Schleifenkopf (wo die Bedingungen drinstehen) ist ganz anders als bei den anderen beiden:

```
for(int i = 0; i <= 10; i++)
{
   //Anweisungen
}
```

Der Rumpf (wo die Anweisungen stehen) ist wie bei der **while** und **do-while**-Schleife. Der Schleifenkopf besteht allerdings aus mehreren Ausdrücken:

```
for( Initialisierung ; Bedingung ; Anweisung )
```

Der **for**-Schleife wird fast immer eine neue Variable, die Zählervariable, zur Verfügung gestellt. Hier können alle einfachen Datentypen (egal ob Ganz- oder Fließkommazahl, aber auch **bool** und **char**) eingesetzt werden. Die Werte sollten dabei gleich mit initialisiert werden (bzw. zugewiesen, wenn die Variable auch schon außerhalb der Schleife verfügbar gewesen ist). Die Initialisierung kann allerdings auch ganz weggelassen werden, dann muss halt schon eine Variable vorhanden sein, um als Zählervariable eingesetzt werden zu können. Meistens wird ein `int i` als Zählervariable eingesetzt.

Die Bedingung wird, wie zuvor auch schon, mit Vergleichsoperatoren und mit logischen Operatoren ausgedrückt. Bei der Anweisung kannst du einen geeigneten Ausdruck hinschreiben, etwa `i++` , der die Zählervariable verändert. Allerdings kann auch die Anweisung weggelassen werden, dann sollte diese Zähler-Anweisung aber im Rumpf stehen.

Die **for**-Schleife arbeitet nach folgendem Schema:

1.) Variable initialisieren
2.) Bedingung prüfen und entweder
  - in den Rumpf einsteigen oder
  - Schleife verlassen
3.) Anweisungen im Schleifenkopf ausführen
4.) Wiederholung ab 2.)

Bis auf den ersten Durchlauf werden die Anweisungen im Schleifenkopf vor dem Prüfen der Bedingung ausgeführt!

Beispiel **for1** macht genau dasselbe wie die Beispielprogramme mit den **while**- und **do-while**-Schleifen, nur unter Verwendung einer **for**-Schleife.

Sprunganweisungen

Sprunganweisungen dienen dazu, im Programm zu verschiedenen Punkten zu springen. Dabei werden die Werte in den Variablen beibehalten, höchstens werden Variablen gelöscht, deren Gültigkeitsbereich verlassen wird. Eine sehr bekannte, wenn auch verpöhnte Sprunganweisung ist das **goto**.

break in Schleifen

Das Schlüsselwort **break** hat auch bei den Schleifen eine Bedeutung: du kannst durch dieses Wort eine Schleife verlassen, auch mittendrin oder am Anfang:

```
int a;

for(int i = 0; i < 10; i++)
{
   cout << "Gib bitte eine Zahl ein : ";
   cin >> a;
   if(a==0) break; //Schleife wird nur einmal durchlaufen
}
```

Die **for**-Schleife ist durch die Initialisierung einer Variable besonders vorteilhaft, da du die zugehörige Zählervariable meistens sofort im Schleifenkopf findest. Wenn du einen solchen Schleifenkopf verwendest, ist es auch kein großes Problem, aus Versehen ein Semikolon danach zu setzen.

Wenn es jemals nötig ist, eine Endlosschleife zu Programmieren, so musst du sie mit break verlassen. Tatsächlich gibt es nur diese Lösung neben der, das Programm zu beenden. Im realen DOS wäre das allerdings problematisch, denn du müsstest den Computer neu starten. Ein Glück, dass es die Eingabeaufforderung gibt.

continue

Mit **continue** kannst du eine Schleife dazu veranlassen, einen Durchgang, der gerade ausgeführt wird, abzubrechen und gleich mit dem nächsten zu starten. Hier ein Beispiel:

```
int a;

for(int i = 0; i < 10; i++)
{
   a = i;
   if(a == 2) continue; //Wenn a == 2, werden nachfolgende
Anweisungen nicht ausgeführt
                        //und der nächste Durchgang gleich
gestartet
   cout << a << '\n';
}
```

goto

Es gibt noch eine Möglichkeit, Schleifen zu erzeugen:

```
int a = 0;
START:
cout << a << '\n';
a++
if(a < 10) goto START;
```

Mit `START:` wird der Startpunkt definiert (wird auch "Label" genannt). Solange `a` kleiner als `10` ist, veranlasst der Befehl **goto** `START;` das Programm, bei `START` erneut loszulegen. Das funktioniert aber nur innerhalb einer Funktion, nicht Funktionsübergreifend.

Ein **goto** anzuwenden gehört aber nicht zu einem guten Programmierstil. Erstens lässt sich ein damit erreichtes Ziel meist auch ohne das **goto** erreichen, und zweitens ergibt sich bei mehreren **goto**s ein schwer lesbarer Code (schon die Labels verwirren) - manchmal auch Spaghetti-Code genannt. Daher ist ein **goto** nur im äußersten Notfall anzuwenden!!

Beispiel für **break**, **continue** und **goto**: **goto1**.

Kapitel 6

Bereits jetzt hast du Millionen von Möglichkeiten, eigene Programme zu gestalten. Nachdem du genug geübt hast, soll es weitergehen mit:

Funktionen

Bisher liefen alle Befehle in der `main`-Funktion ab, wie etwa Ausgaben mit `cout`, Zuweisungen oder Operationen. Mit Funktionen kann man Teile der `main`-Funktion oder sogar den ganzen Inhalt nach außen verlagern. Es handelt sich also um kleine Unterprogramme im Hauptprogramm, das aus der `main`-Funktion besteht. Funktionen dienen dazu, um Code, der mehrmals in Gebrauch ist, allgemein zur Verfügung zu stellen.

Um Funktionen zu erstellen, kannst du zwei Wege gehen:

1.)
- Funktionsprototyp vor der `main`-Funktion deklarieren
- Funktionsinhalt nach der `main`-Funktion definieren

oder

2.) - ganze Funktion vor der `main`-Funktion definieren

Prototypen deklarieren

Ein Prototyp ist eigentlich nur ein Funktionskopf, der dazu da ist, um die Funktion erst einmal bekannt zu geben (deshalb heißt es Prototyp deklarieren). Jede Funktion kann einen Wert zurückgeben und Parameter verlangen. Dies wird neben dem Funktionsnamen bekannt gegeben.

Die allgemeine Form für einen solchen Prototypen sieht so aus:

```
RückgabeTyp FunktionsName (Parameter);
```

Diese Zeile kommt noch vor die `main`-Funktion:

```
int Funktion(int i); // Funktionsdeklaration oder Prototyp

int main(void)       // Hauptfunktion
{
   //Anweisungen...
}
```

Wie du siehst, können bei dem Rückgabetyp und den Parametern einfache Datentypen eingesetzt werden, es funktionieren aber auch selbsterstellte bzw. komplexe Datentypen (die später behandelt werden). Hier haben wir die Funktion "`Funktion`" mit dem Rückgabetyp **`int`** und dem Parameter `i` vom Typ **`int`**.

Was bedeutet dieses void bei der main-Funktion?

In der Parameterliste kommt ständig dieses **void** vor - es kam nicht bei den einfachen Datentypen vor und dieser Parameter hat auch keinen Namen. Welche Bedeutung hat es??

void repräsentiert das Nichts. Eine Funktion kann einen Wert zurückgeben und Parameter verlangen, muss dies aber nicht. Wenn so etwas vorkommt, setzt du eben das **void** ein. Die main-Funktion verlangt also keinen Parameter.

Man kann es auch weglassen, was dann aber kein guter Programmierstil ist. Bisher haben wir nur mit **int** main(**void**) gewerkelt. Schon bald wirst du aber noch mit einer anderen Form umgehen können.

Funktionen definieren

Nun, die Schnittstelle - der Funktionskopf - ist bekannt. Es fehlt uns allerdings noch an Inhalt. Der Funktionsrumpf kommt bei der Definition der Funktion nach dem Funktionskopf:

```
RückgabeTyp FunktionsName (Parameter)
{
   //Anweisungen
}
```

Die Definition stellst du dann nach der main-Funktion:

```
int Funktion(int i); // Funktionsdeklaration

int main(void)       // Hauptfunktion
{
   //Anweisungen
}

int Funktion(int i) //Funktionskopf
{                   //Funktionsrumpf Anfang
   int z = i * 100;
   return z;
}                   //Funktionsrumpf Ende
```

In der ersten Zeile wird die Funktion deklariert. Im letzten Abschnitt wird die Funktion definiert, d.h. sie wird mit Inhalt belegt.

Variante Zwo

Die zweite Variante, um Funktionen zu erstellen, ist es, die komplette Funktionsdefinition vor die main-Funktion zu schreiben. Damit wird ein Prototyp überflüssig.

Was an der ersten Variante besser ist?? Mit Prototypen gibt man dem Compiler die Funktionen erst einmal bekannt. Er weiß dann, dass es Funktion A gibt und wie diese Funktion aufgebaut ist. Punkt eins ist wichtig, wenn Funktion A eine Funktion B aufruft, die wiederum Funktion A aufruft. Der zweite Punkt ist, dass es einen Fehler hagelt, wenn Definition und Deklaration voneinander abweichen. Dadurch können andere Fehler vermieden werden, die später im Programm auftreten würden. Außerdem ist es übersichtlicher, wenn man gleich die main-Funktion zu Gesicht bekommt anstatt erst seitenweise scrollen zu müssen.

Aufrufen einer Funktion

Jetzt muss sie nur noch aufgerufen werden, damit sie etwas tut. Deklarierte Parameter musst du dabei mit Werten belegen - dabei kann es sich um literale oder symbolische Werte handeln (die eingesetzten Werte werden Argumente genannt). Im Beispiel `Funktion` müsste eine Ganzzahl an `i` übergeben werden:

```
int Funktion(int i);

int main(void)
{
   int u = 5;
   int a = Funktion(u);  //5 wird an i übergeben

   return 0;
}

int Funktion(int i)
{
   int z = i * 100;
   return z;
}
```

In der `main`-Funktion wird eine Variable `u` mit `5` deklariert, danach `a` mit dem Rückgabewert von `Funktion()` mit dem Argument `u`. Bei dieser Deklaration von `a` wird also `Funktion()` aufgerufen. In der Funktion selber hat der Parameter `i` den Wert von `u` erhalten. Nachdem `z` mit `i * 100` initialisiert wurde, gibt Funktion den in `z` gespeicherten Wert zurück.

Beispiel funktion1. Dort findest du die Prototypen der Funktionen `f1()` und `f2()`, die gleich nach der `main`-Funktion definiert werden. In der `main`-Funktion drin werden diese beiden insgesamt 6 Mal mit jeweils unterschiedlichen Argumenten aufgerufen.

Als Überblick und Zusammenfassung:

Ein Prototyp ist die Deklaration einer Funktion. Es handelt sich dabei um den Funktionskopf.

Der Rückgabetyp gibt an, von welchem Typ die zu erwartende Rückgabe ist. Der zurückgegebene Wert wird an die Stelle eingesetzt, an der die Funktion aufgerufen wurde.

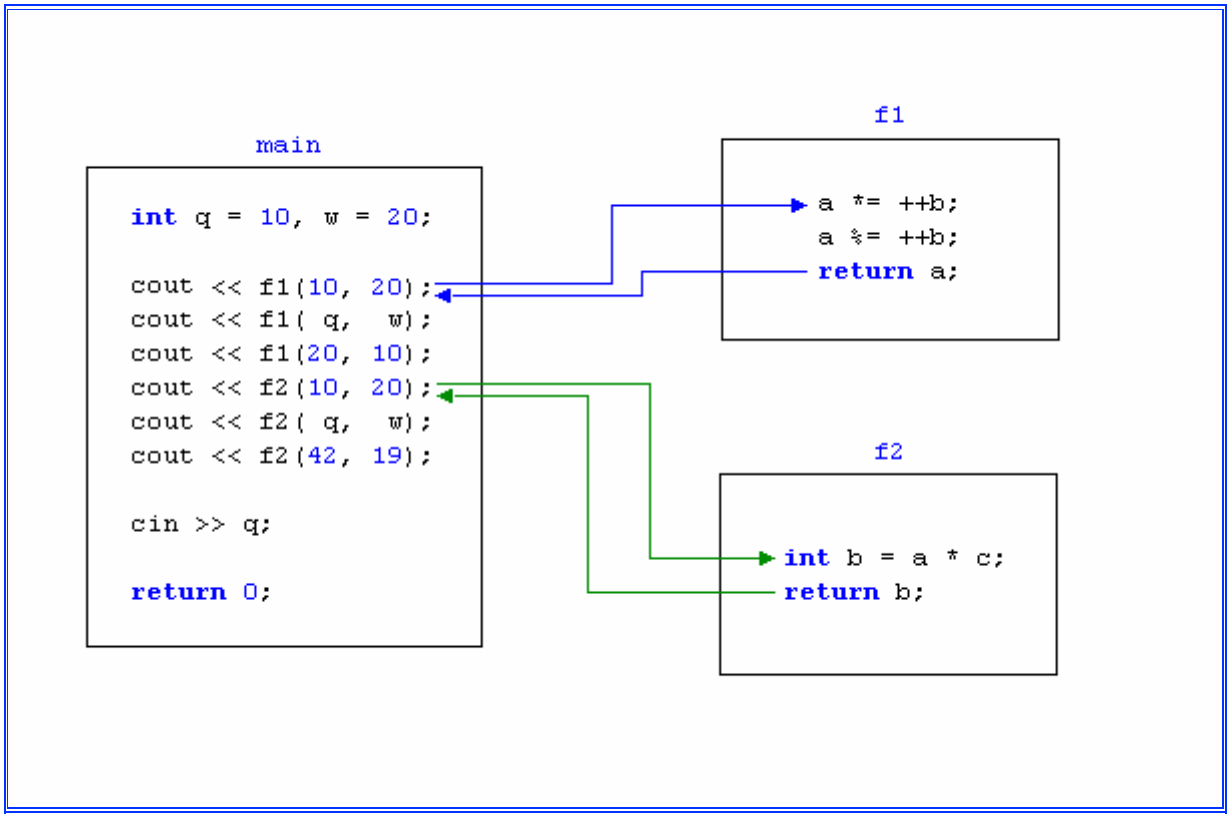
Der Funktionsname identifiziert die Funktion eindeutig. Er ist genau wie bei den Variablen frei wählbar, jedoch musst du dabei dieselben Regeln beachten wie bei den Variablen. Allerdings kann eine Funktion den selben Namen tragen wie eine Variable (der Compiler bemerkt den Unterschied wegen der Klammern).

Ein Parameter ist eine Variable (aus der Parameterliste), die in der Funktion lokal genutzt werden kann, also außerhalb nicht verfügbar ist.

Die übergebenen Variablen bzw. Konstanten (also die in die Parameterliste eingesetzten Werte) heißen Argumente.

Doch wie läuft das genau ab??

Am Projekt **funktion1** (von oben) kannst du sehr gut nachvollziehen, wie das Programm denn eigentlich abläuft:



Diese Darstellung ist realistisch - es werden tatsächlich Sprünge vollführt.

`main` - `f1()` - `main` - `f1()` - `main` - `f1()` - `main` - `f2()` - `main` - `f2()` - `main` - `f2()` - `main` - Ende

Gültigkeitsbereiche

Dabei werden jeweils die Variablen `a`, `b` und `c` mehrmals in den Funktionen benutzt. Diese sind für die Funktionen jeweils lokal und außerhalb nicht verfügbar. Außerdem wird in `f2()` die Variable `b` erstellt - diese ist lokal, also auch nur in `f2()` verfügbar. Auch die Variablen `q` und `w` sind in keiner der anderen Funktionen verfügbar.

Globale Variablen (richtig global, nicht nur in der aufrufenden Umgebung vorhandene) sind in der Funktion natürlich auch verfügbar. Bei gleichnamigen Variablen, die lokal und global vorkommen, musst du mit dem Gültigkeitsoperator `::` unterscheiden.

Parameter sind also lokale Variablen. Beim Funktionsaufruf `f1(q, w)` wird weder `q` noch `w` verändert, sondern immer nur die Parameter (Kopien von diesen Argumenten). Wenn die Argumente aus normalen Variablen bestehen, so nennt man einen Aufruf der Funktion "Call-by-Value". Später werden wir noch "Call-by-Reference" kennenlernen.

Beispiel funktion2.

In diesem Beispiel gibt es 4 Mal die Variable `a`, jeweils mit unterschiedlichem Gültigkeitsbereich:

- `a` global
- `a` in der `main`-Funktion
- `a` in `f1()`
- `a` in `f2()`

Wichtig ist zu wissen, dass das globale `a` überall gilt, aber mit `::a` angesprochen werden muss, wenn es lokalere Variablen gibt.

Funktionen ineinander schachteln

Wie du schon gesehen hast, kannst du Funktionen von anderen Funktionen aus aufrufen. `f1()` und `f2()` wurden jeweils von der `main`-Funktion gestartet. `f1()` und `f2()` könnten sich eigentlich auch gegenseitig aufrufen:

```
int f1(int a);
int f2(int a);
int f3(int a);

int main(void)
{
   cout << f1(1);
   cout << f1(2);

   return 0;
}

int f1(int a)
{
   return a * f2(a);
}

int f2(int a)
{
   return a * a * f3(a);
}

int f3(int a)
{
   return a + 1;
}
```

Einen Aufruf der Funktion `f1()` kannst du dir dann so vorstellen:

```
int main(void)
{
   int a, f1_, f2_, f3_;

   { //f1()
      a = 1;

      { //f2()
         { //f3()
            f3_ = a + 1;
         }

         f2_ = a * a * f3_;
      }

      f1_ = a * f2_;
   }
   cout << f1_;

   { //f1()
      a = 2;

      { //f2()
         { //f3()
            f3_ = a + 1;
         }

         f2_ = a * a * f3_;
      }

      f1_ = a * f2_;
   }
   cout << f1_;

   return 0;
}
```

Hier ist es gleich ersichtlich: mit Funktionen kannst du den Code wesentlich besser strukturieren und kurz halten. Dieser große Block enthält alle wichtigen rechnerischen Anweisungen aus den Funktionen `f1()`, `f2()` und `f3()`. Mit einem einzigen Aufruf der Funktion `f1()` ist der Code im Kasten darüber aber sehr viel verständlicher. Die Blöcke könnten wir natürlich auch kürzen, aber es handelt sich bei Funktionen meist um allgemeine Lösungswege. Es könnte auch einmal ein Aufruf von `f2()` oder `f3()` allein vorkommen.

Funktionsaufrufe als Argumente

Wenn eine Funktion einen passenden Wert zurückgibt, ist es ohne weiteres möglich, eine Funktion als Argument für einen anderen Funktionsaufruf zu verwenden:

```
int f;

int f1(int a, int b);
int f2(int a);
int f3(int a);

int main(void)
{
   cout << f1(f2(5), f3(4));
   cout << f1(f2(6), f3(3));
   cout << f1(f2(7), f3(2));

   cin >> f;

   return 0;
}

int f1(int a, int b)
{
   return a + b;
}

int f2(int a)
{
   return a * a;
}

int f3(int a)
{
   return a + 1;
}
```

Was bei langen Parameterlisten wiederum zu unleserlichen Code führen kann.

Logisch ist, dass zuerst die Funktionen ausgeführt werden, die in einer Parameterliste stehen (bzw. die die innersten Aufrufe sind) und deren Rückgabewerte für andere Funktion als Argument dienen sollen. In der Parameterliste selbst gilt die Regel links vor rechts: erst wird `f2()` ausgeführt, danach `f3()`. Wenn die Funktionen mit denselben globalen Variablen arbeiten, ist dies umso wichtiger!

Das erklärt auch das Phänomen im Beispiel **funktion2** von oben. `cout` ist ein Objekt und es führt eine Funktion aus, wenn man schreibt `cout << "bla";`. Du musst dir die Auflistung von Strings und Variablen und Konstanten hinter dem `cout`-Objekt wie eine Parameterliste vorstellen:

```
cout << " - f1(10) ergibt: " << f1(10) << "\n";
```

übergibt `cout` drei Argumente - `f1(10)` beinhaltet als erste Anweisung `cout << a;`. Deswegen erscheint auf dem Bildschirm zuerst die 10 und dann " - f1(10) ergibt: ...".

Beispiel funktion3 verdeutlicht das nochmals.

Kapitel 7

Dieses Kapitel ist als Erweiterung für den ersten Teil gedacht. Hier erkläre ich dir einige weitere wissenswerte Dinge betreffs Funktionen.

Rekursion

Im vorigen Kapitel hast du die Iteration kennengelernt. Rekursion ist die alternative Möglichkeit zum Wiederholen von Anweisungen.

Weiter oben hast du erfahren, dass Funktionen andere Funktionen aufrufen können. Eine rekursive Funktion geht einen Schritt weiter und ruft sich selber auf (direkte Rekursion) oder sie ruft eine andere Funktion auf, die dann wieder die erste Funktion aufrufen wird (indirekte Rekursion). Dies ist für einige Probleme eine gute Lösungsmöglichkeit - besonders im mathematischen Bereich.

Zur Rekursion gehört aber immer auch ein zweiter Teil. Ohne Abbruchbedingung gleicht eine rekursive Funktion einer Endlosschleife. Um das zu realisieren, muss jede Rekursionsfunktion eine Rückgabe machen und Parameter verlangen.

Es gibt zwei oft gebrachte Beispiele für die Rekursion: die Fakultät einer Zahl und die Fibonacci-Reihe. Beide funktionieren mit Ganzzahlen. Später wirst du eine Funktion zum Sortieren von Daten kennenlernen, und zwar der rekursive Quick Sort.

Fakultät

Die Fakultät einer Zahl ist die Zahl selbst mit all seinen Vorgängern bis zur 1 multipliziert. In der Mathematik drückt man z.B. die Fakultät von 5 mit 5! aus.

```
5! = 5·4·3·2·1 = 120
```

Verallgemeinert gilt:

```
n! = n·(n-1)·...·2·1
```

Wobei n größer oder gleich 2 ist. Die Fakultät von 0 sowie die Fakultät von 1 ist als 1 definiert.

Um nun zu einem gängigen Ausgangspunkt zu kommen:

```
n! = n·(n-1)!
```

So sieht unsere Implementierung aus:

```
int Fakultaet(int n)
{
   if(n >= 0)
   {
      if(n == 0) return 1;
      else return n * Fakultaet(n-1);
   }
   else return 0;
}
```

Und tatsächlich gibt die zuerst aufgerufene Funktion letzten Endes das komplette Produkt zurück. Dass `n` nicht kleiner als `0` sein darf, hat übrigens damit zu tun, dass `n` mit fortlaufenden Rekursionen immer negativer wird und die einzige Abbruchmöglichkeit darin besteht, den Wertebereich von **`int`** zu unterschreiten und so irgendwann zu `0` zurück zu gelangen.

Beispiel funktion4.

Fibonacci-Reihe

Diese Zahlenreihe besteht aus Zahlen, die sich aus den zwei vorhergehenden Zahlen bilden. Nach der zweiten Zahl der Reihe ist jede Zahl die Summe aus den beiden Zahlen, die davor kamen. Der Anfang der Fibonacci-Reihe sieht so aus:

```
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89
```

Am Anfang steht die 1. Die zweite Zahl ergibt sich aus 0 + 1. Dann 1 + 1, 1 + 2 usw.

Das Problem, das jetzt rekursiv zu lösen wäre: Welche Zahl kommt an 14ter Stelle??

Verallgemeinert könnte man das so Formulieren:

```
Fibonacci-Zahl n = (Fibonacci-Zahl n-1) + (Fibonacci-Zahl n-2)
```

Bedingung ist immer, dass n >= 3 ist. Für n == 2 oder 1 ist der Wert 1.

Um das rauszufinden, gibt es folgende Implementierung:

```
int Fibonacci(int n)
{
   if(n < 3) return 1;
   else return Fibonacci(n-1) + Fibonacci(n-2);
}
```

Beispiel funktion5 versucht, die ersten 15 Fibonacci-Zahlen rauszufinden.

Inline-Funktionen

Mithilfe des Schlüsselwortes **`inline`** kannst du Funktionen so modifizieren, dass sie nicht nur einmal in den Speicher geschrieben, sondern immer da, wo sie benötigt werden. Wenn eine normale Funktion aufgerufen wird, springt das Programm zur Stelle, wo der Maschinencode der Funktion steht. Diese Sprünge können ein wenig Zeit in Anspruch nehmen. **`inline`**-Funktionen wollen da etwas verbessern - für kleine Funktionen, die oft ausgeführt werden, ist es sinnvoll, die Sprünge zu vermeiden und einfach den Maschinencode an Ort und Stelle des Aufrufs zu platzieren.

Das lohnt sich aber nur für kleine Funktionen, die den Code nicht allzu groß aufblähen. Große Funktionen, die selten aufgerufen werden, brauchen so etwas natürlich nicht. Hier ein Kollege, für den es sich lohnt, ihn **`inline`** zu machen:

```
inline void F1(void)
{
   cout << "haha";
}

int main(void)
{
   F1();
   F1();
   F1();
   F1();
   F1();
   F1();
   F1();
   F1();
   F1();
   F1();
   F1();
   F1();
   F1();
   F1();
   F1();

   return 0;
}
```

Hier wird mehrere male die Funktion `F1()` ausgeführt. Das fertige Programm hätte auch aus dem folgenden Code entstanden sein können:

```
inline void F1(void)
{
   cout << "haha";
}

int main(void)
{
   cout << "haha";
   cout << "haha";
   cout << "haha";
   cout << "haha";
   cout << "haha";
   cout << "haha";
   cout << "haha";
   //Und so weiter...

   return 0;
}
```

Diese Maßnahme kann - korrekt eingesetzt - die Geschwindigkeit durchaus erhöhen, aber nicht unbedingt dramatisch.

Standardparameter

Manchmal ist es so, dass einige Parameter bei mehreren Aufrufen gleich sind. Um das ein bisschen einfacher zu gestalten, kannst du Parametern Standardwerte verpassen, sodass du beim Aufruf nicht unbedingt alle Argumente reinschreiben musst:

```
void F1(int a, int b = 100)
{
   //Anweisungen;
}

int main(void)
{
   F1(44);
   F1(44, 99);

   return 0;
}
```

b ist, falls man kein anderes Argument angibt, immer 100. Der zweite Aufruf in der main-Funktion hat also 99 anstatt 100 als Argument. Du kannst jedem Parameter einen Standardwert zuweisen, mit der Einschränkung, dass vor einem Parameter ohne Standardwert kein Parameter mit Standardwert in der Parameterliste stehen darf. Es müssen also immer erst die Parameter weiter rechts einen Standardparameter haben, bevor die Parameter weiter links einen bekommen.

Funktionen Überladen

Jetzt wird es mal wieder verwirrend: In C++ kannst du mehrere Funktionen gleich benennen, ihnen dabei jedoch unterschiedliche Parameter geben. Der Compiler kann selbst unterscheiden, welche Funktion ausgeführt werden soll:

```
void A(char A){}
void A(int A){}

int main(void)
{
   int a = 0;
   char b = 'a';
   A(a); // zweite Funktion, weil der Parameter ein int ist
   A(b); // erste Funktion, weil der Parameter ein char ist

   return 0;
}
```

So etwas nennt man Funktionen Überladen. Dabei zu beachten ist, dass zwar unterschiedliche Parameterlisten bei gleichen Rückgabetypen möglich sind, andersrum geht es aber nicht.

Bei der Übergabe kannst du natürlich auch den Type-cast verwenden, wenn es sich um unterschiedliche Datentypen handelt, und so verhindern, dass eine falsche Version der Funktion aufgerufen wird.

In einem späteren Kapitel wirst du mit Templatefunktionen eine etwas modernere C++-Variante des Überladens kennenlernen. Mach dich aber trotzdem mit dem jetzigen Thema vertraut, es könnte mal nützlich sein!

Rückgabe und Parameter der main-Funktion

Schon im ersten Kapitel hab ich dir eine Form der `main`-Funktion vorgestellt:

```
int main(void)
{
   //...

   return 0;
}
```

Der aktuelle Standard schreibt vor, der `main`-Funktion einen Rückgabetypen zu geben. Der mit **`return`** zurückgegebene Wert sollte `NULL` sein, wenn das Programm richtig abgelaufen ist. Wenn ein Fehler im Programm aufgetreten ist, sollte `1` zurückgegeben werden.

Eine zweite Variante der `main`-Funktion hat 2 Parameter:

```
int main(int argc, char *argv[])
{
   //...

   return 0;
}
```

Diese Konstruktion erlaubt den Zugriff auf übergebene Kommandozeilenparameter. Normalerweise wird ein Programm mit einem Befehl "Programm.exe" ausgeführt. Kommandozeilenparameter können daran angehängt werden und somit dem Programm zur Verfügung stehen: "Programm.exe -hallo -welt".

Der erste Parameter, `argc`, enthält die Anzahl der an das Programm übergebenen Parameter. Diese ist immer mindestens `1`, weil der Programmname immer der erste Parameter ist. Im Beispiel von oben wäre "Programm.exe" also 1, "-hallo" 2 und "-welt" 3.

Der zweite Parameter enthält logischerweise die Inhalte der Kommandozeilenparameter. `argv` ist ein Feld von Zeigern. Weil Zeiger und Felder erst später erklärt werden, solltest du zuerst bis dahin weitermachen und dann noch mal hier herblättern.

Da wir jetzt noch keinen richtig geeigneten Verwendungszweck haben, genügt es erst einmal, die Parameter auszugeben. Dev-C++ bietet eine Funktion, um Parameter beim Ausführen mit zu übergeben: Menu Debug -> Parameters. Hier kannst du (im oberen Feld) Parameter eingeben, jeweils mit Leerzeichen dazwischen.

Beispiel funktion6 - zudem wurden 2 Funktionen mit jeweils 2 Parametern implementiert.

Derartige Funktionalität lässt sich gut bei Programmen, die simple Dateiarbeiten machen, gut einsetzen. Tatsächlich lassen sich einige Packprogramme per Konsole bedienen.

Kapitel 8

Komplexe Datentypen

Komplexe Datentypen (schon mehrmals zuvor erwähnt) sind zusammengeschusterte/-gesetzte Datentypen. Einfache Datentypen beinhalten immer nur einen Wert, während komplexe eine bestimmte Anzahl an Werten speichern kann. Hier lernst du Felder und Strings kennen und im Kapitel darauf werde ich dir Strukturen, Aufzählungstypen, Unionen und Namensbereiche erklären.

Statische Felder (oder Arrays)

In dem Beispiel **GDO** aus Kapitel 4 gibt es eine Menge von Variablen - für jeden Arm gibt es Schaden1, Schaden2 ... und Waffe1, Waffe2 usw. Jede dieser Variablen hat einen eigenen Namen. Mit Feldern wollen wir diesen etwas unschönen Programmcode besser in den Griff bekommen.

Ein Feld ist in C++ eine Sammlung von Variablen mit dem gleichen Datentyp. Da dieses eine bestimmte Anzahl an Variablen haben soll, muss das dem Compiler auch klar gemacht werden.

Felder deklarieren

Du musst, um ein Feld zu deklarieren, den Datentyp, den Namen und dann die Anzahl der Elemente in eckige Klammern schreiben.

Verallgemeinert sieht das dann folgendermaßen aus:

```
VariablenTyp FeldName [ AnzahlDerElemente ];
```

Der Variablentyp kann einer der bereits bekannten (**short**, **int**, **char**, **bool** ...) oder ein komplexer Datentyp sein. Da wir aber noch keine anderen komplexen Datentypen kennen, sind wir dazu gezwungen, die einfacheren Wesen zu verwenden.

Für den Namen gelten wie immer die Regeln für Variablennamen. Auf den Namen folgen dann eckige Klammern - darin ist die Anzahl der Feldelemente (so heißen die einzelnen Variablen im Feld) festgelegt; das kann entweder über eine Zahlenkonstante (`1`, `2`, `4`, `39`...), über einen Term (`1*5`, `99+66`, `A+2*B`), oder über eine Variable erfolgen. Dabei ändert sich die Anzahl nicht, wenn die Variable geändert wird; bei einem statischem Feld kann man die Anzahl der Feldelemente nicht während des Programms ändern.

Auf Feldelemente zugreifen

Um auf ein Element im Feld zugreifen zu können, musst du angeben, auf welches der vielen Elemente du zugreifen willst. Das passiert ebenfalls mit eckigen Klammern. Allerdings musst du beachten, dass das erste Element die Zahl `Null` in den Klammern hat. Demzufolge kannst du auf das letzte Element zugreifen, indem du einfach den Wert, mit dem das Feld deklariert wurde, um 1 erniedrigst.

```
#define AnzahlDerElemente 100
#define letztes AnzahlDerElemente - 1

int FeldName[AnzahlDerElemente];
FeldName[12] = 33;
FeldName[0] = 1;
FeldName[letztes] = 9;
```

Beispiel feld1 zeigt die Deklaration und Zugriff auf ein Feld.

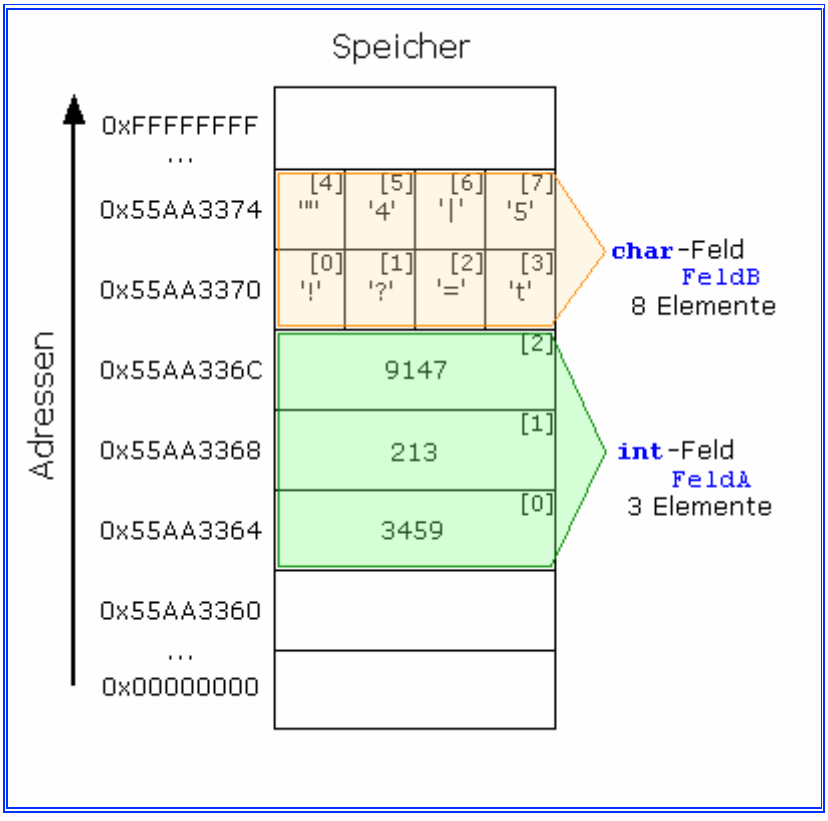
Die Sache mit der Null als Index für das erste Element ist zunächst noch ein bisschen verwirrend. Es gibt sogar einen Begriff dafür, auf das Element nach dem letzten Index zuzugreifen. Man nennt das "Fence Post Error". Der Begriff - zu deutsch "Zaunpfahlfehler" - entstand aus der Frage, wie viele Zaunpfähle man brauchen würde, um einen 10m langen Zaun aufzustellen, bei dem nach jedem Meter ein Pfahl steht. Vorschnell geantwortet sind es 10, mit richtiger Überlegung werden aber 11 draus.

Wie Felder im Speicher aussehen

Wenn du ein Feld deklarierst, reserviert der Computer für x Elemente Speicher. Es entsteht eine Kette von Variablen des genannten Datentyps. Für folgende Deklarationen

```
int FeldA[3];
char FeldB[8];
```

können wir folgenden Speicherinhalt erwarten:



Die darinstehenden Werte sind zufällig noch an den Speicherstellen gewesen. Um ein Feld von Anfang an sinnvoll nutzen zu können, solltest du es immer initialisieren, wie im folgenden Abschnitt gezeigt.

Felder initialisieren

Variablen werden normalerweise mit Werten belegt, die zuvor im Speicher standen. Deswegen ist es sinnvoll, sämtliche Elemente zu initialisieren. Du kannst das tun, indem du

- jedes einzelne Element mit einem Wert belegst, oder
- jedes einzelne Element mittels einer Schleife zu einem Wert verhilfst, oder
- die Werte gleich bei der Deklaration in geschweifte Klammern hintenan setzt.

Die ersten beiden Möglichkeiten kannst du dir selber zusammenreimen. Dei dritte Möglichkeit sieht so aus:

```
int FeldName[10] = { 1, 3, 6, 10, 15, 21, 28, 36, 45, 55 };
```

Der Compiler weist dann jedem Element den Wert an der zugehörigen Stelle zu. Element **0** hätte den Wert 1, Element **1** den Wert 3, Element **2** den Wert 6 und so weiter. Hier musst du für alle Elemente einen Wert in die Klammern setzen.

Du kannst bei einer solchen Initialisierung auch die Angabe in der eckigen Klammer weglassen:

```
int FeldName[] = { 128, 256, 512 };
```

Richtig erkannt, dieses Feld hat drei Elemente. Der Compiler zählt die Werte in den geschweiften Klammern und macht das Feld entsprechend groß genug.

Wie groß ein Feld ist

Je nachdem, was für ein Datentyp verwendet wird und wie groß das Feld ist, wird Speicher im Arbeitsspeicher reserviert. Die Größe in Byte ist dabei das Produkt aus Anzahl der Elemente und Größe des Datentyps. Um die Größe eines Datentyps oder eines Feldes herauszukriegen, gibt es in C++ das Schlüsselwort **sizeof()**:

```
int FeldName[] = { 1, 2, 3 };
int GroesseVomFeld = sizeof(Feldname) / sizeof(int);
```

Die Größe einer **int**-Variable ist (in einer 32 Bit-Umgebung wie Windows) 4 Byte; Die Größe des Feldes ist 3 * 4 Byte, weil es aus drei **int**-Variablen besteht.

Indexberechnung von Feldern

Für jedes Feld wird immer nur die Adresse des ersten Elementes gespeichert. Die restlichen Adressen werden immer live berechnet. Der Computer berechnet diese, indem er die Adresse des ersten Elementes zu der Größe des verwendeten Datentyps mal den Index rechnet:

```
int Feld[5];
//Feld[0] ist bei Adresse 0x1000FF30
//Feld[1] ist bei Adresse 0x1000FF34 weil eine int-Variable 4 Byte
groß ist
```

Die Adresse des ersten Feldelementes wird in einem Zeiger gespeichert. Er hat exakt den Namen des Feldes, nur eben ohne die Indexangabe. Für das obige Beispiel würde der Zeiger einfach "Feld" heißen. Ein Ausdruck

```
int a = *Feld;
```

würde a zu genau dem gleichen Ergebnis verhelfen wie

```
int a = Feld[0];
```

Weil das so ist, kann ein Feld auch ohne Probleme als Argument für einen Zeiger-Parameter verwendet werden.

Mit dem Zeiger ist auch noch einiges weiteres möglich:

Zeigerarithmetik

Nun wissen wir, dass ein Feld auch ein Zeiger sein kann. Mit Zeigerarithmetik können wir diesen Zeiger sogar für das ganze Feld benutzen. Anstatt den Index in die Klammern zu schreiben, können wir den Zeiger in einem Ausdruck um den Index erhöhen.

```
int Feld[5];

Feld[0] = 1;
*Feld *= 2;
cout << "Feld Element 1 : " << Feld[0];

Feld[1] = 2;
*(Feld + 1) *= 2;
cout << "\nFeld Element 2 : " << Feld + 1;

Feld[2] = 3;
*(Feld + 2) *= 2;
cout << "\nFeld Element 3 : " << Feld + 2;

Feld[3] = 4;
*(Feld + 3) *= 2;
cout << "\nFeld Element 4 : " << Feld[3];

Feld[4] = 5;
*(Feld + 4) *= 2;
cout << "\nFeld Element 5 : " << Feld + 4;
```

Dieser Code kompiliert würde allen Feldelementen einen Wert zuweisen, diesen verdoppeln und anschließend ausgeben. Wie du siehst, ist es nicht nötig, die exakten Bytegrößen zu verwenden. Der Compiler nimmt einfach den hinzuzufügenden Wert und verwendet ihn als Index - und der muss ja auch nicht den exakten Bytegrößen entsprechen.

Klar ist, dass dieser Index die Feldgrenzen nicht über- und nicht unterschreiten darf. Stell dir vor, ein gigantisches Feld würde deinen ganzen Arbeitsspeicher beanspruchen; wie würde der Rechner wohl reagieren, wenn du ihn auf ein Element außerhalb deines RAMs zugreifen lassen würdest?

Beispiel feld2.

Felder kopieren

Weil die Felder ihre Adresse als Zeiger gespeichert haben, ist es nicht möglich, ein Felder einem Anderem direkt zuzuweisen (`Feld1 = Feld2;`). Das würde bedeuten, dass zwei Felder die gleiche Speicheradresse haben, und das geht nicht. Du kannst das aber lösen, indem du jedem einzelnem Wert den anderen Wert zuweist - wie im Abschnitt "Felder initialisieren". Am elegantesten geht es natürlich mit einer Schleife:

```
int Feld1[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int Feld2[10];

for(int i = 0; i < 10; i++)
{
   Feld2[i] = Feld1[i];
}
```

Wenn die beiden Felder unterschiedlich groß sind, musst du das natürlich so regeln, dass bei keinem Feld auf Speicher außerhalb zugegriffen wird.

Felder aus Zeigern

Anstatt normale Variablen zu verwenden, kannst du auch Felder aus Zeigern oder Referenzen erstellen.

```
int Feld[5] = { 1, 3, 5, 7, 9 };
int *Feld2[5];

for(int i = 0; i < 5; i++)
{
   Feld2[i] = &(Feld[i]);
}
```

Diese Zeiger enthalten wie gewöhnliche Zeiger Adressen. Oben siehst du, wie man an die Adresse eines Feldelementes außer über Zeigerarithmetik rankommt. Den Wert an der gespeicherten Adresse erhältst du mit dem Dereferenzierungsoperator - du solltest aber Klammern so setzen, dass es leicht als Dereferenzierung erkennbar ist:

```
*(Feld2[2]) = 2;
```

Beispielprojekt dazu ist im folgenden Abschnitt.

Die 4.te Dimension

Bis jetzt waren alle Felder eindimensional. Es geht aber auch, dass Felder mehrere Dimensionen beinhalten. Das lässt sich einrichten, indem du beliebig viele eckige Klammern nach dem Namen setzt:

```
int Feld[8][8];//zweidimensionales Feld
int
Feld2[1888][221321][13123123][123123123][13232][12323][3232312][12312312][1
3123123][12313123];
//Es gehen unvorstellbar viele Dimensionen (begrenzt durch des Computers
Speicher)!!
```

Um derartige Felder zu initialisieren, musst du jede einzelne Dimension initialisieren. `Feld2` hätte damit wohl große Probleme.

`Feld` würde 8·8 also 64 Byte groß sein. Wie es bei `Feld2` steht, weiß ich nicht - Ich bezweifle stark, dass dieses Feld in einen 4 Gigabyte Arbeitsspeicher passen könnte. Es ist eher ein negativ-Beispiel, wie man es nicht machen sollte.

Beispiel feld3.

In der Praxis kommen meistens höchstens zweidimensionale Felder vor. Felder mit mehr als 3 Dimensionen kann man sich nicht so richtig vorstellen; wie stellst du dir ein dreidimensionales Feld vor?? Wie einen Würfel natürlich. Und wie ein vierdimensionales??

Strings

Bis jetzt waren alle Zeichenketten konstant, weil sie direkt im Quellcode standen, also während des Programms nicht änderbar. Mit Feldern ist uns zunächst eine Möglichkeit gegeben, das auszubessern. Variable Zeichenketten kannst du erzeugen, indem du ein Feld aus **char**-Variablen anlegst:

```
char String[] = {88,89,90,0}; // ... = {'X', 'Y', 'Z', '\0'};
String[0] = 65;
String[1] = 66;
String[2] = 67;
```

Die erste Zeile ist schon ein ziemlich umständlich (und durchaus fehleranfällig), das macht sich vor allem bei langen Zeichenketten bemerkbar. Schneller geht es mit den schon verwendeten Zeichenkettenkonstanten:

```
char String[] = "XYZ";
```

Dir ist wahrscheinlich aufgefallen, dass der String mit der einzelnen Zuweisung noch eine Null hinten dran gefügt wurde, die konstante Zeichenkette dies aber nicht hat. Viele Funktionen benötigen das Null-Zeichen, um das Ende des Strings zu identifizieren.

Das obere Exempel bleibt vom Compiler unverändert - die konstante Zeichenkette wird aber beim Compilern angeguckt und das Null-Zeichen wird dazugefügt, wenn der String die Null noch nicht hat. Bei der Ausgabe

```
cout << sizeof(String);
```

würde für beide Strings 4 rauskommen. Um die Größe des wahren Strings zu erfahren, müsstest du folglich `sizeof(String) - 1` rechnen.

Beispiel string1.

In diesem Beispiel wird noch eine andere Sache gezeigt, nämlich die Eingabe eines solchen Strings. Wenn du nur ein Wort eingibst, ist alles normal, wenn du aber ein Leerzeichen und anschließend ein weiteres Wort eingibst, beendet das Programm. Das allerdings liegt in der Eingabefunktion `cin` begründet. Deshalb solltest du hier immer nur ein Wort eingeben. Wir werden später noch eine bessere Möglichkeit kennenlernen, um dann auch mal ganze Sätze eingeben zu können.

Funktionen für Strings

Beispiel string2.

In der Header-Datei **cstring** gibt es eine Reihe von Funktionen, die den Umgang mit Strings ein wenig vereinfachen.

strlen()

`strlen()` (von string lenght) gibt an, wie viele Zeichen einen Wert haben, bis (irgend)ein Null-Zeichen kommt. Verwechsle das nicht mit der Größe des Feldes.

Das Null-Zeichen selbst beachtet `strlen()` dabei nicht. Die Funktion hat die folgenden Prototypen:

```
size_t strlen(const char *String);
```

`size_t` ist ein vorzeichenloser Ganzzahltyp, der Größeninformationen enthalten soll.

strcpy()

Mit `strcpy()` (von string copy) wird der Inhalt eines Strings in einen Anderen kopiert. `strcpy()` hat folgenden Prototypen:

```
char* strcpy(char *Ziel, char *Quelle);
```

Voraussetzung ist, dass das **`Ziel`** mindestens genauso groß ist, wie die Quelle, sonst werden einige Buchstaben weggelassen. Das `Null`-Zeichen setzt `strcpy()` automatisch.

strncpy()

Diese Funktion erweitert `strcpy()` um die Möglichkeit, die Anzahl der zu kopierenden Zeichen besser zu kontrollieren. Ein zusätzlicher Parameter bestimmt die maximale Anzahl an Zeichen, die kopiert werden. Sie hat folgenden Prototypen:

```
char* strncpy(char *Ziel, char *Quelle, size_t MaxCopy);
```

Wieder so ein `size_t` (vorzeichenloser Ganzzahltyp).

strcat()

```
char *strcat(char *String, char *Anhang);
```

`strcat()` hängt den String `Anhang` an `String` an. Das `\0`-Zeichen aus `String` wird mit dem ersten Zeichen aus `Anhang` überschrieben. Zwischen `NULL`-Zeichen und dem Ende von `String` muss soviel Speicher frei sein, dass `Anhang` hineinpasst.

strcmp()

```
int strcmp(char *String1, char *String2); //vergleicht die beiden
Strings;
//wenn beide gleich sind, wird Null zurückgegeben
```

strcmp() vergleicht alle Zeichen der beiden Arrays `String1` und `String2`. Ein Großbuchstabe kommt alphabetisch nach seinem kleinen Äquivalent. Eine Zahl wird zurückgegeben, sobald das erste unterschiedliche Zeichen gefunden wurde oder die Strings vollkommen gleich sind:

- < NULL – String1 ist kleiner als String2
- = NULL – String1 ist genauso groß wie String2
- > NULL – String1 ist größer als String2

Wenn du weitere Funktionen, die Strings arbeiten, kennen lernen willst, schau doch mal bei der Referenz der Funktionen vorbei. Folge dem Link **cstring**. Dort findest du viele Funktionen der C++-Funktionenbibliothek kurz erklärt.

Kapitel 9

Hier geht es gleich weiter mit den komplexen Datentypen. Du lernst in diesem Kapitel Strukturen, Unionen, Aufzählungstypen und Namensräume kennen.

Strukturen

Während Felder Ansammlungen mehrerer Variablen des gleichen Typs sind, handelt es sich bei Strukturen in C++ um einen noch komplexeren Typ - Strukturen können mehrere verschiedene Datentypen beinhalten.

Eine Struktur deklarieren

Um eine Struktur zu erstellen, muss sie mit dem Schlüsselwort **struct** (von structure) deklariert werden, gefolgt vom Namen, der nach der Deklaration als eigens definierter Variablentyp genutzt werden kann:

```
struct Computer
{
   bool An;
   short Mhz;
   short RAM;
};
```

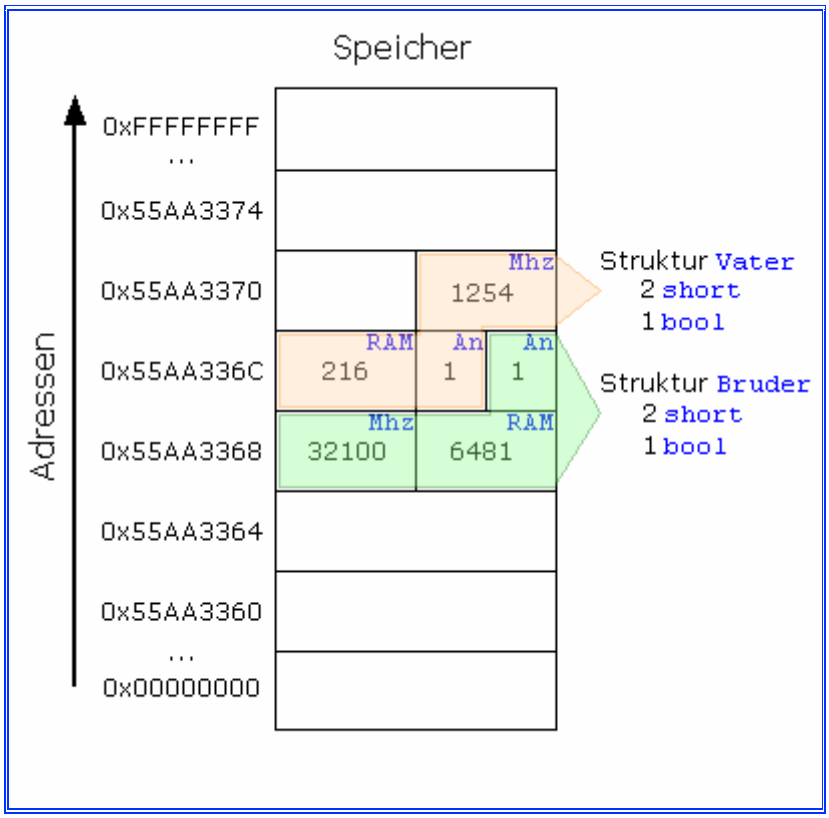
Dieser Code deklariert die Struktur. Das Semikolon hinter der schließenden Klammer ist absolut nötig. Es werden bei der Deklaration keine Variablen dieses Datentyps erschaffen, was aber eigentlich möglich wäre.

Instanzen erstellen

Der Begriff "Instanz" kommt eher aus der objektorientierten Programmierung. Bedeutet ziemlich dasselbe wie "Variable". Eine Instanz zu instanzieren bedeutet, den Computer dazu anzuweisen, Speicher für die Instanz zu beschaffen. Bei Strukturen geht das nicht anders als bei den einfachen Datentypen:

```
Computer Bruder, Vater;
```

Für Bruder und Vater könnten dadurch beispielsweise folgende Speicherabschnitte reserviert werden:



Da den Elementen in den beiden Strukturen noch keine Werte zugewiesen wurden, nehmen sie vorerst die Werte an, die im Speicher stehen. Das ist bei Strukturen nicht anders als bei normalen Variablen.

In diesem Fall hat die Struktur die Größe 5 Byte. Es könnte auch sein, dass der Computer beim Speicher reservieren die einzelnen Elemente nicht direkt aneinander packt, also dass zum Beispiel `An` nicht gleich nach `RAM` folgt, wie das bei der Struktur `Bruder` ist. Die `sizeof(Struktur)`-Angabe könnte dann vom theoretischen Wert abweichen.

Weiter oben habe ich erwähnt, dass Instanzen auch sofort bei der Deklaration erzeugt werden können. Ich hätte also auch schreiben können:

```
struct Computer
{
   bool An;
   short Mhz;
   short RAM;
}Bruder, Vater;
```

Diese Definition von Strukturvariablen ist genau das gleiche wie die Möglichkeit, die aus 2 Anweisungen (so kann man es wohl nennen) besteht.

Wie du auf Elemente zugreifst

Bei einer Struktur kannst du (wie du dir wohl schon gedacht hast) auf jedes deklarierte Element zugreifen. Und damit du das tun kannst, musst du den Element-Auswahl-Operator '`.`' (einen ganz gewöhnlichen Punkt) verwenden:

```
Bruder.An = 0;
Bruder.Mhz = 1234;
Bruder.RAM = 2048;

Vater.An = 1;
Vater.Mhz = 536;
Vater.RAM = 31;

Vater.RAM = Vater.Mhz * Bruder.RAM + Bruder.Mhz;
```

Es sind alle Operationen mit den Elementen wie mit einer normalen Variable, wenn diese vom gleichen Typ wäre. Einzige Schwachstelle ist, dass bei längeren Namen die Übersichtlichkeit den Bach runtergeht.

Beispiel struct1.

Felder von Strukturen

Wenn Felder von einfachen Datentypen möglich sind, dann gibt es in C++ auch Felder von Strukturen. Wie schon bekannt sein dürfte, musst du die Feldstärke in eckigen Klammern angeben:

```
Computer ComputerInDerFirma[16];

//Zugriff:
ComputerInDerFirma[12].Mhz = 33;
ComputerInDerFirma[0].Mhz = 1533;
```

Du solltest immer im Hinterkopf behalten, dass eine Struktur eine Menge Speicher braucht. Hier sind die Strukturen zwar nicht so überwältigend groß, bei großen Projekten solltest du aber darauf achten, dass du nicht zu viel Speicher verschwendest. Irgendwann ist auch mal der größte Speicher erschöpft...

Strukturen in anderen Strukturen

Noch mehr Speicher verschwenden kannst du, indem du eine Struktur in eine andere reinpackst:

```
struct Firma
{
   short Mitarbeiter;
   short ComputerAnzahl;
   Computer ComputerDerMitarbeiter[16];
   short Umsatz;
}Diese, Napoleon, Computerverlag;

Diese.ComputerDerMitarbeiter[3].Mhz = 128;
```

Wenn du mehrere Firmen erstellst, hast du auch gleich eine Armee von Computern. Es ist zu bezweifeln, dass eine Firma namens `Napoleon` gleich 16 Computer braucht. Für den `Computerverlag` allerdings reichen läppische 16 Rechner sicher nicht aus...

Man beachte den komplizierten Zugriff auf die `Mhz`-Zahl des dritten Rechners der Firma `Diese`.

Zeiger auf Strukturen

Ein Zeiger auf eine Struktur ist fast dasselbe wie ein Zeiger auf eine normale Variable. Der Zugriff über den Zeiger auf einzelne Elemente ist dir anfangs vielleicht ein bisschen seltsam:

```
Firma *UltraFirma = &Diese;
(*UltraFirma).Mitarbeiter = 128;
```

Diese Schreibweise mit den Klammern ist wichtig, weil der Punkt eine höhere Priorität hat als der Stern. Würde man hier die Klammern weglassen, würde ein Zugriff auf das Feldelement `Mitarbeiter` erfolgen. Da der Zeiger jedoch auf eine Variable von `Firma` zeigt, und nicht auf dieses Element, gibt es einen Fehler.

Weil ein solcher Zugriff ziemlich oft in C++ verwendet wird, und weil diese Klammerkonstruktion etwas umständlich ist, hat man extra dafür einen Operator eingeführt. Er heißt ebenfalls Element-Auswahl-Operator und sieht so aus: `->`

```
UltraFirma->ComputerAnzahl++;
UltraFirma->Umsatz *= 1.3f;
```

Beispiel struct2.

Zeiger auf Elemente

Hier gibt es eigentlich nichts zu beachten, außer bei Zeigern auf Strukturen:

```
short *MitarbeiterNapoleon = &Napoleon.Mitarbeiter;
short *UmsatzUltraFirma = &(Ultrafirma->Umsatz);
```

Das war's dann erst mal mit Strukturen. Später wirst du Strukturen noch einmal im Zusammenhang mit objektorientierter Programmierung finden.

Unionen

In C++ gibt es noch einen anderen Konstrukt, der aber der Struktur ziemlich ähnlich ist: die Union. Es ist im Prinzip eine Struktur, allerdings wird bei einer Union jede Variable an der selben Stelle gespeichert - wird der Wert überschrieben, gibt es den alten Wert nicht mehr. Die Union ist vom Speicherplatz her gerade so groß wie ihre größte Variable. So sieht der Syntax aus:

```
union Name
{
   char Variable1;
   float X;
   float Y;
}CDU;
Name U1, Alpha;
```

Ähnlich der Struktur kannst du Instanzen einer Union direkt bei der Deklaration oder in einer extra Zeile erstellen.

Wenn etwa auf den Wert `X` zugegriffen wird, wird der Wert genommen, der gerade an der Speicherstelle steht. Zugriff auf einzelne Variable erfolgt wieder mit dem Element-Auswahl-Operator, dem Punkt:

```
Alpha.X = 25.33;
cout << Y; //Der Wert von X wird eigentlich ausgegeben!!
```

Freie Unionen

Es ist auch möglich, den Namen der Union sowie jegliche Variablen dieses Typs wegzulassen. Dann brauchst du keine Elemente mehr auszuwählen:

```
union
{
   char Variable1;
   float X;
   float Y;
};
```

Zugreifen kann man dann wie auf normale Variable:

```
Variable1 = 128;
cout << X; //128 wird ausgegeben (nicht als char sondern als float)
```

Beispiel union1.

Eine freie Union kann mit dem Schlüsselwort **static** initialisiert werden, wenn sie nicht gerade in einer Struktur ist. Dieses **static** bewirkt, dass eine Variable mit Null initialisiert wird. Man kann das aber umgehen, indem man diese Union in einer Struktur deklariert.

Eine Union macht dann Sinn, wenn es mehrere Optionen gibt, von denen aber nur eine ausgewählt sein soll.

Aufzählungstyp enum

Das ist so was wie eine Struktur aus lauter **int**-ähnlichen Variablen, die nach und nach sich um eins erhöhen. Um eine solche Aufzählung zu erstellen, gibt es das Schlüsselwort **enum** (von enumeration). Der Syntax dazu ist:

```
enum ZahlenVonNullBisFuenf { Null, Eins, Zwei, Drei, Vier, Fuenf };
```

Hier steht `Null` für `0`, `Eins` für `1` ... . Danach kann eine Variable dieses Aufzählungstyps deklariert werden. Diese Variable kann dann jeweils nur eine der Werte annehmen:

```
enum ZahlenVonNullBisFuenf AnzahlAepfel = Drei;
ZahlenVonNullBisFuenf AnzahlBirnen = Vier;
```

Das **enum** kann bei der Deklaration noch einmal vor dem Typ stehen (wie das bei C erforderlich war), muss (in C++) aber nicht!!

Man kann jetzt auch darauf zugreifen, sogar auch ohne eine Instanz zu benutzen:

```
if(AnzahlAepfel == AnzahlBirnen) Anweisungen;
if(AnzahlAepfel == Eins) Anweisungen;

int Zahl = 0;
Zahl = static_cast<int>(AnzahlAepfel);
AnzahlBirnen = static_cast<ZahlenVonNullBisFuenf>(Zahl);
```

enum-Elemente können direkt mit anderen Datentypen verglichen werden. Ihnen können aber nur mit einem Type-Cast ein anderer Datentyp zugewiesen werden, was andersrum nicht sein muss.

Beispiel enum1.

Soll die Aufzählung mit einer bestimmten Zahl beginnen, so kann man dem erstem Element auch gleich einen Wert zuweisen:

```
enum alpha {q=5, w, e, r, t, z, u }alpha1;
```

Hier hat q den Wert 5, w 6, e 7 und so weiter.

Namespaces

Ein **namespace**/Namensbereich kennst du schon, und zwar den **namespace** std. In diesem sind sämtliche Funktionen der C++-Standardbibliothek deklariert. Ein kurzes Statement dazu hab ich schon im ersten Kapitel gegeben. Hier wird das jetzt mal etwas genauer erklärt.

Namensbereiche kannst du dir wie Strukturen vorstellen. Sie sind dazu da, um mehrere Variablen oder Funktionen, die den gleichen Namen besitzen, beim Ändern und Lesen bzw. aufrufen richtig auszuwählen. Anders als bei Strukturen musst du allerdings keine Variable erstellen.

Zugriff auf einzelne Elemente erfolgt mittels des Bereichsoperators (::):

```
namespace Eins
{
   int A = 500;
   char B = 'f';
   float C = 13.24f;
}
namespace Zwei
{
   int B = 560;
   char A = 65;
   float C = 24.13f;
}
int B = 200;
short A = 100;

//Zugriff auf Elemente:
Eins::A = 400;
Zwei::C = 33.0f;
B = 644;
```

using namespace

Damit du nicht jedes Mal den Bereichsoperator mit Namen davor schreiben musst, gibt es das Schlüsselwort **using**. Dieses wird so verwendet:

```
using namespace Zwei;
```

Deswegen steht in jedem Beispielprojekt **using namespace** std;!

Falls es schon ein gleichnamiges Element außerhalb des Namespaces gibt, musst du eindeutig unterscheiden:

```
::B++;     //645    //Zugriff auf die globalere Variable
Zwei::B--; //559    //Zugriff ist weiterhin auch mit :: möglich
C *= 1.5;  //36.195 //Zugriff auf Zwei::C
```

Beispiel namespace1.

Mehrere gleiche Namen solltest du natürlich vermeiden. Falls es aber nötig ist, solltest du **namespace**s verwenden. **namespace**s können im Übrigen erweitert werden (im Sinne von Neudefinitionen).