Skript zur vhb-Vorlesung

Programmierung in C++
Teil 1

Prof. Dr. Herbert Fischer
Technische Hochschule Deggendorf

Inhaltsverzeichnis

Primär-Literatur:

Herrmann Dietmar
Grundkurs C++ in Beispielen, vieweg Verlag, 6.Auflage, 2010
ISBN: 3-8266-0910-7

Louis Dirk
C++, Hanser Verlag, 1. Auflage, 2014
ISBN: 3-446-44069-2

Kirch-Prinz Ulla, Kirch Peter
C++ Lernen und professionell anwenden, mitp-Verlag, 5.Auflage, 2010
ISBN: 3-89842-171-6

Arnold Willemer
Einstieg in C++, Galileo Computing, 4.Auflage, 2009
ISBN: 3-8362-1385-0

Sekundär-Literatur:

Helmut Balzert
Lehrbuch der Softwaretechnik, Spektrum Akademischer Verlag, 2.Auflage, 2000
ISBN: 3-8274-0480-0

Peter P. Bothner
Ohne C zu C++, Vieweg, 1.Auflage, 2001
ISBN: 3-528-05780-7

Ernst-Erich Doberkat
Das siebte Buch: Objektorientierung mit C++, B.G. Teubner Stuttgart · Leipzig · Wiesbaden, 2000
ISBN: 3-519-02649-X

John R. Hubbard
C++- Programmierung, mitp, 1.Auflage, 2003
ISBN: 3-8266-0910-7

Dietrich May
Grundkurs Softwareentwicklung mit C++, vieweg, 2.Auflage, 2006
ISBN: 3-8348-0125-9

Tools:

Code::Blocks für Windows, Linux, Mac OS (kostenlose Software): **http://codeblocks.org/downloads/26**

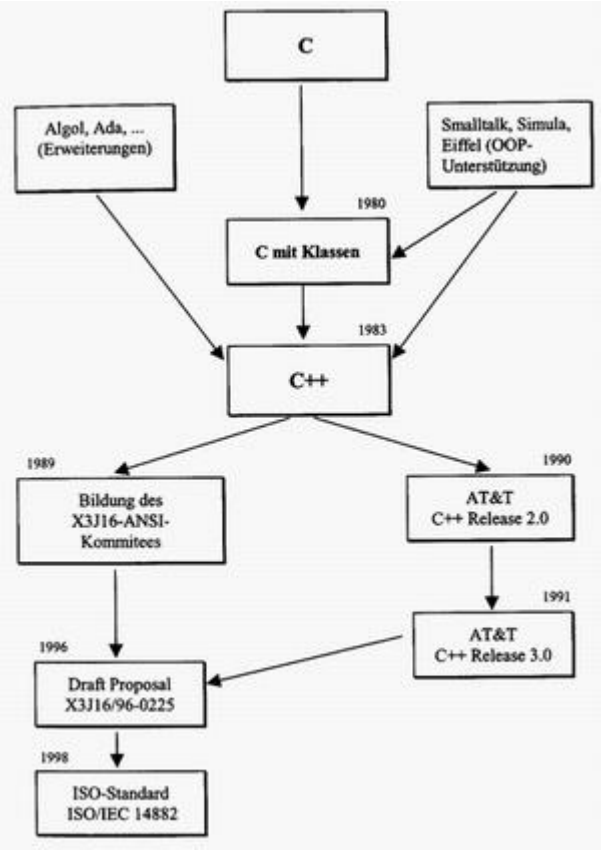
Alternativen:
CodeLite für Windows, Linux, Mac OS: **http://downloads.codelite.org/**
KDevelop für Windows, Linux: **https://www.kdevelop.org/download**
Dev-C++ für Windows: **http://www.bloodshed.net/dev/**
XCode für Mac OS: **https://itunes.apple.com/de/app/xcode/id497799835**

1 Einführung in die objektorientierte Programmierung: C++

In Kapitel 1 erhalten Sie eine kurze Einführung in die C++-Programmierung und werden Ihr erstes C++-Programm erstellen.

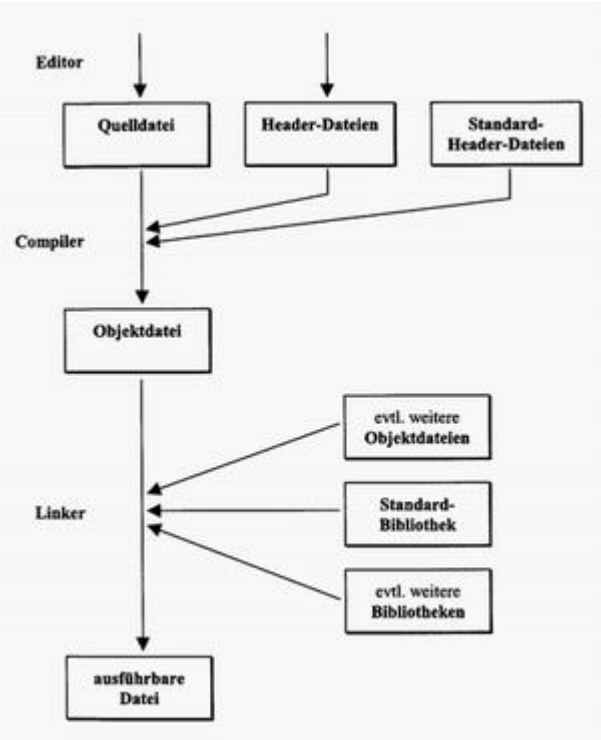
Die Programmiersprache C bildete die Basis für C++, eine Programmiersprache, die häufig auch als »C mit Klassen« bezeichnet wird. In heutigen C++-Programmen ist die Verwandtschaft zu C noch deutlich zu erkennen. C++ wurde nicht geschrieben, um C zu ersetzen, sondern um sie zu verbessern.

1.1 Entwicklung von C++



C++ wurde von Bjarne Stroustrup in den Bell-Laboratorien (Murray Hill, USA) entwickelt, um Simulations-projekte mit minimalem Speicherplatz und Zeitbedarf zu realisieren. Frühe Versionen der Sprache, die zunächst als »C mit Klassen« bezeichnet wurde, gibt es seit 1980. Der Name C++ wurde 1983 von Rick Mascitti geprägt. Er weist darauf hin, dass die Programmiersprache C++ evolutionär aus der Programmiersprache C entstanden ist: ++ ist der Inkrementoperator von C.
C wurde wegen ihrer Effizienz und Portabilität als Grundlage von C++ gewählt. Bei der Weiterentwicklung von C++ wurde stets auf die Kompatibilität zu C geachtet. Somit bleibt die umfangreiche, unter C entwickelte Software auch in C++-Programmen einsatzfähig. Dazu gehören beispielsweise Tools und Bibliotheken für Grafik-Systeme oder Datenbankanwendungen.
Bei der Realisierung objektorientierter Konzepte hatte die Programmiersprache SIMULA67 maßgeblichen Einfluss, insbesondere bei der Bildung von Klassen, der Vererbung und dem Entwurf virtueller Funktionen. Das Überladen von Operatoren und die Möglichkeit, Deklarationen im Programmtext frei platzieren zu können, wurde der Programmiersprache ALGOL68 entlehnt. Die Programmiersprachen Ada und Clu haben die Entwicklung von Templates und die Ausnahmebehandlung beeinflusst.
Schließlich gehen viele Entwicklungen aus den Jahren 1987 bis 1991 auf die direkten Erfahrungen und Probleme von C++-Programmierern zurück. Hierzu gehören beispielsweise die Mehrfachvererbung, das Konzept der rein virtuellen Funktionen und die Nutzung gemeinsamer Speicherbereiche für Objekte.

1.1.1 Der Weg zum ausführbaren C++-Programm



Zur Erstellung und Übersetzung eines C++-Programms sind grundsätzlich die gleichen Schritte wie in C notwendig:

- Das Programm wird mit einem Editor erstellt.
- Das Programm wird kompiliert, d.h. in die Maschinensprache des Rechners übersetzt.
- Der Linker erzeugt schließlich die ausführbare Datei.

1.1.2 Editor

Mit einem Editor werden die Textdateien erstellt, die den C++-Code enthalten. Dabei sind zwei Arten von Dateien zu unterscheiden:

- Quelldateien

Quelldateien, auch Source-Dateien genannt, enthalten die Definitionen von globalen Variablen und Funktionen. Jedes C++-Programm besteht aus mindestens einer Quelldatei.

- Header-Dateien

Header-Dateien, auch Include-Dateien genannt, verwalten zentral die Informationen, die in verschiedenen Quelldateien gebraucht werden.
Dazu gehören:

- Typdefinitionen, z. B. Klassendefinitionen
- Deklarationen von globalen Variablen und Funktionen
- Definition von Makros und Inline-Funktionen

Bei der Benennung der Dateien muss die richtige Endung (engl. Extension) verwendet werden. Diese variieren jedoch von Compiler zu Compiler: Für Quelldateien sind die gebräuchlichsten Endungen .cpp und .cc. Die Namen von Header-Dateien enden entweder wie in C mit .h oder sie haben keine Endung. Aber auch Endungen wie .hpp können vorkommen. Die Header-Dateien der C-Standard-Bibliothek können natürlich weiter benutzt werden.

1.1.3 Compiler

Eine Übersetzungseinheit besteht aus einer Quelldatei und den inkludierten Header-Dateien. Der Compiler erzeugt aus jeder Übersetzungseinheit eine Objektdatei (auch Modul genannt), die den Maschinen-Code enthält. Neben den Compilern, die direkt den Maschinen-Code erzeugen, gibt es auch C++- nach C-Übersetzungsprogramme, sogenannte »C-Front-Compiler«. Diese übersetzen ein C++-Programm in ein C-Programm. Erst anschließend wird die Objektdatei mit einem Standard-C-Compiler erzeugt.

1.1.4 Linker

Der Linker bindet die Objektdateien zu einer ausführbaren Datei. Diese enthält neben den selbsterzeugten Objektdateien auch den Startup-Code und die Module mit den verwendeten Funktionen und Klassen der Standardbibliothek.

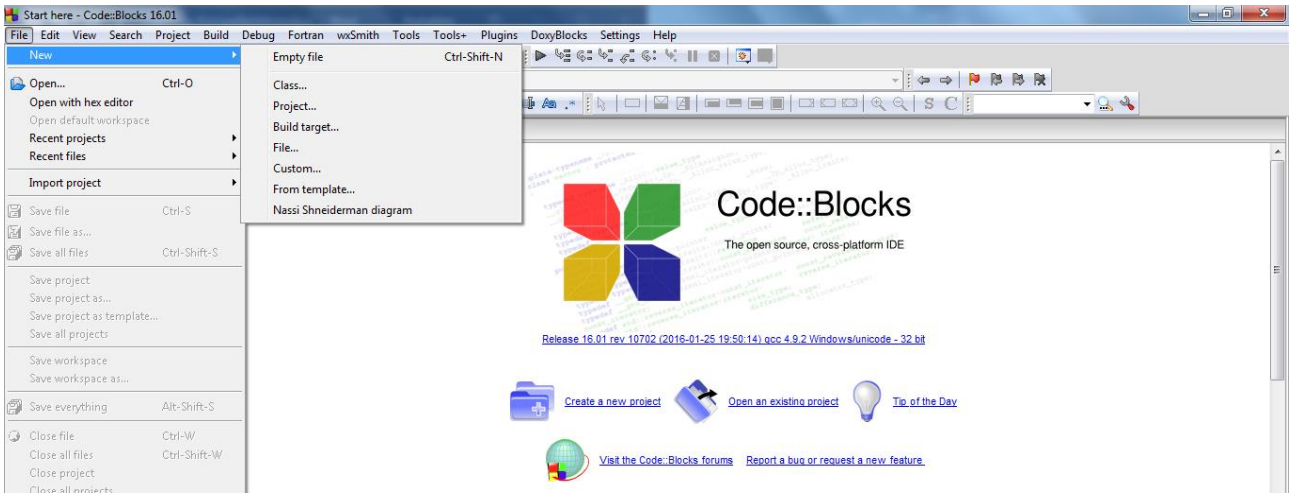
1.2 Einführung in die Programmierumgebung: C++

Nach dieser kurzen Einführung wollen wir nun gleich unser erstes C++-Programm realisieren. Wir werden in diesem Kurs Win32-Konsolenanwendung erstellen. Das sind Programme, die in einem DOS-Fenster laufen. Als Entwicklungsumgebung verwenden wir Code::Blocks. Sie können jedoch analog auch jedes andere C++-Entwicklungstool nutzen (z.B. Dev-C++, CodeLite, XCodes, MS Visual Studio, usw.).
Code::Blocks können Sie unter **http://codeblocks.org/downloads/26** herunterladen.
Ein Videotutorial zu Code::Blocks finden Sie auf der Kurshomepage.

Erstellen einer Konsolenanwendung mit Code::Blocks

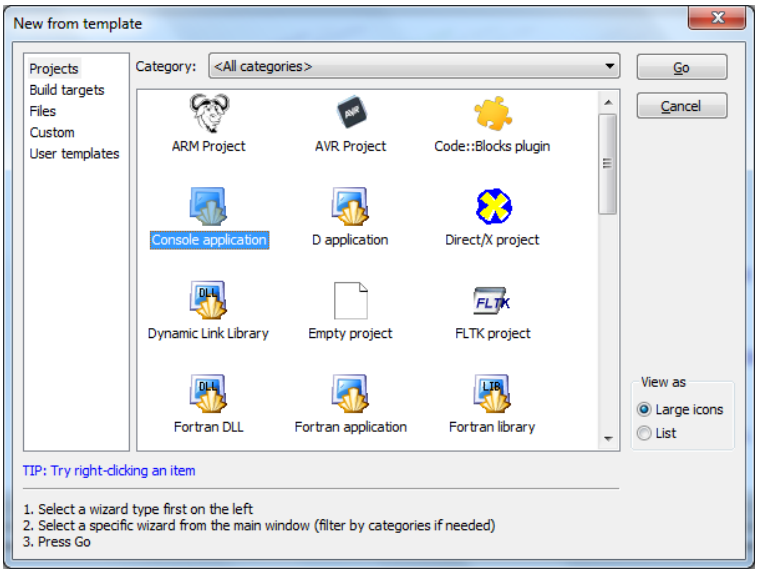
Schritt 1

Starten Sie die Entwicklungsumgebung Code::Blocks. Wählen Sie unter dem Menüpunkt *File* zuerst *New* und dann *Project* oder klicken Sie direkt im Infobereich auf *Create a new project*.



Schritt 2

In dem folgenden Dialogfeld wählen Sie unter der Registerkarte das Icon *Console* application. Klicken Sie dann auf *Go*. Wählen Sie als Sprache C++ aus und vergeben Sie einen von Ihnen frei wählbaren Projekt-Titel und sowie einen Speicherort. Zuletzt prüfen Sie, ob der Compiler *Gnu GCC Compiler*, sowie die beiden Häkchen bei *Debug* und *Release* gesetzt wurden. Klicken Sie zuletzt auf *Finish*.



Schritt 3

Klicken Sie links bei der Navigationsleiste auf *Sources* und machen Sie einen Doppelklick auf *main.cpp*. Dies ist unser Hauptprogramm. Code::Blocks hat schon ein Codegerüst erzeugt, das wir verwenden können:

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char *argv[ ])
{
        cout << "Hello world!" << endl;
        system("pause");
        return 0;
}
```

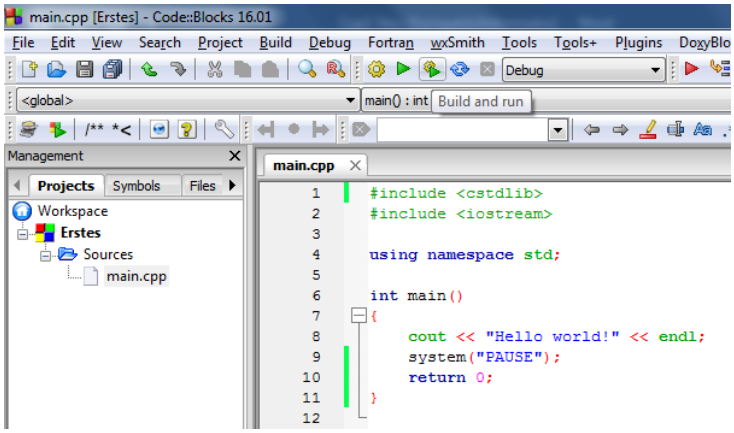
Durch Klicken auf das entsprechende Symbol erzeugt der Compiler ein lauffähiges Programm. D.h. in unserem Fall öffnet sich ein DOS-Fenster, welches die Ausgabe „Hello world!“ erzeugt.

Es gibt folgende Symbole:

Build:	Kompiliert das Programm und erstellt eine ausführbare Anwendung.
Run:	Führt das zuletzt kompilierte Programm aus.
Build & Run:	Kompiliert das Programm und führt es anschließend aus.

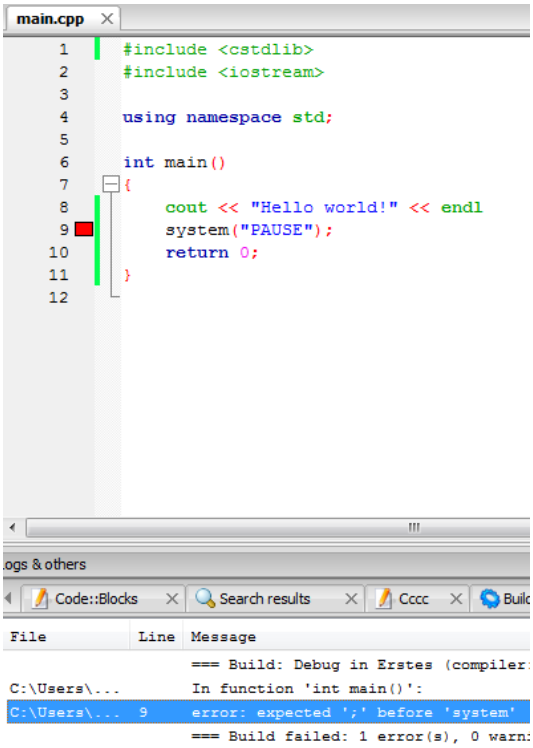
Schritt 4

Klicken Sie auf „Build & Run“, um das Programm zu kompilieren und auszuführen.



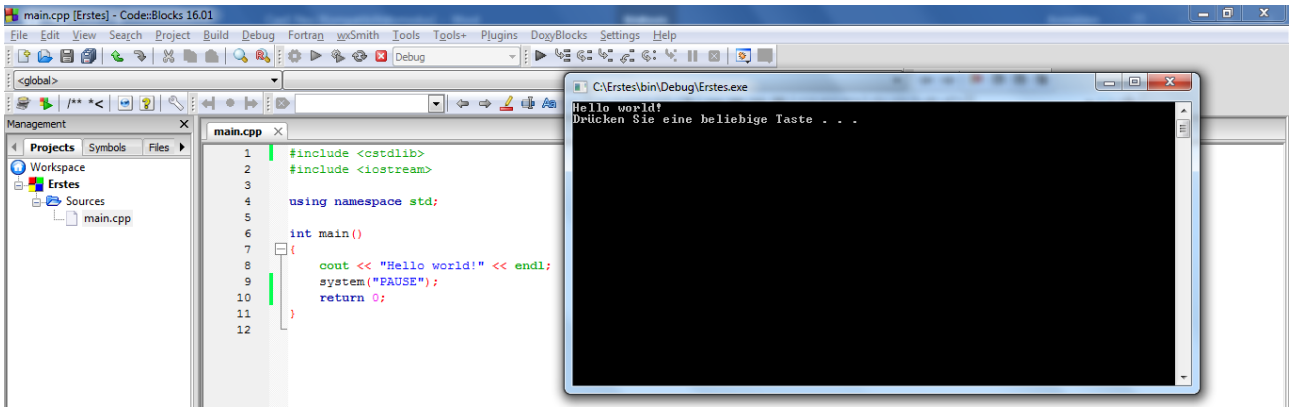
Schritt 5

Wenn der Compiler einen Fehler entdeckt, markiert er die fehlerhafte Zeile links mit einem roten Rechteck und gibt im unteren Fenster eine Meldung aus. Korrigieren Sie den Fehler (hier der fehlende Strichpunkt) und kompilieren Sie nochmal. Speichern Sie die Datei.



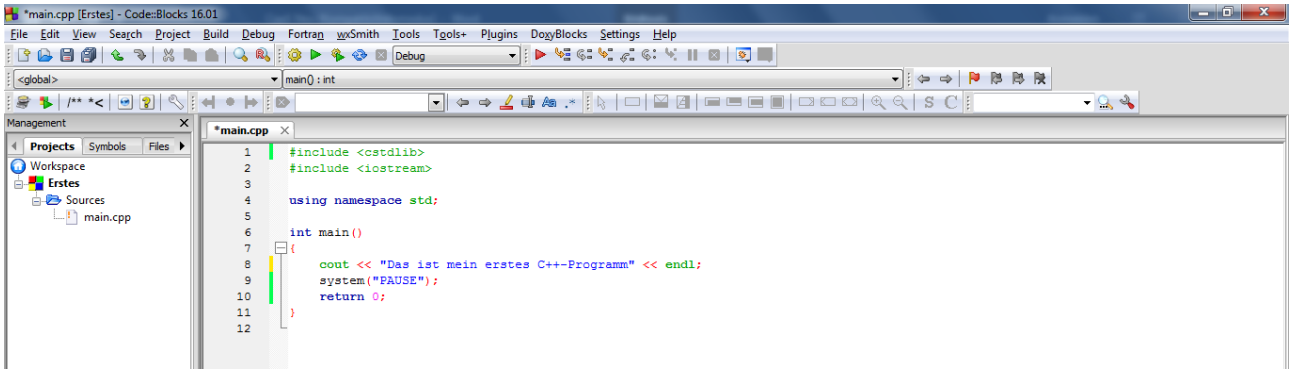
Schritt 6

Nach klicken auf „Run“ wird das Programms in einem DOS-Fenster ausgeführt.



Schritt 7

Speichern Sie (*Save All files* ) und schließen Sie das Programm.



Nun haben Sie Ihr wahrscheinlich erstes C++-Programm mit Erfolg realisiert.

Meinen Glückwunsch!

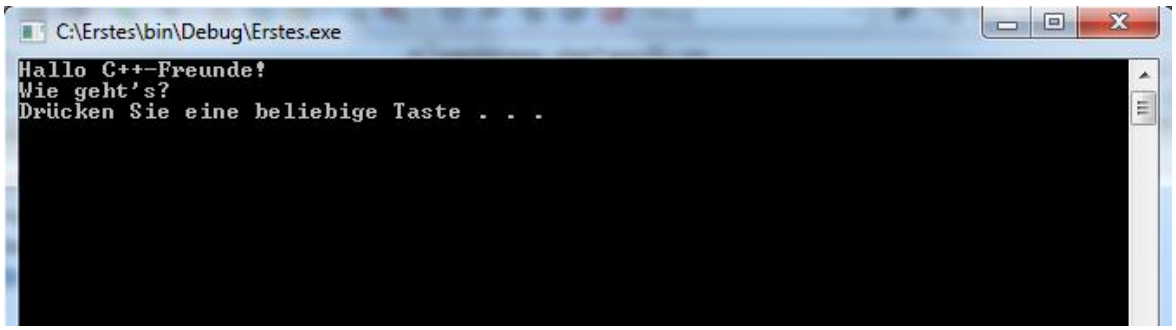
1.2.1 Einfache Ausgabe am Bildschirm

Betrachten wir nochmals unser Programm. Fügen Sie nun folgende Zeilen ein:

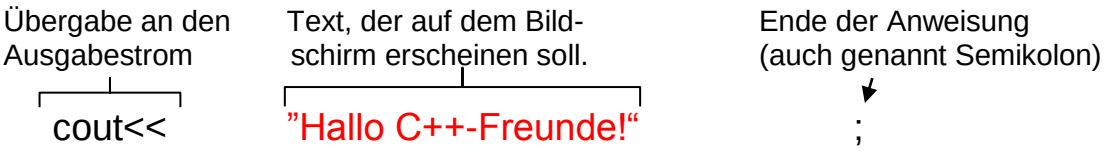
```
cout << "Hallo C++-Freunde!";
cout << endl;
cout << "Wie geht's?" << endl;
```

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char *argv[ ])
{
        cout << "Hallo C++-Freunde!";
        cout << endl;
        cout << "Wie geht's?" << endl;

        system("pause");
        return 0;
}
```

Wenn Sie nun auf „Build & Run“ klicken, erscheint folgende Ausgabe im Konsolenfenster:



Um Text auf dem Bildschirm anzeigen zu können, benötigen wir eine C++-Klasse namens *iostream.* Eine kurze Beschreibung dieser Klasse wäre also angebracht. Sie wissen zwar noch nicht, was Klassen sind, doch sollte Sie das nicht bekümmern. Die Klasse iostream verwendet *streams* (Ströme), um grundlegende Ein-/Ausgabeoperationen durchzuführen – beispielsweise die Ausgabe von Text auf dem Bildschirm oder das Einlesen der Benutzereingabe.



Über den cout-Strom werden Daten an den Standardausgabestrom gesendet. Für Konsolenanwendungen ist dies die Konsole oder der Bildschirm. Die Klasse iostream verwendet spezielle Operatoren, um Informationen in einen Strom zu schreiben. Der *Übergabeoperator* (<<) wird verwendet, um Daten in einen Ausgabestrom zu leiten. Um also Informationen auf der Konsole auszugeben, würden Sie folgendes eingeben: cout << "Tue etwas!";
Damit teilen Sie dem Programm mit, den Text "Tue etwas!" in den Standardausgabestrom einzufügen. Achten Sie dabei darauf, dass der Text in Anführungszeichen steht und die Codezeile mit einem Strichpunkt endet. Wenn die Programmzeile ausgeführt wird, erscheint der Text auf Ihrem Bildschirm.

Anmerkung: cout wird nur in Konsolenanwendungen eingesetzt.

1.2.3 Header-Dateien

Bevor Sie cout einsetzen können, müssen Sie dem Compiler mitteilen, wo die Beschreibung *(Deklaration* genannt) der *iostream*-Klasse steht, in der cout zu finden ist. Die Klasse iostream ist in der Datei IOSTREAM deklariert. Diese Datei wird auch *Header-Datei* genannt.
Um dem Compiler mitzuteilen, dass er in IOSTREAM nach der Klassendeklaration von iostream suchen muss, benutzen Sie die #*include*-Direktive: Durch die #I*nclude*-Direktive werden häufig verwendete Funktionen in den Quellcode eingebunden und so im Programm nutzbar gemacht.

#include <iostream>

Wenn Sie vergessen haben, für eine Klasse oder Funktion die dazugehörige Header-Datei in Ihr Programm mit aufzunehmen, ernten Sie einen Compiler-Fehler. Die Fehlermeldung des Compiler könnte beispielsweise lauten: *Undefined symbol 'cout'*
Wenn Sie diese Meldung erhalten, sollten Sie umgehend überprüfen, ob Sie alle für Ihr Programm benötigten Header mit aufgenommen haben.

using namespace std;

Gibt den Namensraum an, in dem alle Bibliothekselemente in C++ deklariert sind.

Anmerkung:
Die erste Zeile *#include <cstdlib>* wird nur verwendet um System-Anweisungen wie system("pause"); und system("cls“); auszuführen und kann entfallen, wenn man diese Anweisungen nicht nutzt.

1.2.4 Bibliotheken

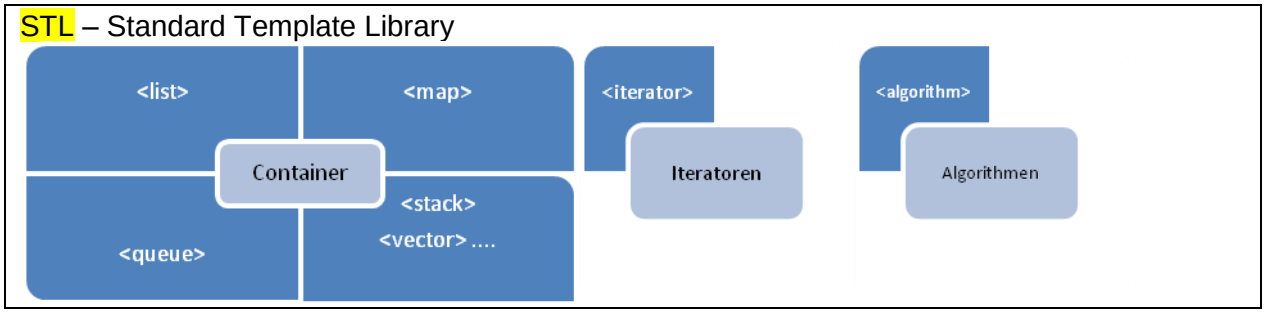
In C/C++ gibt es Bibliotheken mit verschiedenen Klassen und Funktionen. Die Funktionsbibliotheken entstammen teilweise der Programmiersprache C. Der große Vorteil von C und C++ ist die Standardisierung der Bibliotheken. Die C++-Standardbibliothek stellt folgende Komponenten als erweiterbares Rahmenwerk zur Verfügung: Strings, Container, Algorithmen, komplexe Zahlen, Ein-/Ausgabe und vieles mehr. Alle Datentypen, Klassen und Funktionen sind in der Namespace *std* enthalten.

Bereits beim Linken werden die für das Programm benötigten Funktionen aus den Bibliotheksdateien dazu gebunden. Die Verwendung von Bibliotheken erfolgt in zwei Schritten. Zunächst muss die Header-Datei mit Hilfe des Präprozessor-Kommandos *#include* in die Quelltextdatei eingebunden werden. Im zweiten Schritt dagegen werden dem Linker die eigentlichen Bibliotheksdateien bekannt gemacht, z.B. <*string*>.

Beispiele für die wichtigsten Bibliotheken in C++:

<iostream>	Ein-/Ausgabe
<string>	Zeichenketten
<cstdlib>	Hilfsfunktionen
<cmath>	Mathematische Funktionen
<ctime>	Datum und Uhrzeit
<random>	Zufallszahlen

Eine Liste mit weiteren Bibliotheken ist auf folgender Webseite zu finden:
http://www.cplusplus.com/reference/



STL ist die Abkürzung für **S**tandard **T**emplate **L**ibrary. STL ist eine Sammlung von Template-Klassen und stellt einen Container zu Verfügung. Dieser ist in der Lage gleichartige Daten zu organisieren. Die einfachste Form eines Containers ist ein Array (Abschnitt 4.1). Für diese Container stellt die STL einige Funktionalitäten zur Verfügung, wie z.B. Such- und Sortierfunktionen oder Einfüge- und Löschoperationen.

1.2.5 endl;

Die iostream-Klasse enthält spezielle Manipulatoren, mit denen die Behandlung der Ströme gesteuert werden kann. Der einzige Manipulator, mit dem wir uns im Moment befassen wollen, ist *endl;* (end line), der dazu benutzt wird, eine neue Zeile in den Ausgabestrom einzufügen. Wir verwenden *endl,* um eine neue Zeile einzufügen, nachdem wir den Text auf dem Bildschirm ausgegeben haben. Bitte beachten Sie, dass das letzte Zeichen von *endl* ein l und keine 1 ist.
endl kann an den Schluss eine cout-Anweisung angehängt werden, oder mit cout in einer gesonderten Zeile stehen. Man kann auch mehrere *endl* hintereinanderschreiben, um mehrere Leerzeilen auszugeben. Alternativ kann man in den String aber auch \n schreiben, um eine neue Zeile zu erstellen.

```
        cout << ”Hallo C++-Freunde!“ << endl;

oder:   cout << ”Hallo C++-Freunde!“;
        cout << endl;
oder:   cout<< ”Hallo C++-Freunde!\n“;
```

1.2.6 main-Funktion

Die *main*–Funktion (Hauptprogramm) ist der Einstiegspunkt für das Programm. Nach Abarbeitung sämtlicher in ihr enthaltenen Anweisungen, die in den geschweiften Klammern enthalten sind, ist das Programm beendet.

Das Programmgerüst in Code::Blocks lautet:

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

Es genügt jedoch auch:

```
        int main()
```

1.2.7 Klammern und Whitespace-Zeichen

Auffällig sind auch die geschweiften Klammern im Programm. In C++ beginnt ein Codeblock mit einer öffnenden { und endet mit einer schließenden geschweiften Klammer }. Diese Klammern dienen dazu, den Beginn und das Ende der Codeblöcke von Schleifen, Funktionen, if-Anweisungen etc. zu markieren. In unserem Programm gibt es nur einen Satz Klammern, da es sich um ein sehr einfaches Programm handelt.

In C++ werden *Whitespace*-Zeichen einfach ignoriert. Meistens ist es völlig unerheblich, wo Sie Leerzeichen oder neue Zeilen einfügen. Natürlich können Sie Leerzeichen nicht innerhalb von Schlüsselwörtern oder Variablennamen verwenden, aber ansonsten sind Sie nicht gebunden.

So sind die folgenden Quelltexte beispielsweise vollkommen äquivalent:

```
        int main()
        {
                cout << "Hello World!";
        }
entspricht
        int main(){cout<<"Hello World!";}
```

1.2.8 Kommentare

```
#include <iostream>
#include <cstdlib>

using namespace std;       // Das ist ein Kommentar

int main()
{
        cout << ”Hallo C++-Freunde!“;   // Das ist ein weiterer Kommentar
        system(“pause”);
}
```

Nach den Zeichen // können Sie einzeilige Kommentare in Ihren Quelltext eingeben. Kommentarzeilen dienen dazu, Ihr Programm zu dokumentieren.

Mehrzeilige Kommentare kann man auch folgendermaßen schreiben:

```
/*   Kommentar
        .....
*/
```

1.2.9 system("pause");

Die C-Bibliothek stellt uns die Funktion *system("pause");* zur Verfügung, die auf eine Eingabe über die Tastatur wartet. Mit der Tastatureingabe wird das Konsolenfenster geschlossen. Diese Funktion ist compilerspezifisch. Beim Visual C++-Compiler kann *system("pause");* einfach weggelassen werden. Ebenso kann Return EXIT_SUCCESS; entfallen oder durch return 0; ersetzt werden.

Aufgabe: Schreiben Sie ein Programm, das Ihren Namen und Ihre Anschrift wie folgt auf dem Bildschirm ausgibt:

Moritz Mustermann
Am Stadtplatz 1

94469 Deggendorf

Das Fenster soll nach einem Tastendruck geschlossen werden. Kommentieren Sie jede Zeile.

Lösung:

```
#include <iostream>                  // Einbinden der Headerdatei iostream
#include <cstdlib>                   // Einbinden der Headerdatei cstdlib
using namespace std;                 // Namensraum

int main()                           // Hauptprogramm
{                                    // Programmanfang
   cout<<"Moritz Mustermann"<<endl;        // Textausgabe auf dem Bildschirm mit Zeilenumbruch
   cout<<"Am Stadtplatz 1"<<endl;          // Textausgabe auf dem Bildschirm mit Zeilenumbruch
   cout<<endl;                             // Leerzeile
   cout<<"94469 Deggendorf"<<endl;         // Textausgabe auf dem Bildschirm mit Zeilenumbruch

system("pause");                     // Schließen des Fensters erst nach Tastatureingabe
}                                    // Programmende
```

1.2.10 system("cls");

Mit dem Befehl *system("cls");* kann die komplette Bildschirmausgabe gelöscht werden. Der Cursor steht anschließend oben links am Anfang der Befehlszeile.

2 Basis-Syntax in C++

Kapitel 2 vermittelt die grundlegende Basissyntax von C++.
Sie lernen, wie man Variablen und Funktionen verwendet, Rechenoperationen durchführt und Werte von der Tastatur einliest und auf dem Bildschirm ausgibt.

2.1 Ausdruck und Anweisung

Beim Erlernen der Programmiersprachen C bzw. C++ muss zwischen einem *Ausdruck* und einer *Anweisung* unterschieden werden. Die »offizielle« Definition einer Anweisung lautet: »ein Ausdruck, gefolgt von einem Semikolon«. Das Semikolon schließt einen Ausdruck ab und macht daraus quasi einen einzeiligen Quelltextblock. Ein *Ausdruck* ist eine Codeeinheit, die zu einem bestimmten Wert ausgewertet werden kann, während eine *Anweisung* einen abgeschlossenen Ausdruck darstellt. Betrachten wir die folgende Anweisung:

- `c = a + b;`

In diesem Beispiel ist der Teil rechts des Gleichheitszeichens, a + b, ein *Ausdruck*.

Die gesamte Zeile jedoch ist eine *Anweisung*. Im Moment reicht es zu wissen, dass eine *Anweisung* von einem Semikolon gefolgt wird und ein geschlossener *Ausdruck* ist.

2.2 Datentypen

In C/C++ bestimmt ein *Datentyp* die Art, in der der Compiler die Informationen im Speicher ablegt. Die folgende Tabelle enthält die Größen und die sich daraus ergebenden Bereiche für die grundlegenden Datentypen.

Typ	Größe (Bits)	Zahlenbereich	Beispielanwendungen
unsigned char	8	0 <= x <= 255	Kleine Zahlen und kompletter PC-Zeichensatz
char	8	- 128 <= x <= 127	Sehr kleine Zahlen und ASCII-Zeichensatz, z.B. char zeichen = 'A';
short int	16	- 32 768 <= x <= 32 767	Zähler, kleine Zahlen, Schleifensteuerung
unsigned int	32	0 <= x <= 4 294 967 295	Große Zahlen und Schleifen
int	32	-2 147 483 648 <= x <= 2 147 483 647	Zähler, kleine Zahlen, Schleifensteuerung
unsigned long	32	0 <= x <= 4 294 967 295	Astronomische Distanzen
long	32	-2 147 483 648 <= x <= 2 147 483 647	Sehr große Zahlen, statistische Grundgesamtheiten
float	32	1,18 10^-38 <= x <= 3,40 10^38	Wissenschaftlich, 7-stellige Genauigkeit
double	64	2,23 10^-308 <= x <=1.79 10^308	Wissenschaftlich, 15-stellige Genauigkeit
long double	80	3,37 10^-4932 <= x <= 1.18 10^4932	Finanzrechnung, 18-stellige Genauigkeit
bool	8	„true“ (1) oder „false“ (0)	Wahrheitswert abfragen

Einige Datentypen gibt es als *signed* und *unsigned*. Ein *signe*d-Datentyp umfasst sowohl negative als auch positive Zahlen, während ein *unsigne*d-Datentyp nur positive Zahlen enthält.
Sie werden sich vielleicht fragen, warum es in C/C++ zwei gleiche Datentypen mit unterschiedlichen Namen gibt? Dies ist ein Relikt aus alten Tagen. In 16-Bit-Programmierumgebungen belegt der Typ *int* einen Speicherplatz von 2 Byte und der Datentyp *long* 4 Byte. In einer 32-Bit-Programmierumgebung jedoch belegen beide 4 Byte Speicherplatz und verfügen über den gleichen Wertebereich. Bei 32-Bit-Programme sind *int* und *long* identisch.

Nur die Datentypen *double*, *long double* und *float* akzeptieren Fließkommazahlen (Zahlen mit Dezimalstellen). Die anderen Datentypen arbeiten ausschließlich mit Integerwerten (ganzzahlig). Es ist zwar zulässig, einem Integer-Datentyp einen Wert mit Bruchanteil zuzuweisen, doch wird dann der Dezimalanteil nach dem Komma übergangen und nur der ganzzahlige Anteil zugewiesen.

Beispiel:

- `int x = 3.75;`

Der Variablen x wird der Wert 3 zugewiesen, da int nur ganzzahlige Werte annehmen kann. (Das Thema „Variablen“ wird gleich im anschließenden Kapitel behandelt.)

Achtung: In C++-Programmen schreibt man Gleitkommazahlen mit Punkt und nicht mit Komma!

Der Datentyp **bool** kann nur die Werte „true“ oder „false“, bzw. 1 oder 0 aufnehmen. Es wird vor allem benutzt, um Wahrheitswerte in if-Anweisungen (siehe Kapitel 3.1.1) abzufragen.

2.3 Variablen

Eine *Variable* ist ein Name, der mit einer Speicherposition verbunden ist.

2.3.1 Variablendeklaration

Variablennamen können mit einem Unterstrich beginnen. In der Regel ist es jedoch keine gute Idee, einen Variablennamen mit einem Unterstrich zu beginnen, da viele Compiler besondere Variablen- und Funktionsnamen ebenfalls mit einem oder einem doppelten Unterstrich am Anfang kennzeichnen. Die maximale Länge eines Variablennamens ist von Compiler zu Compiler unterschiedlich. Soweit Sie im Rahmen von 31 Zeichen oder weniger bleiben, werden Sie keine Probleme bekommen.

C/C++ unterscheidet bei Variablennamen zwischen Groß- und Kleinschreibung. Bei den folgenden Beispielen handelt es sich um zwei verschiedene Variablen:

```
int Brutto;
int BRUTTO;
```

In C++ müssen Sie den *Variablentyp* (=Datentyp der Variablen) deklarieren, bevor Sie die Variable festlegen können. Der *Variablentyp* sagt aus, was in der Variablen abgelegt werden kann (z.B. Text, Fließkommazahlen, usw.)
Es wird zuerst der Datentyp angegeben, danach der Name der Variablen.

Im Folgenden sehen Sie einige Beispiele für gültige bzw. ungültige Variablennamen:

```
➢ int aVeryLongVariableName;    // ein langer Variablenname
➢ int my_variable;              // eine Variable mit einem Unterstrich
➢ int _x;                       // OK, aber nicht zu empfehlen
➢ int Label2;                   // ein Variablenname mit einer Zahl
➢ int return;                   // ungültig, da return ein Schlüsselwort ist
➢ int zähler;                   // ungültig, Umlaute und Sonderzeichen sind nicht erlaubt
➢ int 123zahl;                  // ungültig, da eine Variable nicht mit einer Zahl beginnen darf
```

Die Festlegung des Variablentyps ermöglicht dem Compiler, eine Typenüberprüfung durchzuführen und sicherzustellen, dass alles korrekt ist, wenn das Programm ausgeführt wird. Die Verwendung eines falschen Datentyps löst einen Compiler-Fehler oder eine Fehlermeldung aus. Durch die Analyse und Korrektur dieser Fehler können Probleme behoben werden, bevor sie sich nachteilig auswirken.

Merke:
Eine *Variable* ist ein bestimmter Bereich im Computerspeicher, der für die Aufnahme eines Werts freigehalten wird.

2.3.2 Variableninitialisierung

Mit der *Initialisierung* bekommt die Variable einen Wert zugewiesen.
Eine Variable kann gleich bei ihrer Deklaration initialisiert werden. So ist beides zulässig:

```
    int Kontostand_Bank1;
    Kontostand_Bank1=100;
oder:
    int Kontostand_Bank2=150;
```

Variablen, die *deklariert,* aber nicht *initialisiert* werden, enthalten Zufallswerte. Da der Speicher, auf den die Variable zeigt, nicht initialisiert wurde, kann auch nicht genau gesagt werden, was dieser Speicherplatz enthält.

Beispiel:

```
int x;
int y;
x = y + 10;      // ACHTUNG! y hat Zufallswert
```

In obigem Beispiel kann die Variable x jeden beliebigen Wert enthalten, da y vor seiner Verwendung nicht initialisiert wurde.

Sie können auch mit einer Zuweisung mehrere Variablen initialisieren:

Beispiel:

```
int x = 2;
int y = 2;
```

oder:

```
int x=2, y=2;
```

2.3.3 Typumwandlung

Werden zwei Variablen unterschiedlichen Typs durch einen Operator miteinander verknüpft, dann bekommt das Ergebnis den genaueren Typ der beiden Variablen. Es werden zwar zwei Datentypen gemischt, aber C/C++ ist imstande, eine automatische *Typenumwandlung* vorzunehmen.

Beispiel:

```
float x=1.0;
int i=2;
x=x/i;
```

Das Ergebnis der Berechnung ist deshalb 0,5.

2.3.4 Arithmetischer Überlauf

Das Ergebnis einer Multiplikation zweier Zahlen vom Typ long soll einer Variablen vom Typ short zugewiesen werden.

Beispiel:

```
short ergebnis;
long a1 = 200;
long a2 = 200;
ergebnis = a1 * a2;
```

Das Ergebnis der Multiplikation lautet -25.536. Dies liegt daran, dass die Zahlen *umbrochen* werden: 200 * 200 = 40 000, der Datentyp *short* hat jedoch einen gereingeren Wertebereich als long und kann somit nur die Zahlen von - 32 768 bis + 32 767 aufnehmen.
Bei einem Ergebnis über 32 767 hinaus, wird das Ergebnis umbrochen, d.h. der übersteigende Rest zu - 32 768 addiert.

```
40 000 - 32 767 = 7 233
7 233 - 1 = -7 232              // von 32 767 zu - 32 768 wird 1 verbraucht
- 32 768 + 7 232 = - 25 536
```

Beispiel: Datentyp short

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

int main()
{
        short x = 32767;
        cout << "x = " << x << endl;          // Programmausgabe: x=32767
        x = x + 1;
        cout << "x = " << x << endl;          // Programmausgabe: x= -32768
        system("pause");

}
```

2.3.5 Cast-Operator

Der Ergebnistyp arithmetischer Verknüpfungen kann auch mit dem cast-*Operator* festgelegt werden. In Klammern steht der Datentyp, in den umgewandelt werden soll.

Syntax: (typ)(variable)

Beispiel: Cast

Ohne cast-Operator	**Mit cast-Operator**

#include <cstdlib>
#include <iostream>
using namespace std;
 	
int main()
{
int a=5, b=2;
float c;
c=a/b;
 	
cout << "c = " << c << endl;
system("pause");
}
	#include <cstdlib>
	#include <iostream>
	using namespace std;
	
	int main()
	{
	int a=5, b=2;
	float c;
	c=(float)(a)/(float)(b);
	
	cout << "c = " << c << endl;
	system("pause");
	}

Ergebnis: 2	Ergebnis: 2.5

2.3.6 Konstanten

Stellt man das Schlüsselwort *const* vor den Datentyp, so kann eine Variable nur gelesen werden, d.h. der Wert der Variable kann nur bei der Initialisierung festgelegt werden und somit nicht nachträglich im weiteren Quellcodeverlauf geändert werden. Typischerweise werden Konstanten mit diesem Schlüsselwort abgelegt, dessen Wert fest definiert ist.

Beispiel: const double pi = 3.14159;

Konstanten müssen bei der Deklaration initialisiert werden! Sobald Sie eine Variable als const deklariert haben, stellt der Compiler sicher, dass die Variable nicht geändert wird.

2.4 Rechenoperatoren

Operatoren dienen zur Manipulation von Daten. Mit Operatoren werden Berechnungen durchgeführt, Ausdrücke auf Gleichheit geprüft, Zuweisungen vorgenommen, Variablen manipuliert und noch etliche weitere Aufgaben erledigt.

Übersicht über die wichtigsten Rechenoperatoren in C++.
Für Teil 1 dieses Kurses benötigen Sie lediglich die in Fettdruck in blauer Farbe formatierten Operatoren.

Operator	Beschreibung	Beispiel
➢ **Mathematische Operatoren**		
+	**Addition**	**x = y + z;**
-	**Subtraktion**	**x = y – z;**
•	**Multiplikation**	**x = y * z;**
/	**Division**	**x = y / z;**
➢ **Zuweisungsoperatoren**		
=	**Zuweisung**	**x = 20;**
+=	**Zuweisen und addieren**	**x += 20; (wie x = x + 20;)**
-=	**Zuweisen und subtrahieren**	**x -= 20; (wie x = x - 20;)**
***=**	**Zuweisen und multiplizieren**	**x *= 20; (wie x = x * 20;)**
/=	**Zuweisen und dividieren**	**x (= 20; (wie x = x (20;)**
&=	Zuweisen, Bit-weises AND	x &= 0x03;
|=	Zuweisen, Bit-weises OR	x |= 0x03;
➢ **Logische Operatoren**		
&&	**Logisches AND**	**if (x && 0xFF) {...}**
||	**Logisches OR**	**if (x || 0xFF) {...}**
➢ **Gleichheitsoperatoren**		
==	**Gleich**	**if (x == 20) {...}**
!=	**Nicht gleich**	**if (x != 20) {...}**
<	**Kleiner als**	**if (x < 20) {...}**
>	**Größer als**	**if (x > 20) {...}**
<=	**Kleiner oder gleich**	**if (x <= 20) {...}**
>=	**Größer oder gleich**	**if (x >= 20) {...}**
➢ **Unäre Operatoren**		
*	Dereferenzierungsoperator	int x = *y;
&	Referenzierungsoperator	int* x = &y;
~	Bit-weises NOT	x &= ~0x03;
!	**Logisches NOT**	**if (!gueltig) {...}**
++	**Inkrementoperator**	**x++; (entspricht x = x + 1;)**
--	**Dekrementoperator**	**x--; (entspricht x = x - 1;)**
➢ **Klassen- und Strukturoperatoren**		
::	**Gültigkeitsbereichoperator**	**MeineKlasse::Funktion();**
->	Indirekter Auswahloperator	MeineKlasse->Funktion();
.	**Direkter Auswahloperator**	**MeineKlasse.Funktion**

Beispiel: Zuweisungsoperatoren

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

int main()
{
        int x= 2;                   // Der Variablen x wird der Wert 2 zugewiesen
        x+=10;                      // entspricht x = x +10;
        cout << x<< endl;           // Ausgabe des Wertes 12 auf dem Bildschirm

        system("pause");
}
```

Inkrement und Dekrementoperatoren

Der *Präinkrementoperator* (++x) teilt dem Compiler mit, den Wert der Variable zu inkrementieren (die Variable um den Wert 1 zu erhöhen) und dann erst weiterzuverwenden, während beim *Postinkrementoperator* (x++) die Variable zuerst verwendet und dann der Wert um 1 erhöht werden soll. Der Dekrementoperator (--) arbeitet analog (verringert den Wert der Variable um 1).

Aufgabe: Bestimmen Sie die Werte folgender Ausgabeanweisungen und überprüfen Sie ihre Lösung mit dem Ergebnis der Konsolenausgabe:

```
int x = 10;
cout << "x = " << x++ << endl;
cout << "x = " << x << endl;
cout << "x = " << ++x << endl;
cout << "x = " << x << endl;
```

2.5 Funktionen

Funktionen sind Programmabschnitte, die getrennt vom Hauptprogramm stehen. Diese werden aufgerufen (ausgeführt), wenn sie in einem Programm für die Durchführung bestimmter Aktionen benötigt werden. Sie könnten zum Beispiel eine Funktion haben, die zwei Werte übernimmt, mit diesen eine komplexe mathematische Berechnung durchführt und dann das Ergebnis zurückliefert.

Funktionen sind ein wesentlicher Bestandteil einer jeden Programmiersprache und C/C++ bilden da keine Ausnahme. Der einfachste Typ von Funktionen erfordert keine *Parameter* und liefert *void* zurück (das heißt, es wird nichts zurückgeliefert). Andere Funktionen erfordern einen oder mehrere *Parameter* und können auch einen Wert zurückliefern. Funktionen unterliegen den gleichen Namenskonventionen wie Variablen. Ein *Parameter* ist ein Wert, der einer Funktion übergeben wird und dazu dient, deren Ablauf zu beeinflussen oder das Ausmaß ihrer Operation anzuzeigen.

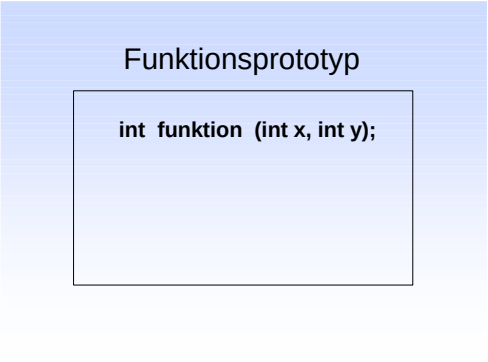
Merke:
Eine Funktion kann keinen (void) oder einen Wert zurückliefern.
Eine Funktion kann einen oder mehrere Eingabeparameter haben.

2.5.1 Deklaration einer Funktionsanweisung (Prototyp):

Bevor eine Funktion verwendet werden kann, muss sie erst einmal deklariert werden. Die *Funktionsdeklaration* (auch *Prototyp* genannt) teilt dem Compiler mit, wie viele Parameter die Funktion aufnimmt und welche Datentypen die einzelnen Parameter und der Rückgabewert der Funktion haben.

- ret_typ funktions_name(argtyp_1 arg_1, argtyp_2 arg_2, ..., argtyp_n arg_n);

Die Funktionsdeklaration identifiziert eine Funktion, die in den Code mit aufgenommen werden soll. Sie gibt den Datentyp des Rückgabewerts (ret_typ) und den Funktionsnamen (funktions_name) an. Darüber hinaus legt sie die Reihenfolge (arg_1, arg_2, ..., arg_n) und Typen (argtyp_1, argtyp_2, ..., argtyp_n) der Datenargumente fest, die die Funktion erwartet.



Im Prototyp muss mindestens der Datentyp des Rückgabewertes und die Datentypen der Eingabeparameter angegeben werden:

Beispiele:
```
int funktion(int,float,float);     // 1 Rückgabewert, 3 Eingabeparameter
void funktion();                   // keine Eingabeparameter, kein Rückgabewert
```

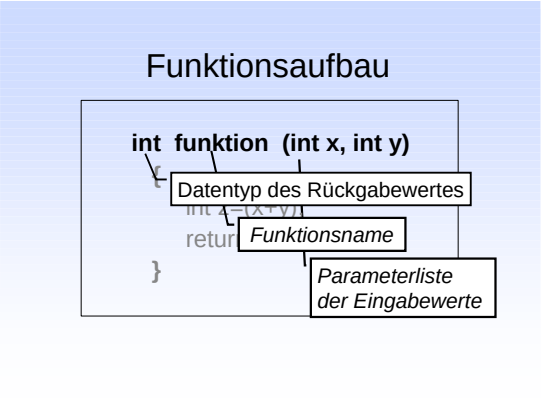
Auf den Funktionsprototyp kann verzichtet werden, wenn die Funktion erst nach ihrer Definition (siehe unten) aufgerufen wird.

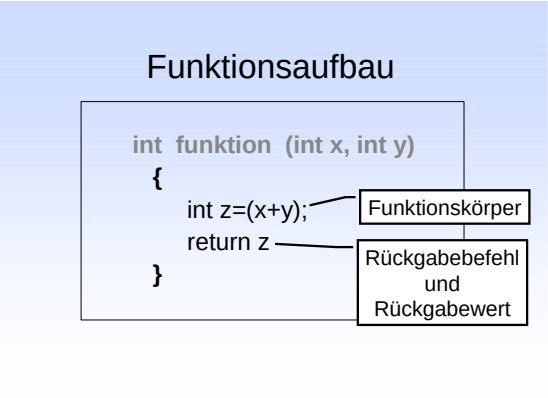
2.5.2 Definition einer Funktionsanweisung:

```
ret_type function_name(argtype_1 arg_1, argtype_2 arg_2,..., argtype_n arg_n)
{
    Anweisungen;
    return ret_type;      // Typ der Variable bzw. Wert muss gleich sein wie ret_type
}
```

Die Funktionsdefinition identifiziert den Codeblock (Anweisungen), aus dem die Funktion besteht, und gibt den Datentyp des Rückgabewerts (ret_typ) an. Der funktions_name identifiziert die Funktion. Die Parameter, die der Funktion übergeben werden (arg_1, arg_2, ..., arg_n), und ihre Datentypen (argtype_1, argtype_2, ..., argtype_n) sind ebenfalls in der Definition enthalten.

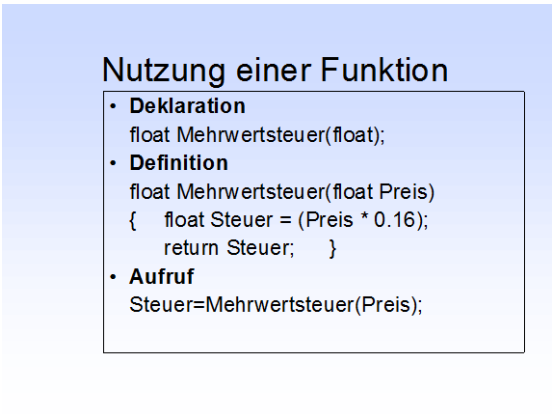
Enthält der Funktionsprototyp einen Rückgabewert, so muss die Definition eine *return*-Anweisung enthalten, welche den Wert zurückgibt.
Bei Rückgabewert *void* kann auf die return-Anweisung verzichtet werden oder es kann return 0; stehen.
Wenn Sie versuchen, einen Wert aus einer Funktion mit Rückgabetyp void zurückzuliefern, erhalten Sie einen Compiler-Fehler. Wurde keine return-Anweisung vorgesehen, kehrt die Funktion automatisch zurück, wenn das Ende des Funktionsblocks erreicht ist (die schließende geschweifte Klammer).





2.5.3 Funktionsaufruf

Eine Funktion wird mit ihrem Namen gefolgt von der Liste der übergebenen Parameter (innerhalb von Klammern, durch Kommata getrennt) aufgerufen.



Deklaration, Definition und Aufruf einer Funktion sind eigenständige Bausteine und damit wiederverwendbar. So kann der Prototyp (Deklaration) auch in einer Header-Datei abgelegt und für mehrere Programme nutzbar gemacht werden.
Eine Funktion kann in der main-Funktion mehrmals mit verschiedenen Eingabewerten aufgerufen werden und liefert dann das dementsprechende Ergebnis.
Zu beachten ist jedoch, dass dabei die Anzahl und der Datentyp der Eingabeparameter übereinstimmen müssen.

Beispiel: Funktionen zur Multiplikation

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

int multiply(int, int);                    // Prototypen der Funktionen multiply und showResult
void showResult(int);

int main()
{
        int ergebnis;                      // Deklaration einer Variablen

         ergebnis = multiply(3, 4);        // Aufruf der Funktion multiply;  mit den Konstanten 3 und 4 als Eingabeparameter
                                           // Das Ergebnis der Berechnung wird der Variablen ergebnis zugewiesen
         showResult(ergebnis);             // Aufruf von showResult mit der Variablen ergebnis als Eingabeparameter

        cout << endl;
        system("pause");
        return 0;
}

int multiply(int a, int b)                 // Definitionen der Funktionen multiply und showResult
{
    return a * b;                          // Rückgabewert ist das Ergebnis der Multiplikation der Variablen a und b
}

void showResult(int result)
{
    cout << "Ergebnis der Multiplikation: " << result << endl;      // kein Rückgabewert nur Bildschirmausgabe
}
```

Erläuterung:
Jedes Programm beginnt mit der *main*-Funktion.
Dieses Programm übergibt der Funktion multiply() die Werte 3 und 4 und ruft diese auf, um die zwei Zahlen miteinander zu multiplizieren. Dann wird die Funktion showResult() aufgerufen, um das Ergebnis anzuzeigen. Die beiden *Prototypen* für die Funktionen multiply() und showResult() stehen direkt über dem Hauptprogramm main(). Der Prototyp gibt nur den Typ des Rückgabewerts, den Funktionsnamen und den Datentyp der Funktionsparameter an. Dies ist die Mindestanforderung an die *Deklaration* einer Funktion. Falls gewünscht, können im Funktionsprototyp auch Variablennamen mit aufgenommen werden, um die Arbeitsweise der Funktion besser zu dokumentieren. Die Funktion multiply() hätte auch wie folgt deklariert werden können:

```
int multiply(int firstNumber, int secondNumber);
```

In diesem Fall ist es zwar ziemlich offensichtlich, wozu die Funktion multiply() eingesetzt wird, aber es kann nicht schaden, Ihren Code durch Kommentare und durch den Code selbst zu dokumentieren.

Die Definition der Funktion multiply() befindet sich außerhalb des Programmbereiches der main-Funktion. Die *Funktionsdefinition* enthält den eigentlichen Funktionskörper. Im Beispiel ist der Körper der Funktion minimal, da die Funktion lediglich zwei Funktionsparameter multipliziert und das Ergebnis zurückliefert.
Die multiply()-Funktion kann auf verschiedene Weisen aufgerufen werden. Im obigen Beispiel wurde die Funktion mit Konstanten aufgerufen. Sie können aber auch Variablen, oder sogar die Ergebnisse von anderen Funktionsaufrufen übergeben.

Beispiel: Variablen als Parameter

```
#include <cstdlib>
#include <iostream>

using namespace std;

int multiply(int, int);                         // Prototypen der Funktionen multiply und showResult
void showResult(int);
int main()                                      // Programm beginnt mit main-Funktion
{
        int x, y, ergebnis;                     // Deklaration von 3 Variablen

        cout << "Bitte geben Sie den ersten Wert ein: ";        // Ausgabe am Bildschirm
        cin >> x;                                               // Eingabe des Wertes von x durch den Benutzer
        cout << "Bitte geben Sie den zweiten Wert ein: ";               // Ausgabe am Bildschirm
        cin >> y;                                               // Eingabe des Wertes von y durch den Benutzer

        ergebnis = multiply(x, y);      // Aufruf der Funktion multiply; ergebnis wird der Variablen result zugewiesen

        showResult(ergebnis);           // Rückgabewert der Funktion showResult
                                        // mit der Variablen ergebnis als Eingabeparameter

        cout << endl << endl << "Weiter mit Tastendruck  ...";  // Ausgabe am Bildschirm
        system("pause");                                        // Programmende
        return 0;
}

int multiply(int a, int b)                                      // Definition der Funktion multiply
{
    return a * b;                                               // Rückgabewert ist Ergebnis der Multiplikation von a und b
}

void showResult(int result)                                     // Definition der Funktion showResult
{
     cout << "Ergebnis der Multiplikation: " << result;          // kein Rückgabewert (Datentyp void)
                                                                // nur Textausgabe am Bildschirm
}
```

Beispiel: Rückgabewert (von *multiply)* dient als Parameter für eine andere Funktion (*show-Result*)

```
#include <cstdlib>
#include <iostream>

using namespace std;

int multiply(int, int);
void showResult(int);

int main()
{
        int x=3, y=4;

        showResult(multiply(x,y));      // Rückgabewert der Funktion multiply dient als
                                        // Eingabeparameter für die Funktion showResult
        cout << endl;
        system("pause");
        return 0;
}

int multiply(int a, int b)
{
    return a * b;
}

void showResult(int result)
{
     cout << "Ergebnis der Multiplikation: " << result << endl;
}
```

Funktionen können andere Funktionen und sogar sich selbst aufrufen. Man spricht dann von *Rekursion.* Bei den in diesem Abschnitt vorgestellten Funktionen handelt es sich ausschließlich um alleinstehende Funktionen *(alleinstehend* insofern, als sie nicht Element einer Klasse sind). Die Anwendung alleinstehender Funktionen in C++ unterscheidet sich in keiner Weise von C, wenn auch in C++ das Leistungsspektrum der Funktionen etwas erweitert wurde.

Regeln für Funktionen:
- Eine Funktion kann keinen oder eine beliebige Anzahl von Parametern übernehmen, sie kann jedoch nur einen oder keinen Wert zurückliefern.
- Definiert eine Funktion einen Rückgabewert vom Typ void, kann sie keinen Wert zurückliefern.
- Wird im Funktionsprototyp angegeben, dass die Funktion einen Wert zurückliefert, sollte der Funktionskörper eine return-Anweisung enthalten, die einen Wert zurückliefert. Ansonsten gibt der Compiler eine Warnung aus.
- Variablen können den Funktionen als Wert, über Zeiger oder über Referenzen übergeben werden (wird in Teil 2 behandelt).

2.5.4 Gültigkeitsbereich von Variablen

Der Begriff *Gültigkeitsbereich* bezieht sich auf die Sichtbarkeit von Variablen innerhalb der verschiedenen Teile Ihres Programms. Die meisten Variablen haben einen *lokalen Gültigkeitsbereic*h, was bedeutet, dass die Variable nur innerhalb des Anweisungsblocks, in dem sie deklariert wurde, sichtbar oder gültig ist.

Beispiel: Testprogramm für Gültigkeitsbereiche von Variablen

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

int a,b;                                    // Deklaration der globalen Variablen a und b

void aendern()                              // Definition der Funktion aendern()
{
    int a;                                  // Deklaration einer lokalen Variablen a
    a=0;
    {                                       // Zweiter Anweisungsblock
        int a=20;                           // Deklaration einer lokalen Variablen a
        cout << "a im zweiten Anweisungsblock ist: " << a << endl;
    }
    cout << "a in aendern ist: " << a << endl;
}

int main()                                  // Aufruf der main-Funktion = Programmstart
{
    a=b=10;                                 // den globalen Variablen a und b wird der Wert 10 zugewiesen

    cout << "Vor Aufruf der Funktion aendern ist a: " << a << " und b: " << b << endl;
    aendern();                              // Aufruf der Funktion aendern()
    cout << "Nach Aufruf der Funktion aendern ist a: "<< a << " und b: " << b << endl;;
    system("pause");
    return 0;
}
```

Programmausgabe:

Vor Aufruf der Funktion aendern ist a: 10 und b: 10
a im zweiten Anweisungsblock ist 20
a in aendern ist 0
Nach Aufruf der Funktion aendern ist a: 10 und b: 10

Erläuterung:
In der Funktion *aendern()* wird zweimal die Variable a deklariert (Zeile7 und Zeile11).
In der Regel gibt der Compiler einen Fehler aus, wenn eine Variable zufällig mehr als einmal deklariert wurde *(Multiple declaration for 'a*') und bricht das Kompilieren ab. Bei diesem Programm jedoch verlaufen Kompilation und Ausführung erfolgreich. Sie werden sich fragen, warum?
Weil jede der Variablen *a* einem anderen Gültigkeitsbereich angehört und damit ***lokale Variablen*** sind. Lokale Variablendefinitionen müssen immer die ersten Anweisungen des Blocks sein, in dem sie stehen. Eine Variable, die innerhalb eines Anweisungsblocks definiert wurde, ist lokal und gilt nur innerhalb dieses Blocks (z.B. hier im zweiten Anweisungsblock). Gibt es mehrere Variablen mit gleichem Namen, spricht man über den Namen die lokalste (aus dem innersten Block) der Variablen an.

Merke:
Wird ein Anweisungsblock verlassen, werden alle *lokalen Variablen*, die in ihm definiert wurden gelöscht.

Zum Schluss gehen wir noch auf die Deklaration der a-Variablen in Zeile 3 ein. Da diese Variable außerhalb aller Funktionen deklariert wurde, heißt sie auch ***globale Variable*** und hat auch einen *globalen Gültigkeitsbereic*h. Konkret bedeutet dies, dass die globale Variable a überall in dem Programm verfügbar ist: innerhalb der main()-Funktion, innerhalb des zweiten Anweisungsblocks und innerhalb der Funktion aendern().

Wichtig:
Definieren Sie Variablen so lokal wie möglich und so global wie nötig. Das ist eine der Grundregeln der modularen Programmierung.

2.5.5 Inline Funktion

Der Aufruf einer Funktion benötigt eine gewisse Zeit. Die Rücksprungadresse wird auf einen Stapelspeicher, auch genannt „Stack“, gelegt. Die Parameter werden ebenfalls auf den Stack gelegt. Das Programm springt an eine Stelle und nach Funktionsende wieder zurück zu der Anweisung nach dem Aufruf. Der absolute Verwaltungsaufwand macht sich bemerkbar, sobald die Anzahl der Aufrufe steigt, z.B. in Schleifen oder wenn es sich um zeitkritische Anwendungen handelt. Um einen solchen Aufwand zu vermeiden, kann eine Funktion als inline deklariert werden. Dieses bewirkt, dass bei der Compilierung der Aufruf durch den Funktionskörper ersetzt wird. Das bedeutet, dass kein echter Funktionsaufruf erfolgt. Die Syntaxprüfung bleibt erhalten und die Parameter werden entsprechend ersetzt.

Zu beachten ist, dass die inline Deklaration nur für Funktionen mit einem Funktionskörper kurzer Ausführungszeit im Vergleich zum Aufwand für den Aufruf geeignet ist. inline ist auch nur eine Empfehlung an den Compiler, um die Ersetzung vorzunehmen, er muss sich aber nicht daran halten. Wenn der Compiler feststellt, dass eine Ersetzung keine Laufzeitvorteile bringt, so steht es ihm frei, entweder die inline Funktion zu übersetzen oder auch nicht.

Beispiel: Inline Funktion

```
#include <cstdlib>
#include <iostream>

using namespace std;

inline int maximum (int a, int b) {
      return a > b ? a : b;
}

int main()
{
   cout << maximum(4,7) << endl;
   cout << maximum(3,9) << endl;

   system("pause");
   return 0;
}
```

Im Beispiel wird eine maximum Funktion nicht als Funktion übersetzt, sondern es wird der Programmcode der Funktion direkt an der Aufrufstelle eingefügt. Dadurch wird ein Sprung zur Funktion, das Kopieren der Parameter und der Rücksprung gespart. Der übersetzte Programmcode schaut so aus, als würde er direkt an der Aufrufstelle (in main Funktion) stehen.

Beispiel:

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main()
{
   cout << (4 > 7 ? 4 : 7) << endl;
   cout << (3 > 9 ? 3 : 9) << endl;

   system("pause");
   return 0;
}
```

2.6 Ein- und Ausgabe

2.6.1 I/O-Streams

Zusätzlich zum Ein- und Ausgabesystem von C verfügt C++ über neue Streams auf der Basis von Klassen. Der Begriff Stream (dt. Strom) spiegelt die Situation wider, dass Zeichenfolgen bei der Ein- und Ausgabe als Datenstrom behandelt werden.
Ein wesentlicher Vorteil der neuen Streams ist die Typsicherheit:
Bei den klassischen C-Funktionen, wie zum Beispiel *printf(),* kann der Compiler nicht immer den richtigen Aufruf überprüfen. Ein falsches Formatelement oder der falsche Typ eines Arguments führen unweigerlich zu einem Laufzeitfehler.

Beispiel:

```
double x = 1.5;
printf("/d", x);          // Fehler!
                          // Falsches Formatelement.
```

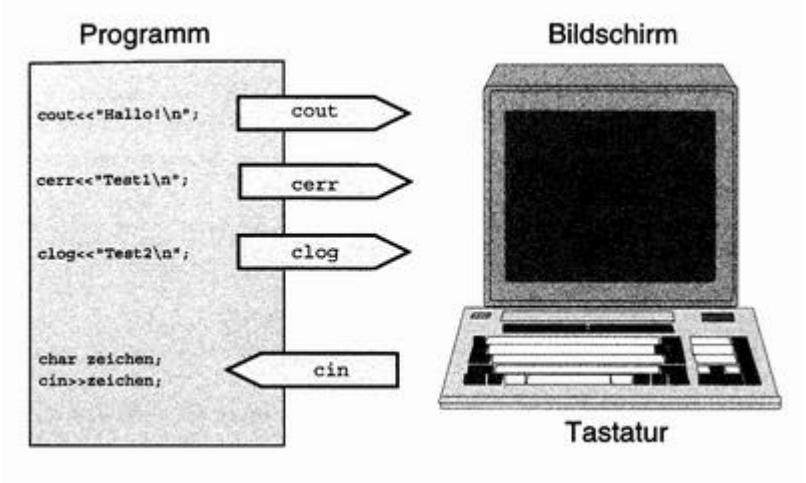
Derartige Fehler sind bei Verwendung der neuen Streams ausgeschlossen: Der Compiler entscheidet anhand des Argumenttyps, welche Eingabe- bzw. Ausgabe-Routine aufzurufen ist.

Bildschirmausgabe mit printf()	**Bildschirmausgabe mit cout**

float zahl = 12.345f;
printf(„Die Zahl heißt: %f\n“, zahl);
	float zahl = 12.345f; // hier wird zum Schluss f
	// verwendet, da standardmäßig
	// der Speicherraum für double
	// Variablen verwendet wird,
	// welcher größer als float ist. Wir
	// erzwingen also den Speicher-
	// raum für float zu verwenden.
	
	cout << “Die Zahl heißt:“; // ein String
	cout << zahl; // ein float-Wert
	cout << ’\n’ // eine neue Zeile
	
	// Normalerweise schreibt man nur eine Anweisung:
	cout << “Die Zahl heißt:“ << zahl << ’\n’;

2.6.2 Standard-Ein- und Ausgaben

Die Klassen für die neuen Streams sind in der Header-Datei *iostream* deklariert, und es gibt folgende Objekte:



cout: Standard-Ausgabe

cerr: Standard-Fehlerausgabe

clog: Standard-Fehlerausgabe

cin: Standard-Eingabe

Im Unterschied zu cerr ist der Stream clog gepuffert und wird normalerweise für Protokollausgaben verwendet.
Alte und neue Stream-Funktionen sollten in einem Programm nicht gemischt werden, d.h. wenn beispielsweise für die Eingabe cout benutzt wird, sollte an anderer Stelle nicht printf( ) aufgerufen werden.
Im Folgenden wird gezeigt, wie die neuen Streams zu verwenden sind. Alle in der C++-Standard-Bibliothek definierten globalen Bezeichner gehören zum Namensbereich std. Im Anschluss an die Direktive

- using namespace std;

können diese Bezeichner direkt angesprochen werden.

Anstatt die Bezeichner mit *std::cout* anzusprechen, kann man durch Benutzen des Namensraums lediglich *cout* verwenden.

2.6.3 Ausgaben mit cout

Die Anweisung *cout << "Hello world ! \n";* bewirkt, dass der String "Hello world ! \n" zur Standard-Ausgabe (also auf den Bildschirm) „geschoben“ wird. Mit seiner Richtung zeigt der Shift-Operator << an, dass nach links in den Ausgabestream geschoben wird.

Ebenso wie Strings können auch Werte mit einem anderen Datentyp ausgeben werden.

Beispiel:
```
int zaehler = 10;
cout << zaehler;        // Ausgabe: 10
```

Das Objekt cout ist „intelligent“ genug, den Typ der Daten zu erkennen: Mit cout und dem Shift-Operator << können außer Strings alle Werte ausgegeben werden, die einen elementaren Datentyp haben. Darüber hinaus kann der Anwendungsbereich des Operators << jederzeit auf andere Datentypen, auch selbstdefinierte Datentypen, ausgedehnt werden. Dazu muss der Operator << überladen werden. Diese Technik wird im Kursteil 2 behandelt. Der Ausdruck *cout << wert;* repräsentiert selbst wieder das Objekt cout. Daher kann der Operator << auch mehrfach hintereinander angewendet werden.

Beispiel: cout << "Dieser String und ein Zeichen: " << 'A';

In diesem Beispiel wird die Zeichenkonstante (einzelnes Zeichen) 'A' benutzt. Hier ist der folgende Unterschied zwischen C und C++ zu beachten:

C: Eine Zeichenkonstante ist vom Typ int.
C++: Eine Zeichenkonstante ist vom Typ char.

Am Bildschirm wird also wirklich A ausgegeben, und nicht etwa der Zeichencode von ' A'.

Um Umlaute in C++ ausgeben zu können, muss man diese über die Escapesequenzen und anschließend dem Hexadezimal-/Oktalwert ansprechen.

Zeichen	Hex	Okt
Ä	8E	216
ä	84	204
Ö	99	231
ö	94	224
Ü	9A	232
ü	81	201
ß	E1	341

Eine Ausgabe mit dem Umlaut 'ö' unter Verwendung der Hexadezimalschreibweise würde wie folgt aussehen:

```
cout << "Heute ist ein sch\x94ner Tag!" << endl; // \x leitet eine Hexadezimal-Escapesequenz ein
```

Diese Version hat allerdings einen Haken: Ist das nächste Zeichen ebenfalls hexadezimal bzw. oktal darstellbar, wird dieses Zeichen mitinterpretiert. Zum Beispiel würde im Wort "Oberfl\x84che" nicht '\x84' mit 'ä' ersetzt werden sondern 'x84c' mit der Zahl '2124'. Für dieses Problem gibt es allerdings einen kleinen Trick:

```
cout << "Oberfl\x84 \bche!" << endl;
```

Nach dem Hex-Code fügt man ein Leerzeichen hinzu, dass sofort danach wieder mit der Escapesequenz '\b' für die Rücktaste wieder gelöscht wird.

2.6.4 Eingaben mit cin

Das Lesen von der Standard-Eingabe (also von der Tastatur) erfolgt mit cin und dem Operator >> .

Beispiel:
```
float var;
cin >> var;      // Gleitpunktzahl in var einlesen
```

Die Daten werden also von cin in die Variable var „geschoben“. Über den Typ der Variablen legt cin selbständig fest, was als Eingabe akzeptiert wird und wie die Konvertierung erfolgt. Auf diese Weise können Werte mit einem elementaren Datentyp und Strings eingelesen werden.

Im Gegensatz zur Eingabe mit scanf() (entsprechender Befehl in C) können keine leichtsinnigen Fehler durch die Wahl falscher Formatelemente gemacht werden.
Das obige Beispiel ist wirkungsgleich mit folgender scanf-Anweisung:

```
float var;
scanf("%f", &var);  // Gleitpunktzahl in var einlesen
```

Beim Einlesen mit cin und dem Operator >> gelten folgende Regeln:
- Führende Zwischenraumzeichen werden überlesen.
- Die Verarbeitung der Eingabe bricht ab, wenn ein Zeichen nicht verarbeitet werden kann.

Beispiel:
```
float var;
cin >> var; // Gleitpunktzahl in var einlesen
```

Erfolgt hier die Eingabe 1.20 € so wird 1.20 in der Variablen var gespeichert, und alle weiteren Zeichen bleiben im Eingabepuffer. Bei der Eingabe von € 1.20 würde kein Wert an var zugewiesen und die Eingabe bleibt unverarbeitet im Puffer.

2.6.5 Ausgaben mit cerr

Die Standard-Fehlerausgabe cerr bietet dieselben Möglichkeiten wie cout. Insbesondere sendet auch cerr seine Ausgabe normalerweise zum Bildschirm.

Die Unterscheidung zwischen cout und cerr ist dann wichtig, wenn Ausgaben umgelenkt werden (z.B. in eine Datei). Fehlermeldungen sollten deshalb über cerr ausgegeben werden. Nur so ist sichergestellt, dass diese auch dann noch auf dem Bildschirm erscheinen, wenn die Standard-Ausgabe umgeleitet wurde.

Beispiel:
```
float var;
if( ! (cin >> var) )        // Wenn kein Wert eingelesen wurde
cerr << "Fehler: Keine Zahl eingelesen!\n";
```

Das Programm kann feststellen, ob ein Wert eingelesen wurde. Erfolgt die Eingabe nämlich in einer Verzweigungs- oder Laufbedingung, so liefert der Ausdruck »wahr«, wenn die Eingabe erfolgreich war, andernfalls »falsch«.

3 Kontrollstrukturen

In Kapitel 3 lernen Sie, wie man in C++ Verzweigungen und Schleifen programmiert.

3.1 Verzweigungen

3.1.1 if-Anweisung

Die if-Anweisung wird verwendet, um eine Bedingung zu prüfen und dann, in Abhängigkeit davon, ob die Bedingung „wahr“ oder „falsch“ ist, bestimmte Programmabschnitte auszuführen.

Syntax:
```
if(Bedingung){
}
```

Zur Formulierung der Bedingung werden Vergleichsoperatoren benötigt (siehe Rechenoperatoren 2.4).

Wichtig:
- Bitte beachten Sie bei der Abfrage auf Gleichheit die spezielle Schreibweise „==“.
- Mit dem !-Operator vor der Bedingung können Sie Bedingungen negieren.
- Eine Bedingung darf in einem berechenbaren Ausdruck stehen. Sie nimmt, für „wahr“ den Wert 1 und für „falsch“ den Wert 0 an.

Beispiel:
```
int x;
cout << "Geben Sie eine Zahl ein: ";
cin >> x;
if (x > 10)
{
        cout << "Die eingegebene Zahl ist größer als 10." << endl;
}
```

Erläuterung:
Der Benutzer wird zunächst aufgefordert, eine Zahl einzugeben. Ist diese Zahl größer als 10, wird die Bedingung x > 10 auf „wahr“ ausgewertet und der Text angezeigt; im anderen Falle wird nichts angezeigt. Beachten Sie, dass für den Fall, dass der bedingte Ausdruck „wahr“ ergibt, die unmittelbar auf den if-Ausdruck folgende Anweisung ausgeführt wird. Besteht der Anweisungsblock nur aus einer Anweisung, so können die geschweiften Klammern weggelassen werden.

Sie können mit einer if-Anweisung je nachdem, ob die Bedingung „wahr“ beziehungsweise „falsch“ ist, unterschiedliche Operationen ausführen lassen. Insgesamt gibt es sechs Vergleichsoperatoren. Sie liefern jeweils den Wert true, wenn die beiden Operanden (die links und rechts des Operators stehen) dem Vergleichskriterium genügen, ansonsten den Wert false.

==	identisch
<=	ist kleiner (oder) gleich
>=	ist größer (oder) gleich
<	ist kleiner
>	ist größer
!=	ist ungleich

Beispiel:
```
if (x == 20) {
        cout << "Die eingegebene Zahl ist 20." << endl;
        }
else if (x <= 19) {
        cout << "Die eingegebene Zahl ist kleiner oder gleich 19." << endl;
else {
        cout << "Die eingegebene Zahl ist größer als 20." << endl;
        }
```

Erläuterung:
Das Schlüsselwort else erweitert die Einsatzmöglichkeiten der Verzweigung. Der Code nach else wird nur ausgeführt, wenn die vorherigen Bedingungen falsch sind. Sie können in einer Verzweigungsanweisung auch mehr als zwei Alternativen angeben. Durch das else if kommt eine zweite Verzweigung zustande, falls die erste Abfrage falsch war. Es können beliebig viele Zweige mit else if vorkommen.

Sie können ebenfalls mehrere Bedingungen zu einem Ausdruck verknüpfen. Hierfür gibt es folgende Möglichkeiten:

!	NICHT	Resultat wahr, wenn der Operand falsch ist
&&	UND	Resultat wahr, wenn beide Operanden wahr sind
||	ODER	Resultat wahr, wenn mindestens ein Operand wahr ist

Beispiel:
```
int i = 10, j = 20;
if ((i == 10) && (j == 10))
{
        cout << "Beide Werte sind gleich zehn" << endl;
}
else if ((i == 10) || (j == 10))
{
        cout << "Ein Wert oder beide Werte sind gleich zehn" << endl;
}
else if ((i != 10) && !(j == 10))
{
        cout << "Beide Werte sind ungleich zehn" << endl;
}
```

Hinweis:
In C/C++ gibt es viele Möglichkeiten, Befehle abzukürzen. Eine davon ist die Überprüfung der Bedingung auf „wahr“, bei der lediglich der Variablenname verwendet wird.

Beispiel:
```
bool dateiOK;
if (dateiOK) DatenLesen();

Dies ist die Kurzform der folgenden Zeile:
if (dateiOK == true) DatenLesen();
```

3.1.2 Bedingter Ausdruck

Häufig werden Verzweigungen eingesetzt, um abhängig vom Wert eines Ausdrucks eine Zuweisung vorzunehmen. Das können Sie mit dem ?-Operator auch einfacher formulieren.

Beispiel:
```
int min;
min = (a < b) ? a : b;
```

Erläuterung:

Die Syntax hierzu ist:

Wert = Abgefragte Bedingung ("Frage") **?** Rückgebewert für WAHR **:** Rückgabewert für FALSCH ;

Im obigen Beispiel also: Ist a kleiner als b? Wenn ja (d.h. a kleiner), dann a zuweisen, wenn nein (d.h. b kleiner), dann b zuweisen.

3.1.3 switch-Anweisungen

Die *switch*-Anweisung kann man auch als erweiterte if-Anweisung betrachten. Sie erlaubt es, in Abhängigkeit von dem Wert eines bestimmten Ausdrucks, zu verschiedenen Anweisungsblöcken zu springen. Der Ausdruck kann eine Variable, das Ergebnis eines Methodenaufrufs oder ein gültiger C/C++-Ausdruck sein. Der Default-Zweig ist optional.

Syntax einer switch-Anweisung

```
switch(Ausdruck)
    {
    case x1:    <Anweisung;>
                <break;>
    case x2:    <Anweisung;>
                <break;>
     .....
    default:    <Anweisung;>
                <break;>
     }
```

Erläuterung:

- Die x_i (Sprungmarken) einer case-Anweisung müssen ganzzahlige Konstanten (Zahlen oder einzelne Buchstaben z.B.: case 'A' sein.)
- Sie können mehrere case-Anweisungen mit gleichem Ziel zusammenfassen, indem Sie gezielt break-Anweisungen einsetzen oder weglassen.
- Es können beliebig viele case-Anweisungen das gleiche Sprungziel haben.
- Existiert für einen Ausdruck der switch-Anweisung keine case-Sprungmarke x_i, dann wird, falls vorhanden, die default-Sprungmarke angesprungen.

Aufgabe: Ergänzen Sie den folgenden Quelltextauszug zu einem lauffähigen Programm und experimentieren Sie damit!

```
switch(UeberGeschwindigkeitsbegrenzung) {
    case 0 : {
    Strafe = 0;
    break;
    }
    case 10 : {
    Strafe = 20;
    break;
    }
    case 15 : {
    Strafe = 50;
    break;
    }
    case 20 :
    case 25 :
    case 30 : {
    Strafe = UeberGeschwindigkeitsbegrenzung * 10;
    break;
    }
    default : {
    Strafe = Vorladung();
    Strafmass = Urteilsspruch();
    }
}
```

Erläuterung:
Die switch-Anweisung setzt sich aus mehreren Teilen zusammen. Der erste Teil besteht aus dem Ausdruck – in unserem Falle die Variable UeberGeschwindigkeitsbegrenzung. Danach folgen die case-Anweisungen, die den Ausdruck auf Gleichheit prüfen.
Ist die Variable *UeberGeschwindigkeitsbegrenzung* gleich 0 (case 0 :), wird der Variablen *Strafe* der Wert 0 zugewiesen. Beträgt die *UeberGeschwindigkeitsbegrenzung* 10, wird *Strafe* der Wert 20 zugewiesen und so weiter.
In jedem der ersten drei case-Blöcke finden Sie eine break-Anweisung, mit der Sie aus einem switch-Block aussteigen können.
Nachdem also eine Übereinstimmung mit einem der case-Fälle gefunden wurde, wird der Rest der switch-Anweisung übergangen. Am Ende befindet sich die default-Anweisung. Der Anweisungsblock, der auf die default-Anweisung folgt, wird ausgeführt, wenn in keinem der case-Fälle eine Übereinstimmung gefunden werden konnte.
Beachten Sie, dass hinter case 20 und case 25 keine Anweisungen folgen. Beträgt also der Ausdruck *UeberGeschwindigkeitsbegrenzung* 20 oder 25, werden diese case-Fälle übergangen, und der nachfolgende Block wird ausgeführt. Das heißt konkret, dass für die Werte 20, 25 und 30 jeweils derselbe Code ausgeführt wird.

3.2 Schleifen

Mit einer Schleife kann man ein Feld durchlaufen, eine Aktion mehrmals auszuführen, eine Datei einlesen, usw. Wir werden die for-, die while- und die do-while-Schleifen besprechen.

Schleifen haben folgende Grundelemente:
- einen Einstiegspunkt,
- einen Schleifenkörper, der von Klammern umschlossen ist und die Anweisungen enthält, die bei jedem Durchgang ausgeführt werden,
- das Schleifenende,
- die Überprüfung einer Bedingung, die festlegt, wann die Schleife verlassen wird.

3.2.1 for-Schleifen

for-Schleifen sind die am häufigsten verwendeten Schleifen.
Sie benötigt drei Parameter: den Ausgangswert (Anfang), die Testbedingung (Bedingung), die festlegt, wann die Schleife abbricht, und den Inkrementausdruck (Anpassen).

Syntax einer for-Schleife

```
for (Anfang; Bedingung; Anpassen)
{
Anweisung;
}
```

Die for-Schleife führt den Anweisungsblock Anweisungen solange aus, wie der bedingte Ausdruck „Bedingung“ wahr (ungleich Null) ist. Initialisiert wird die Schleife mit der Anweisung „Anfang“. Nach der Ausführung der Anweisungen wird der Status der Schleife nach Vorgabe der Anweisung „Anpassen“ geändert.

Beispiel:

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

int main()
{
    cout << endl << "Programmstart FOR..." << endl << endl;
    int i;
    cout << "   1. FOR-Schleife " << endl;
          for (i=0;i<12;i++)
          {
                    cout << "Aufstieg  " << i << endl;
          }
    cout << "   2. FOR-Schleife " << endl;
          for (int j=10; j>0; j--)
          {
                    cout << "Abstieg  " << j << endl;
          }
   system("pause");
}
```

Erläuterung:
Die erste Schleife beginnt mit dem Anfangswert 0. Die Anweisung darin wird solange ausgeführt, bis die Variable i den Wert 12 erreicht hat. Bei jedem Durchlauf erhöht sich die Variable i um 1. Bei der zweiten Schleife wird eine neue Variable namens j deklariert und mit dem Anfangswert 10 initialisiert. Die Anweisung wird solange ausgeführt, bis die Variable j den Wert 0 erreicht hat. Bei jedem Schleifendurchlauf verringert sich die Variable j um 1.

3.2.2 while-Schleifen

while-Schleifen enthalten nur eine Testbedingung, die zu Beginn einer jeden Iteration neu geprüft wird. Solange die Testbedingung „wahr“ ist, wird die Schleife durchlaufen.

Syntax einer while-Schleife

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

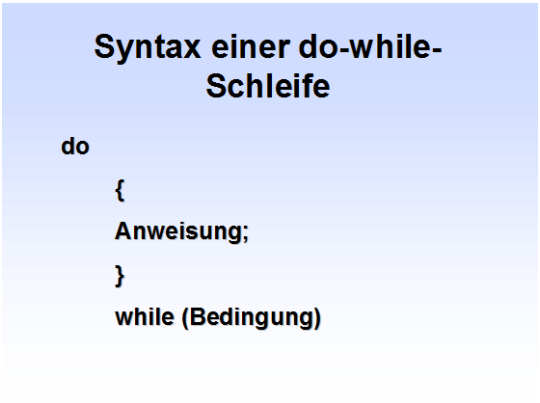
Beispiel:

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

int main()
{
        cout << endl << "Programmstart WHILE ..." << endl << endl;
        int i = 6;

        while (i > 0)
        {
                i--;
                cout << endl << "Ich muss noch " << i;
                cout << " Pakete versenden.";
        }

        cout << "GESCHAFFT !" << endl << endl;

        system("pause");
}
```

Erläuterung:
Bei jedem Schleifendurchlauf verringert sich der Wert der Variable i um 1. Die Schleife wird solange durchlaufen, bis die Variable i den Wert 0 erreicht. Hat sie das, dann ist die Bedingung der Schleife false und sie wird verlassen.

3.2.3 do-while-Schleifen

In der *do-while-Schleife* erfolgt die Überprüfung der Bedingung am Ende der Schleife.



Die do-Schleife wiederholt den Anweisungsblock Anweisungen solange, wie die Bedingung *true* (ungleich Null) ist. Der Status der Schleife muss vor der do-Anweisung initialisiert werden und eine Änderung des Status muss explizit im Anweisungsblock angegeben werden. Wird der bedingte Ausdruck erfüllt, bricht die Schleife ab.

Beispiel:

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

int main()
{
	cout << endl << "Programmstart DO WHILE ..." << endl << endl;
	int i = 6;

	do {
		i--;
		cout << endl << "Ich muss noch " << i;
		cout << " Pakete versenden.";
	}
	while (i>0);

	cout << "GESCHAFFT !" << endl << endl;

	system("pause");
}
```

Unterschied zwischen while-Schleife und do-while-Schleife:

While	**Do-While**

`int i = 1;`
`while(i<0) {`
`  cout << i << endl;`
`  i++;`
`}`
	`int i = 1;`
	`do {`
	`  cout << i << endl;`
	`  i++;`
	`} while(i<0);`

`Drücken Sie eine beliebige Taste . . .`	`1` `Drücken Sie eine beliebige Taste . . .`
Prüfung vor der Ausgabe. Prüfung ist false, somit Abbruch am Anfang und keine Ausgabe.	Ausgabe vor der Prüfung. Anschließend Prüfung, welche false ist, somit Abbruch.

4 Felder und Zeichenketten

Kapitel 4 beschäftigt sich mit dem Thema „Felder“. Sie lernen, wie man diese in C++ programmiert und wie man Zeichenketten behandelt.

4.1 Felder

Sie können jeden elementaren C++-Datentyp in einem Feld, auch Array genannt, ablegen. Ein Feld ist einfach eine Sammlung von Werten des gleichen Datentyps. Ein Array hat einen Namen und, um auf die einzelnen Elemente des Arrays zugreifen zu können, wird eine Positionsnummer, auch genannt Index, verwendet.

4.1.1 Eindimensionale Felder

Wie jedes andere Objekt muss auch ein Array definiert werden. Die Definition legt den Feldnamen, den Typ und die Anzahl der Feldelemente fest.

Syntax: typ name[anzahl];

Dabei muss beachtet werden, dass anzahl **stets** eine ganzzahlige Konstante oder ein ganzzahliger Ausdruck ist, der nur aus Konstanten besteht. Die anzahl kann nicht durch eine Variable oder später im Programm/Quellcode verändert werden.

Angenommen, Sie benötigen ein Integer-Feld, das fünf Integerwerte aufnehmen soll. Dieses Feld würden Sie wie folgt deklarieren:

- int myFeld[5];

Da jeder Integerwert einen Speicherbereich von 4 Bytes belegt, benötigt das Feld insgesamt 20 Byte.

Jetzt, da Sie ein Feld deklariert haben, können Sie es mit Werten füllen, indem Sie den Feld-Indizierungsoperator [ ] verwenden:

- myFeld[0] = -200;
- myFeld[1] = -100;
- myFeld[2] = 0;
- myFeld[3] = 100;
- myFeld[4] = 200;

Wie aus diesem Code ersichtlich ist, sind Felder in C/C++ 0-basiert. Beim Zugriff auf das Feld beginnt die Zählung bei 0. Das heißt, dass ein Feld mit 5 Elementen den Indizes 0 bis 4 enthält. Somit ist der Index des letzten Feldelements stets um 1 niedriger als die Anzahl der Elemente. Ein Array kann mit jedem beliebigen Datentyp gebildet werden. Es gibt jedoch Ausnahmen, wie z.B. void oder bestimmte Klassen. Im weiteren Verlauf Ihres Programms können Sie mit Hilfe des Feld-Indizierungsoperators auch wieder auf die einzelnen Elemente des Feldes zugreifen:

- int result = myFeld[3] + myFeld[4]; // Ergebnis lautet 300

Es gibt einen kürzeren Weg, ein Feld gleichzeitig zu deklarieren und zu füllen:

- int myFeld[5] = { -200, -100, 0, 100, 200 };

Wenn Sie genau wissen, wie viele Elemente Ihr Feld enthält, und wenn Sie das Feld bei der Deklaration füllen, können Sie die Feld-Größe bei der Deklaration des Feldes weglassen. In diesem Fall würde die Deklaration wie folgt lauten:

- `int myFeld[] = { -200, -100, 0, 100, 200 };`

Dies funktioniert, da der Compiler aus der Liste der zugewiesenen Werte ablesen kann, wie viele Elemente sich in dem Feld befinden und wie viel Speicherplatz er dafür allokieren muss.

Beispiel: Lotto Zahlen in Felder

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

int main()
{
   int lotto[6];
   int i, j;
   bool neueZahl;

   srand(time(NULL));                  // Zufallszahlen anhand der Uhrzeit neu generieren
   for(i=0; i<6; i++)                  // ziehe nacheinander sechs Zahlen
   {
      do                               // wiederhole die Ziehung, bis die neue Zahl
      {                                // nicht mit einer der vorigen identisch ist.
         lotto[i] = rand() % 49 + 1;   // Zufällige Zahl zwischen 1 und 49
         neueZahl = true;              // positive Grundeinstellung
         for (j=0; j<i; j++)
         {                             // durchlaufe alle bisher gezogenen Kugeln
            if (lotto[j]==lotto[i])
            {                          // Hier wurde ein Doppelter entdeckt
               neueZahl = false;
            }
         }
      } while (!neueZahl);
   }
   for (i=0; i<6; i++)
   {
      cout << lotto[i] << " ";
   }
   cout << endl;
}
```

Erläuterung:
Mit srand werden zuerst die Zufallszahlen anhand der aktuellen Uhrzeit neu generiert, da diese ansonsten bei jedem Programmaufruf gleichbleiben würden. Die äußere Schleife mit der Variablen i als Index durchläuft die Anzahl der zu ziehenden Lottozahlen. Die Ziehung selbst erfolgt innerhalb der do-while-Schleife, denn sie soll so lange wiederholt werden, bis man eine Zahl findet, die bisher noch nicht gezogen wurde. Die Entscheidung kann also erst nach der Aktion im Schleifenkörper gefällt werden. Damit ist eine fußgesteuerte Schleife (do-while-Schleife) die Idealbesetzung. Nach der Ziehung erfolgt die Prüfung. In einer inneren for-Schleife werden alle bisher gezogenen Zahlen durchlaufen. Das bedeutet, der Index j startet bei 0 und bleibt kleiner als i, d. h. kleiner als der Index für die aktuell gezogene Zahl. Im Falle einer Gleichheit mit den bisher gezogenen Zahlen wird die boolesche Variable neueZahl auf false gesetzt. Das führt zu einer Wiederholung der Ziehung. Nach jeder Ziehung muss diese Variable auf true gesetzt werden, sonst wird die Schleife nie verlassen, sobald einmal eine Doppelziehung entdeckt wurde.

Das Ende eines Feldes darf nicht überschrieben werden. Eine der mächtigsten Eigenschaften von C/C++ ist der direkte Zugriff auf den Speicher. Aufgrund dieser Eigenschaft wird in C/C++ nicht verhindert, dass Sie in eine bestimmte Speicherposition schreiben, auch wenn Ihr Programm auf diesen Speicherbereich eigentlich keinen Zugriff haben sollte. Der folgende Code ist zwar gültig, führt aber dazu, dass Ihr Programm abstürzt:

Beispiel:

```
int Feld[5];
Feld[5] = 10;
```

Dies ist ein häufiger Fehler, der dadurch entsteht, dass Felder 0-basiert sind. Man ist schnell verleitet anzunehmen, dass das letzte Element des Feldes den Index 5 hat, während es tatsächlich den Index 4 hat. Wenn Sie das Ende eines Feldes überschreiben, wissen Sie nicht, welcher Speicher davon betroffen ist. Das Ergebnis ist bestenfalls unvorhersehbar. Im schlimmsten Fall wird Ihr Programm oder sogar das Betriebssystem abstürzen. Dieses Problem ist nur sehr schwer einzugrenzen, da auf den betroffenen Speicherbereich erst viel später zugegriffen wird und auch erst dann der Absturz erfolgt. Seien Sie also vorsichtig, wenn Sie in ein Feld schreiben!

Regeln für Felder:

- Feld-Indizes beginnen mit 0. Das erste Element im Feld hat also den Index 0, das zweite Element 1, das dritte Element 2 und so fort.
- Feld-Größen müssen Kompilierzeitkonstanten sein. Der Compiler muss zur Kompilierzeit wissen, wie viel Platz er für das Feld allokieren muss. Deshalb können Sie eine Feld-Größe nicht mit einer Variablen deklarieren. Der folgende Code wäre demzufolge nicht gültig und würde einen Compiler-Fehler auslösen:
  ```
  int x = 10;
  int myFeld[x];       // Kompilierfehler
  ```
- Achten Sie darauf, das Ende eines Feldes nicht zu überschreiben.
- Allokieren Sie große Felder eher auf dem dynamischen Speicher (=Heap) als auf dem Stapelspeicher (=Stack) (mehr dazu in Kursteil 2).
- Felder, die auf dem Heap allokiert wurden, können mit Hilfe einer Variablen die Feld-Größe zuweisen:
  ```
  int x = 10;
  int myFeld[x] = new int[x]; // dies ist OK (ebenfalls Stoff von Kursteil 2)
  ```

4.1.2 Mehrdimensionale Felder

Felder können mehrdimensional sein. Für ein zweidimensionales Integer-Feld würden Sie folgenden Code eingeben:

- int meinFeld[3][5];

Damit allokieren Sie Speicher für 15 Integerwerte (insgesamt 60 Byte). Im Prinzip greifen Sie auf die Elemente eines solchen Feldes wie bei einem einfachen Feld zu – nur mit dem Unterschied, dass Sie zwei Feld-Indizierungsoperatoren angeben müssen:

- int x = meinFeld [1][1] + meinFeld [2][1];

Beispiel:

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

int main()
{
        int tabelle[2][3]={{10,11,12},{2,3,4}};         // Feld mit 2 Reihen und 3 Spalten

        cout<<"In der 1. Reihe, 2. Spalte steht " <<tabelle[0][1] <<endl;          // Ergebnis: 11
        cout<<"In der 2. Reihe, 1. Spalte steht " <<tabelle[1][0] <<endl;          // Ergebnis: 2
        cout<<"In der 2. Reihe, 2.Spalte steht " <<tabelle[1][1] <<endl;// Ergebnis: 3

        system("pause");
        return 0;
}
```

Ein weiteres Beispiel wäre um z.B. den Umsatz pro Jahr zu ermitteln. Hierbei geben die definierten Daten (Zahlen) im Array die monatliche Umsatzzahlen an, welche aufaddiert werden.

Beispiel: Umsatz

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

int main()
{
  // Umsatzzahlen für 12 über 2 Jahre hinweg
  // [2] → Zeile, d.h. 2 Jahre, und [12] --> Spalte, Anzahl der Monate im Jahr
   float zahlen [2] [12] = {{20.5, 22, 25.2, 30, 55, 66, 74, 20.55, 22, 25.2, 30, 55},
                            {20.55, 22, 25.2, 55, 8, 99, 44, 22.5, 66.3, 10, 11, 12.5}};
   // Summierte Umsatzzahlen pro Jahr über 2 Jahre hinweg
   float umsatz [2] [1];

   for (int i = 0; i < 2; i++) {
      for (int j = 0; j < 12; j++) {
          umsatz[i][0] += zahlen [i][j];          // Umsatzzahlen der einzelnen Monate zusammenaddieren
                                                  // und in der Variable speichern
      }
   }

   cout << "Umsatz 1. Jahr: " << umsatz[0][0] << endl;
   cout << "Umsatz 2. Jahr: " << umsatz[1][0] << endl;

   system("pause");
   return 0;
}
```

4.2 Strings (Zeichenketten)

C++ kennt zwei Arten von Zeichenketten:

- C-Strings: Felder von char-Variablen
- C++-strings: Klasse *string* der Standardbibliothek

4.2.1 C-Strings

Für ein einzelnes Zeichen existiert der Datentyp *char.*

➢ char zeichen = 'A';

Achten Sie darauf, dass das *A* in einfachen Anführungsstrichen steht. Der Compiler schreibt in die Variable nicht das Zeichen A, sondern den Wert, den der Buchstabe in der ASCII-Tabelle repräsentiert.

Merke: Ein Zeichen innerhalb einfacher Anführungsstriche wird als Wert betrachtet.

Strings werden in C/C++-Programmen durch Felder des Datentyps char repräsentiert. Beachten Sie, dass ein String in doppelten Anführungsstrichen steht.
Sie könnten zum Beispiel einen String einem Zeichen-Feld wie folgt zuweisen:

➢ char text[] = "This is a string.";
Damit reservieren Sie 18 Byte an Speicher und legen den String in diesen Speicherbereich ab. Je nachdem wie pfiffig Sie sind, ist Ihnen vielleicht bereits aufgefallen, dass dieser String eigentlich nur 17 Zeichen enthält. Dass dennoch 18 Byte reserviert werden, liegt darin begründet, dass am Ende eines jeden Strings ein Nullzeichen steht und C/C++ dieser Tatsache Rechnung trägt, indem es bei der Speicherallokation Platz für das Nullterminierungszeichen lässt. Das *Nullterminierungszeichen* ist ein Sonderzeichen, das durch \0 dargestellt wird, was einer numerischen Null 0 entspricht. Wenn das Programm in dem Zeichen-Feld auf eine 0 stößt, interpretiert es diese Position als das Ende des Strings.

Beispiel:

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

int main()
{
        char str[] = "This is a string.";
        cout << str << endl;
        str[7] = '\0';                                  // einfache Anführungszeichen !
        cout << str << endl;
        cout << endl << "Press any key to continue...";
        system("pause");
}
```

Erläuterung:
Am Anfang enthält das Zeichen-Feld die Zeichenkette „This is a string.“, gefolgt von dem abschließenden Nullterminierungszeichen. Dieser String wird mit Hilfe von cout auf dem Bildschirm ausgegeben. In der nächsten Zeile wird dem siebten Element des Feldes der Wert \0, das Nullterminierungszeichen zugewiesen. Der String wird erneut auf dem Bildschirm ausgegeben. Diesmal wird jedoch nur „This is“ angezeigt. Dies liegt darin begründet, dass aus der Sicht des Computers der String bei dem Element 7 im Feld endet. Die restlichen Zeichen befinden sich noch im Speicher, können aber aufgrund des Nullterminierungszeichens nicht angezeigt werden.

4.2.2 C++ Strings

In C++ ist die Klasse *string* in der Headerdatei <string> definiert.

Deklaration:

```
#include <string>
using namespace std;
string str1; // leerer String
```

Beispiele für Definitionen:

```
string str2 = „Ich bin ein String“;
string str3(10, ‚+’);      // Die Variable str3 enthält als Wert 10 Pluszeichen
string str4(str2,4,7);    // Die Variable str4 enthält als Wert: bin ein
```

Strings können mit dem überladenen Operator + verkettet werden (Operator-Überladung siehe Teil 2 Kapitel 5):

```
string str5 = str2 + „ zum Testen“; // Die Variable str5 enthält als Wert: Ich bin ein String zum Testen
```

Beispiel: Weitere String-Funktionen

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

int main()
{
        string str1= "Mein Haus";          // ANMERKUNG: Zeichenketten beginnen wie Felder ab der Stelle 0
        string str2= str1.substr(5,4);     // Gibt Teilstring aus. Entnimmt ab dem 6ten Zeichen [da beginnend bei 0]
                                           // die 4 kommenden Zeichen im String (Ergebnis: Haus)
        string str3= str1.erase(1,5);      // Löscht die Zeichen 1-5 [da beginnend bei 0, also “ein H“] (Ergebnis: Maus)
        cout << str2 << endl;
        cout << str3 << endl;

        string strA="Klein";
        string strB="Schw";
        string strC= strA.replace(0,2,strB);  // Ersetzt das 0te und die 2 darauffolgende Zeichen
        cout << strC << endl <<endl;       // mit dem übergebenen String (Ergebnis: Schwein)

        string str4="abcd";
        cout << str4.length() << endl;     // Gibt die Länge des Strings aus (Ergebnis: 4)
        cout << str4.find("d") << endl;    // Gibt Index des gefundenen Buchstabens aus, falls Buchstabe nicht
                                           // vorhanden => Ausgabe einer höheren Zahl als Länge des Strings
                                           // (Ergebnis: 3)

        return 0;
}
```

4.3 Sortieren

Gleichartige Daten in größerem Umfang wird man meist sortiert ausgeben wollen. So wird Ihnen das Sortieren in Ihrem Programmiererleben immer wieder begegnen. Aus diesem Grunde verfügt die Standard C++-Bibliothek auch über ein Sortierverfahren. Dieses würden Sie vermutlich auch sinnvollerweise in der Praxis einsetzen. Dennoch ist es nicht verkehrt, als Übung für den Umgang mit Arrays einmal einen einfachen Sortieralgorithmus selbst zu programmieren.

Es gibt verschiedene Möglichkeiten, ein Feld zu sortieren. Wir verwenden den sogenannten Bubblesort. Dieser benötigt zwei ineinander verschachtelte Schleifen. Die äußere Schleife gibt die Anzahl der Vergleiche vor. Der Algorithmus vergleicht immer zwei nebeneinanderliegende Elemente und vertauscht die beiden, falls das rechte kleiner ist als das linke. Der Name kommt daher, dass die großen Werte wie Blasen aufsteigen und nach rechts wandern.
Da nach jedem Durchlauf der Liste das größte Element ganz rechts steht, muss man nur noch eine um ein Element kürzere Liste sortieren. Somit muss man nur genauso oft durch die Liste gehen, wie sie Elemente hat.

Beispiel: Bubblesort

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

int main()
{
        int zahlen[6] = {43,22,100,500,3,99};

        int i, j;                                       // Zählindizes
        int tmp;                                        // Hilfsvariable zum Vertauschen von Werten
        for (i = 0; i < 6 - 1; ++i){                    // durch das ganze Feld gehen
                for (j = 0; j < 6 - i - 1; ++j){        // am Ende beginnen das kleinste Element zu suchen
                if (zahlen[j] > zahlen [j + 1])         // prüfen ob das Element kleiner als der Vorgänger ist
                {       // Werte vertauschen
                        int tmp = zahlen [j];
                        zahlen [j] = zahlen [j + 1];
                        zahlen [j + 1] = tmp;
                }
                }
           }

        for(int i=0;i<6;i++)
                cout << zahlen[i] << endl;

        system("pause");
        return 0;
}
```

5 Paradigmen der Objekt-Orientierung (OO)

In Kapitel 5 lernen Sie die Merkmale und grundlegenden Konzepte der Objektorientierung kennen.

5.1 Überblick

5.1.1 Wie ist die OOP entstanden?

Ursprünglich dienten Computer ausschließlich dazu, numerische Aufgaben zu lösen. Ihre Aufgabe war es Rechenaufgaben zu lösen, deren Umfang und Komplexität die Grenzen menschlichen Rechenvermögens erreichte oder gar überschritt. Ein typisches Beispiel für diese Verwendung ist das Berechnen von Logarithmentafeln, eine Aufgabe, für die viele der ersten Computer konstruiert wurden. Diese wurden für viele Berechnungen im militärischen und ingenieurwissenschaftlichen Bereich benötigt, und die von Hand erstellten Tafeln waren zumeist sehr fehlerbehaftet. Für diese numerischen Zwecke war ein Programmierkonzept völlig ausreichend, das auf der einen Seite einen Satz von Daten vorsah (gegebenen und berechneten) und auf der anderen Seite Funktionen, um diese zu ändern. Hinzu kam, dass die Programme noch relativ klein waren und nur für eine bestimmte Nutzungszeit, den sogenannten Software-Lebenszyklus, entworfen wurden.

Mit steigenden Anforderungen an die Computer wuchs aber der Umfang der Programme sehr stark. Auch beschränkten sich die Anwendungen nicht mehr nur auf das Lösen numerischer Probleme, sondern der Computer vollzog einen Wandel zu einem universellen Arbeitsgerät, bis hin zu Multimedia, Künstlicher Intelligenz und computervermittelter Kommunikation.

Diese Entwicklung erforderte ein Umdenken auch bei dem Programmierkonzept. Die Komplexität der Software erforderte nicht nur eine größere Übersicht bei der Programmierung.
Es war auch nicht mehr praktikabel Software jedes Mal von Grund auf neu zu programmieren, um dann ein Produkt zu haben, das sehr starr angelegt war.
Es konnte fast nur noch ersetzt werden, wenn es nach einigen Jahren überholt war.
Es wurde nötig, Software in kleine Sinneinheiten aufzuteilen, die immer wieder verwendet werden können. Dabei stellte sich eine Aufteilung in Datensatz und Funktionalität, wie sie bislang üblich war, als sehr unhandlich heraus. Man strukturierte stattdessen die Programmteile in Funktionseinheiten, welche die für sie benötigten Daten gleich mit enthielten, womit das Konzept der OOP geboren war.

5.1.2 Wozu dient OOP?

In den vorherigen Abschnitten sind viele der Motive OOP zu nutzen schon angeklungen. Dieser Abschnitt soll sie noch einmal zusammenfassend aufführen, da gerade OOP-Neulinge sich oft nicht mit ihren Konzepten anfreunden können. Meist ist ihnen der Sinn und Zweck dieses Programmierkonzeptes nicht klar ist.
Die grundlegende Idee der OOP ist es, durch Bildung von Sinneinheiten (Objekten) einen besser strukturierten Programmcode zu erzeugen. Diese Sinneinheiten bestehen zum einen aus Daten (Elementen), zum anderen aus Funktionen (Methoden). Dadurch entsteht eine leicht definierbare Schnittstelle zu den Objekten, welche die Wiederverwendung erleichtert. Auch eine nachträgliche Änderung ist leicht durchführbar, solange sie die bestehende Schnittstelle nicht verändert (eine Erweiterung ist natürlich möglich).

Durch das Vererbungskonzept (Unterklassen erben alle Eigenschaften der Oberklasse) ist es zusätzlich möglich, bestehende Objekte leicht um neue Funktionen zu erweitern bzw. Variationen bestehender Objekte zu erzeugen.
Die OOP versucht, sich an der menschlichen Denkweise zu orientieren. Die grundlegende Idee der OOP lautet wie folgt:

- Die Welt besteht aus agierenden Objekten, welche in Klassen zusammengefasst werden können. Diese Objekte werden für die Programmierung typisiert. Bei der Typisierung entstehen die Klassen. Ein Objekt ist stets ein Exemplar einer Klasse.
- Beispiel: Die Daten zu einer Person *Name: Willi, Alter: 20* werden in einem Objekt gespeichert. Dieses Objekt ist vom Typ Person, welches die Klasse ist.

Objekte haben auch Eigenschaften und können miteinander kommunizieren. Es wird versucht, die Realität in einem gewissen Ausmaß wahrzunehmen, um einem Programmierer der OOP eine leichtere und alltagsbezogene Abstraktion in der Implementierung zu ermöglichen.

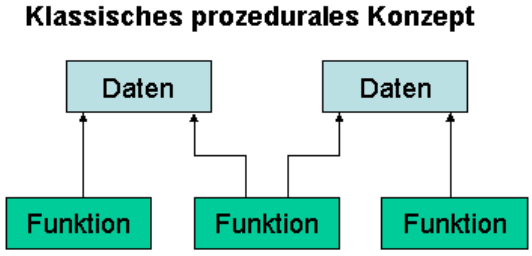
Die OOP hat aber auch Nachteile:

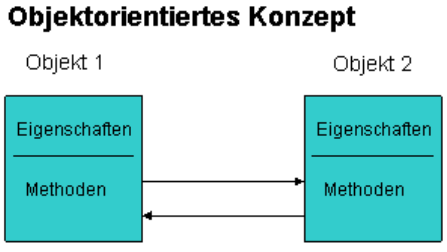
- Durch die Modellierung von Klassenhierarchien oder Vererbungsdiagrammen wird viel Zeit in Anspruch genommen, weil Planung in der OOP im Gegensatz zu prozeduralen Sprachen einen hohen Stellenwert besitzt.
- Zum anderen sind OOP-Anwendungen in den meisten Fällen auch rechen- und speicherintensiver und laufen deswegen oft langsamer ab, als dies etwa bei C-Anwendungen der Fall war.

Trotzdem sollte man wegen der immer weiter fortschreitenden Entwicklung, OOP-Sprachen wie Java oder C++ beherrschen, um den Anschluss an eine immer stärker wachsende, objektorientierte Softwarebranche nicht zu verpassen.

5.2 Die wichtigsten Grundlagen

Was ist denn OOP nun genau?





5.2.1 Objektorientierte Denkweise

Für die traditionelle, prozedurale Programmierung ist es charakteristisch, dass Daten und Funktionen, die diese Daten verarbeiten, keine Einheit bilden. Daraus resultiert eine größere Fehleranfälligkeit, z.B. durch fehlerhafte Zugriffe auf die Daten oder Verwendung nicht initialisierter Daten. Hinzu kommt ein hoher Wartungsaufwand, etwa wenn die Daten an neue Anforderungen angepasst werden müssen.

Dagegen bilden die Objekte in der OOP eine Einheit aus Daten (Eigenschaften) und Funktionen (Methoden, Fähigkeiten).

Für die Software-Qualität ergeben sich daraus entscheidende Vorteile:

- Höhere Zuverlässigkeit
- Geringerer Wartungsaufwand
- Bessere Wiederverwendbarkeit

Merkmale der OOP:
Eine objektorientierte Programmiersprache ist dadurch gekennzeichnet, dass sie Sprachelemente zur Unterstützung der folgenden OOP-Paradigmen besitzt:

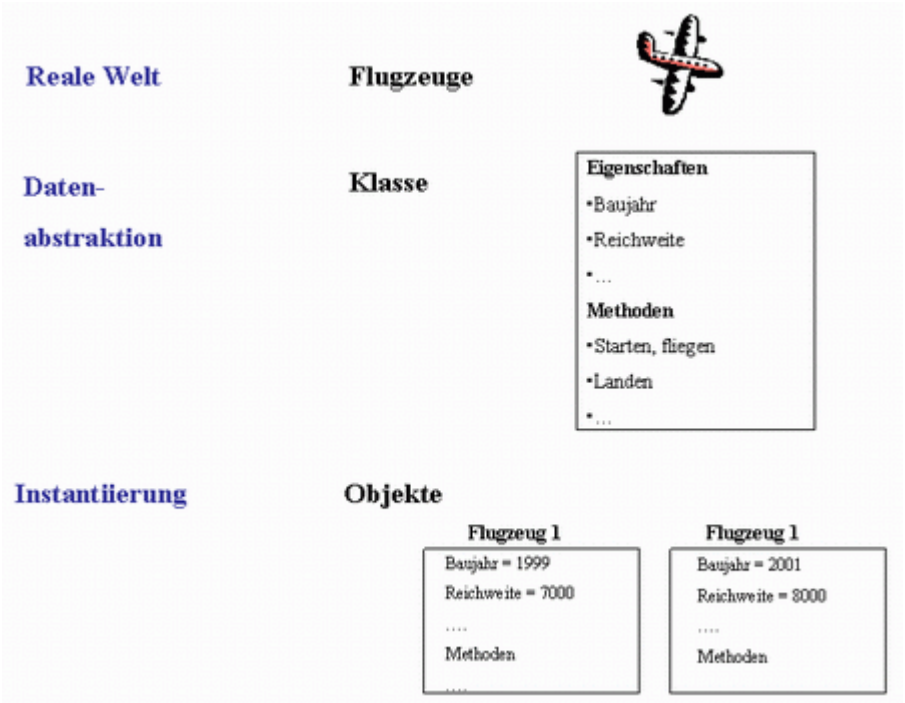
- **Datenabstraktion**
  Es können abstrakte Datentypen (Klassen) definiert werden, welche die Eigenschaften und Fähigkeiten von Objekten beschreiben.

- **Datenkapselung**
  Elemente eines Objekts können vor unkontrolliertem Zugriff von außen geschützt werden. Für die Kommunikation mit der Außenwelt, d. h. mit anderen Objekten, besitzt jedes Objekt eine »öffentliche Schnittstelle«.

- **Vererbung**
  Neue Objekte lassen sich aus schon vorhandenen Objekten ableiten. Sie »erben« die vorhandenen Eigenschaften und Fähigkeiten, die ergänzt und verändert werden können.

5.2.2 Datenabstraktion

Eine Möglichkeit, um komplexe Sachverhalte zu handhaben, ist die Abstraktion. Eigenschaften und Vorgänge werden auf das Wesentliche reduziert und mit einem Oberbegriff versehen.



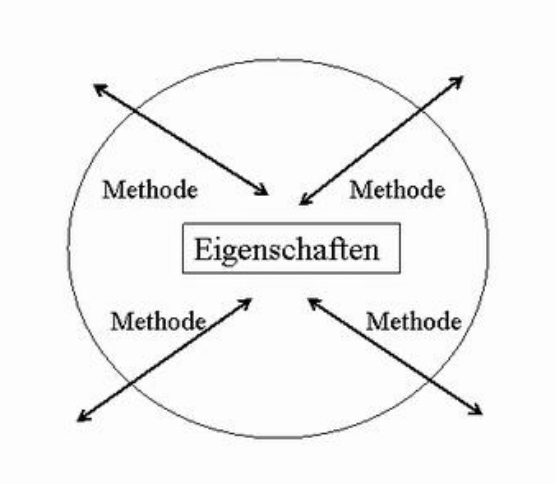
Ein Flugzeug ist ein Beispiel für eine in verschiedener Hinsicht durchgeführte Abstraktion:

- Ein Flugzeug ist die Zusammenfassung verschiedener Einzelteile wie Motor, Tragflächen, Räder usw. Darüber hinaus besitzt jedes Flugzeug Eigenschaften wie Reichweite, Höchstgeschwindigkeit und Transportkapazität. Auch der aktuelle Status, wie z. B. »Das Flugzeug ist geparkt«, gehört zu den Eigenschaften.
- Die Fähigkeiten eines Flugzeugs bestehen natürlich in erster Linie aus Starten, Fliegen und Landen. Außerdem muss ein Flugzeug getankt, beladen, entladen und gewartet werden.
- Ein Flugzeug ist auch der Oberbegriff für verschiedene Flugzeugtypen wie Segelflugzeug, Motorflugzeug, Düsenjäger, Transportflugzeug, Passagierflugzeug usw.

Sofern die Details eines Flugzeugs bzw. die Unterschiede zwischen verschiedenen Flugzeugtypen nicht interessieren, wird einfach der abstrakte Begriff »Flugzeug« verwendet. Damit können die für ein Flugzeug typischen Aktionen in den Vordergrund rücken: Ein Flugzeug kann gebaut, gewartet oder demontiert werden. Ein Flugzeug startet, fliegt, z.B. von München nach Köln, oder landet.

5.2.3 Kapselung

Wie in der Einführung schon dargestellt, ist es einer der zentralen Aspekte der OOP, Daten mit den zugehörigen Funktionen zusammenzuführen. Diesen Prozess bezeichnet man als ***Kapselung*****,** die daraus resultierende Datenstruktur nennt sich ***Klasse***. Eine Variable oder Konstante, die diese Datenstruktur nutzt, nennt sich eine ***Instanz*** der Klasse. Eine Klasse kann zusätzlich zu den Daten auch weitreichende Funktionalitäten übernehmen. Dadurch wird sie zu einem ***Abstrakten Datentyp***, der zu den Inhalten (den Daten) auch die Möglichkeiten spezifiziert, mit ihnen umzugehen (die Funktionen). Die Daten einer Klasse werden dabei als *Elemente*, und die Funktionen als *Elementfunktionen* oder *Methoden* bezeichnet.

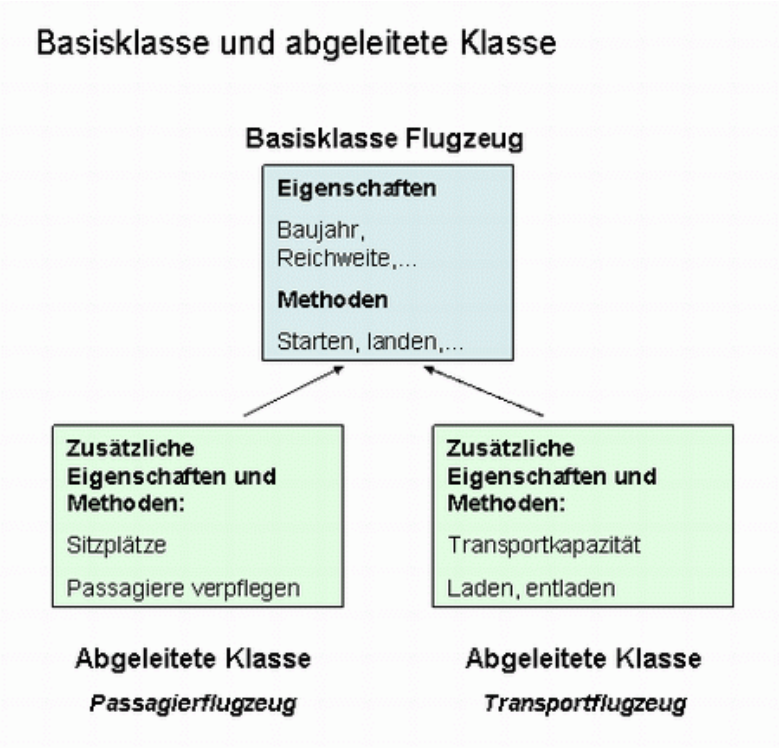


Objekte haben ein Innen und ein Außen. Innen heißt dabei, dass es Elemente und evtl. auch Methoden gibt, die der Nutzer (d.h. der aufrufende Programmteil) dieses Objektes nicht sieht, während das Außen die Menge der Elemente und Methoden ist, die der Nutzer auch sehen kann. Objektorientierte Sprachen haben in der Regel die Möglichkeit, eine solche Unterscheidung zu treffen. In C++ werden Elemente und Methoden des inneren Bereichs ***privat*** *(private)* genannt, die des äußeren Bereichs ***öffentlich*** *(public)*. Die Nutzung der privaten Datenstrukturen ist dann nur noch über öffentliche Methoden möglich. Bei einem konsequent objektorientierten Programmierstil sollten alle Elemente privat sein und der Zugriff auf die Elemente nur über Methoden erfolgen (sogenannte *Zugriffsmethoden*). Damit ist u.a. gewährleistet, dass man später die Datenstrukturen der Objekte austauschen kann, ohne die Aufrufe der entsprechenden Zugriffsmethoden zu ändern - und damit auch keine Programme ändern muss, die dieses Objekt nutzen. Außerdem ist es möglich, zusätzliche Effekte in die Zugriffsmethoden mit einzubauen, so z.B. ein Flag, das anzeigt, ob eine Änderung der Daten erfolgte. Wenn ein direkter Zugriff auf die Elemente der Klasse möglich wäre, wäre dieses Flag unzuverlässig.

Aus dem Konzept des Innen und Außen eines Objektes folgt, dass ein Objekt eine Außenseite hat. Diese Außenseite wird gebildet durch die Zugriffsmöglichkeiten auf Methoden und Elemente des Objektes. Sie werden vor allem von den Namen der öffentlichen Elemente und Methoden sowie der Parameter dieser Methoden bestimmt. Die Menge der Zugriffsmöglichkeiten wird als ***Schnittstelle*** *(Interface)* bezeichnet.
Der Aufruf einer Methode eines Objektes wird häufig als ***Nachricht*** *(Message)* bezeichnet. Dahinter steckt die Vorstellung, dass ein Programmteil durch eine Nachricht ein Objekt dazu bringt, etwas zu tun. Das kann dann durchaus auch nur das Setzen eines Elementes sein. Wesentlich an diesem Konzept der Programmierung ist aber, dass niemals ein externer Programmteil das Objekt verändert, sondern immer nur eine Nachricht an das Objekt schickt, dies selbst zu tun. Damit kann ein Objekt immer die Kontrolle über sein Innen behalten.

5.2.4 Vererbung

Ein weiterer wichtiger Aspekt der OOP ist die Möglichkeit der ***Vererbung***. Um nicht jedes Mal eine komplette neue Klasse programmieren zu müssen, kann man in den objektorientierten Sprachen angeben, dass eine Klasse alle Elemente und Methoden einer anderen Klasse 'ererben' soll, d.h. die neue Klasse enthält von vorneherein alle Daten und Funktionen der alten Klasse. Diese können dann um weitere ergänzt oder bei Bedarf auch überschrieben werden. Damit wird eine Beziehung wie z.B. 'ein Apfel ist eine Frucht' realisiert, durch zusätzliche Elemente und Methoden wird die neue Klasse genauer spezifiziert. Die alte Klasse wird dabei als ***Basisklasse*** bezeichnet, die neue als ***abgeleitete* Klasse** und die Struktur, die durch die Vererbung entsteht, als ***Klassenhierarchie*.**



5.3 Vorteile der objektorientierten Vorgehensweise

Seit Beginn der 90er Jahre hat die Bedeutung der objektorientierten Analyse- und Entwurfsmethoden ständig zugenommen und ist heute die Basis für die meisten Softwareentwicklungen – vor allem, wenn es sich um umfangreiche oder komplexe Systeme handelt. Bei einer objektorientierten Softwareentwicklung werden die Ergebnisse der Phasen Analyse, Entwurf und Implementierung objektorientiert erstellt. Ein großer Vorteil der objektorientierten Konzepte ist, dass sie in allen Phasen der Entwicklung eingesetzt werden können. Die Implementierung erfolgt schließlich auch in einer objektorientierten Programmiersprache wie z.B. JAVA oder C++. Zusätzlich kann die Benutzeroberfläche der Software objektorientiert gestaltet sein, und es können objektorientierte Datenbanken eingesetzt werden. Damit entsteht eine Softwareentwicklung "aus einem Guss".

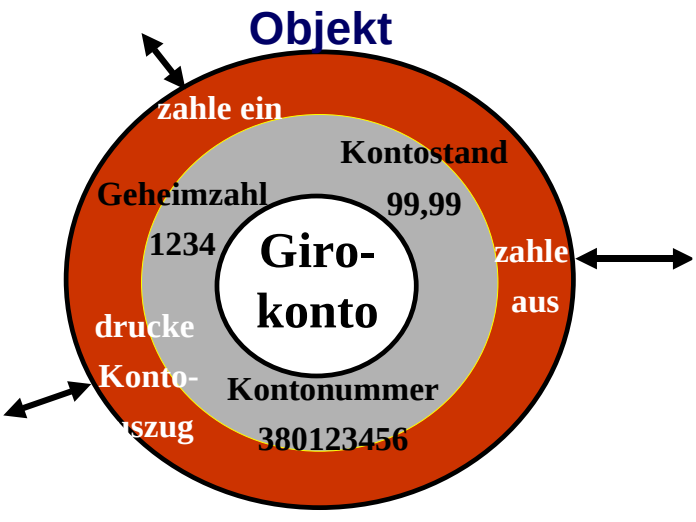
Letztlich bedeutet Softwareentwicklung immer auch Abstraktion. Eine konkrete Fragestellung der realen Welt muss in ein abstraktes Modell des Rechners überführt werden, das auf unterster Ebene mit Bits und Bytes des Computers realisiert wird. Diese Aufgabe, ein abstraktes Modell der realen Fragestellung zu erstellen, wird wesentlich vereinfacht, wenn man die Objekte der realen Welt identifiziert und direkt mit Hilfe von objektorientierten Programmiersprachen realisiert.
Wenn bereits die Analyse der Aufgabenstellung objektorientiert erfolgt und in einem objektorientierten Analysemodell beschrieben wird, kann daraus relativ einfach ein objektorientierter Entwurf gewonnen werden. Und, falls dieser Entwurf sorgfältig entwickelt wurde, kann daraus sehr einfach – ja sogar teilweise automatisch – die nötige Implementierung in C++ oder JAVA gewonnen werden.

Die bessere Durchgängigkeit wird bei den objektorientierten Techniken also dadurch erreicht, dass in allen Phasen dieselben Konzepte verwendet werden. Letztlich liegt der größte Vorteil der objektorientierten Softwareentwicklung darin, dass Anforderungen an die Software einfacher, d.h. mit weniger Aufwand und Fehlern, realisiert werden können.

5.4 Objekte

5.4.1 Objektbegriff

Beginnen wir mit der Wiederholung des Grundlagenbegriffs **“Objekt”**.
Jedes Objekt hat zur Identifikation einen eindeutigen **Namen.** Die Daten eines Objekts werden als **Attribute bzw. als Attributwerte** des Objektes bezeichnet. Die Prozesse werden als dynamische Eigenschaften, **Verhalten oder Methoden** des Objektes bezeichnet.
Attribute und Methoden bilden innerhalb des Objektes eine Einheit.



In unserem Beispiel hat das Objekt den **Namen** “Girokonto”
mit den **Attributen bzw. Attributwerten**

- “Kontostand=99,99”
- “Kontonummer=380123456”
- “Geheimzahl=1234”

und den **Methoden** “zahle ein”, “zahle aus” und “drucke Kontoauszug”.
Die Daten des Objekts “Girokonto” können nur unter Anwendung der “eigenen” Methoden gelesen und geändert werden.
Das bedeutet:

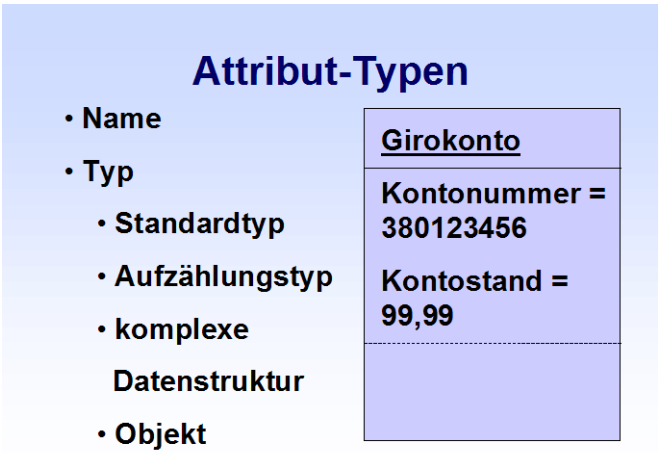
- Ein Objekt kapselt Attribute und Methoden

Man kann auch sagen:
- Ein Objekt kapselt die eigenen Daten und Operatoren

Dieses Grundprinzip nennt man auch **Geheimnisprinzip.**
Über die Methoden und deren Operatoren kann ein Objekt mit anderen Objekten kommunizieren.

5.4.2 Attributtypen

Sehen wir uns die Attribute eines Objektes näher an. **Attribute** beschreiben die **Daten** eines Objektes und werden durch ihren **Namen** und ihren **Typ** definiert.



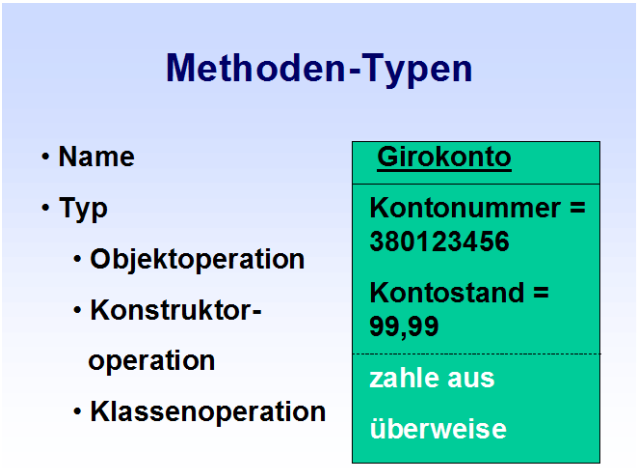
Für die Attribute stehen die **Standardtypen**, wie "integer", "float", "char" in der Programmiersprache C++ zur Verfügung. So könnte in unserem Beispiel die Kontonummer von Typ "integer" und der Kontostand vom Typ "float" definiert werden.

Attribute können jedoch auch als Aufzählungstypen, komplexe Datenstruktur oder selbst als Objekt definiert werden.

5.4.3 Methodentypen

Methoden bezeichnen Funktionen, die von einem Objekt ausgeführt werden. Sie werden auch als **Operationen** bezeichnet.

Jede Operation kann auf alle Attribute eines Objektes zugreifen. Methoden sind immer an Objekte gebunden! Deshalb lässt sich eine Methode nur im Zusammenhang mit einem Objekt aufrufen.



Es stehen
- **Objektoperationen** zur Verfügung.

Diese werden stets auf ein bereits existierendes Objekt angewendet. In unserem Beispiel sind die Operationen "zahle aus" und "überweise" definiert.

Ein weiterer Operationstyp sind die
- **Konstruktor-Operationen.**

Sie erzeugen ein neues Objekt und führen Initialisierungen und Datenerfassungen durch. Die Operation "eröffne Cashkonto" ist beispielsweise eine Konstruktor-Operation.

Im Zusammenhang mit dem Basiskonzept "Klassen" werden wir noch die
- **Klassenoperationen** kennenlernen.

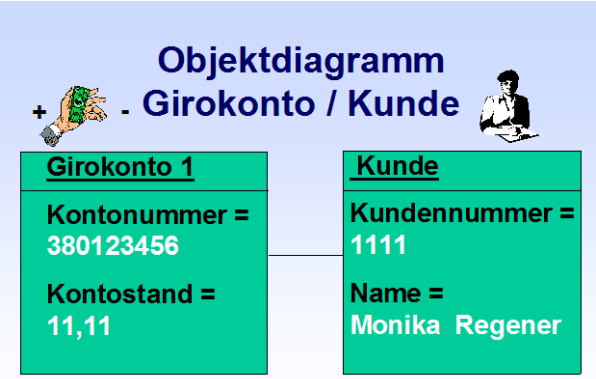
Kann man nachträglich einem Objekt neue Attribute und neue Methoden hinzufügen?

Ja, genau das ist die Stärke der Objektorientierung!

Das Geheimnisprinzip stellt sicher, dass neue Attribute bzw. Attributwerte von einem Objekt aufgenommen werden können, ohne dass die “Umwelt” davon erfährt. Sie können beispielweise dem Objekt “Girokonto” das Attribut “PIN für Online-Banking” hinzufügen. Dieser Vorgang bleibt anderen Objekten verborgen. Auch die Einführung neuer Methoden ist jederzeit möglich.

5.4.4 Objektdiagramme

Objekte und deren Beziehungen untereinander werden in **Objektdiagrammen** dargestellt.



In unserem Beispiel stellen wir in einem Objektdiagramm die Beziehung zwischen dem Objekt “Girokonto” und dem Objekt “Kunde” dar.

Die Objekte werden mit ihren Objektattributen und Attributwerten dargestellt und in Verbindung gebracht.

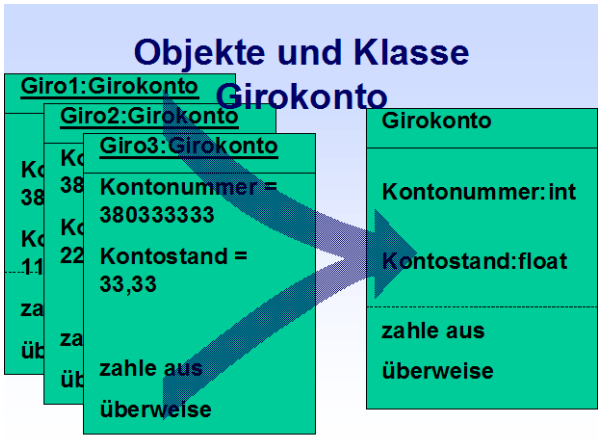
Die Angabe der Operationen ist optional.

5.5 Klassen

Wir kommen nun zu einem weiteren Basiskonzept, das auf dem Objektbegriff aufsetzt, den **Klassen.**

5.5.1 Klassenbegriff

Eine **Klasse** definiert für eine Sammlung von gleichartigen Objekten die zugehörigen Attribute, Methoden und Beziehungen. Der Klassenname sollte im gesamten Betrachtungsraum eindeutig sein. Die Klasse besitzt die Fähigkeit neue Objekte zu erzeugen. Jedes erzeugte Objekt gehört genau einer Klasse an.



Sehen wir uns auch hier ein Beispiel aus der “Welt der Bankkonten” an.

Die Girokonten 1, 2 und 3 können zur Klasse “Girokonto” zusammengefasst werden. Die Methoden “zahle aus” und “überweise” sind für die gesamte Klasse “Girokonten” gültig und anwendbar. Die Klassen-Attribute, wie Kontonummer und Kontostand gelten für alle Objekte der Klasse. Die Attributwerte sind für jedes Objekt dieser Klasse unterschiedlich.

Welche Arten von Methoden gibt es, um innerhalb einer Klasse oder zwischen verschiedenen Klassen operieren zu können?

Wir stellen die verschiedenen Arten jeweils anhand eines kurzen Beispiels dar:

- Operationen mit lesendem Zugriff auf Attribute derselben Klasse
  Beispiel: drucke Kontoauszug

- Operationen mit schreibendem Zugriff auf Attribute derselben Klasse
  Beispiel: zahle ein

- Operationen zur Durchführung von Berechnungen
  Beispiel: ermittle mittleren Kontostand eines ausgewählten Girokontos

- Operationen zum Erzeugen und Löschen von Objekten
  Beispiel: eröffne neues Girokonto

- Operationen, die Objekte einer Klasse nach bestimmten Kriterien selektieren
  Beispiel: sortiere Girokonten nach Kontostand

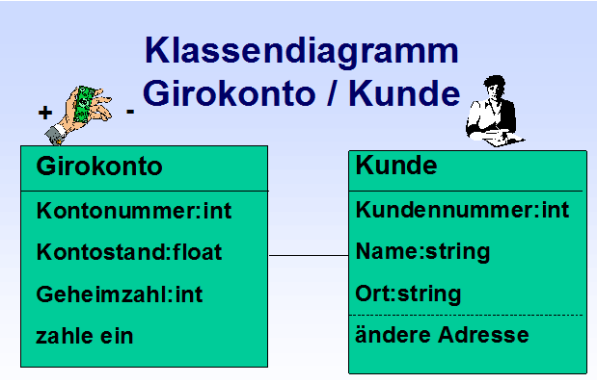
- Operationen, die Operationen anderer Klassen aktivieren
  Beispiel: erstelle Quellensteuernachweis

- Operationen zum Herstellen von Verbindungen zwischen Objekten
  Beispiel: verbinde mit Online-Zugang

Der **Name einer Klassen-Operation** muss innerhalb der Klasse eindeutig sein.
Außerhalb der Klasse wird die Operation mit “KLASSE.OPERATION” bezeichnet.

5.5.2 Klassendiagramme

Klassen werden in **Klassendiagrammen** dargestellt.



In unserem Beispiel stellen wir in einem Klassendiagramm die Beziehung zwischen der Klasse “Girokonto” und der Klasse “Kunde” dar.

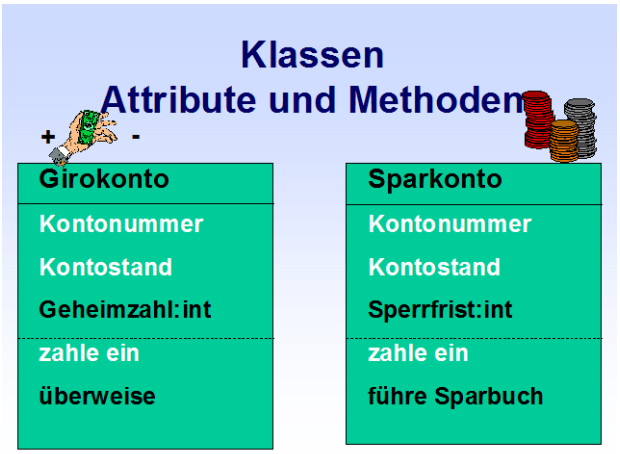
Die Klassen werden mit ihren Klassenattributen und den Klassenoperationen dargestellt.
Im Unterschied zum Objektdiagramm werden hier die Attribute mit ihrem Datentyp spezifiziert, jedoch ohne Werte. Die Initialisierung mit konkreten Werten erfolgt ja erst bei der Erzeugung von Objekten einer Klasse.

5.6 Vererbung

Wir werden nun ein weiteres Basiskonzept besprechen, welches sich aus der Abstraktion von Klassen ergibt, das **Prinzip der Vererbung.**

5.6.1 Vererbungsprinzip

Die **Vererbung** beschreibt eine Beziehung zwischen einer allgemeinen Klasse (auch Basisklasse genannt) und einer spezialisierten Klasse.
Wir sehen uns das zunächst wieder an einem praktischen Beispiel an.

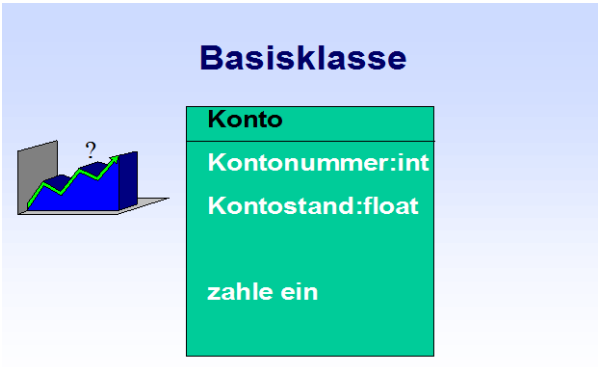


Girokonten und Sparkonten haben gemeinsame, aber auch unterschiedliche Eigenschaften (Attribute) und Methoden.

Beide Kontoklassen werden durch eine Kontonummer identifiziert und führen den aktuellen Kontostand, aber nur für das Girokonto erhalten sie eine Geheimzahl und ein Sparkonto ist vielleicht mit einer Sperrfrist belegt.
Auf beide Kontoklassen können Sie Geld bar oder unbar einzahlen, aber nur für das Sparkonto wird ein Sparbuch geführt.

Es scheint also eine Basisklasse zu geben, welche die gemeinsamen Attribute und Methoden beschreibt und spezialisierte Klassen, welche noch zusätzliche Attribute und Methoden besitzen.

Wie könnte nun in unserem Beispiel eine Basisklasse aussehen?

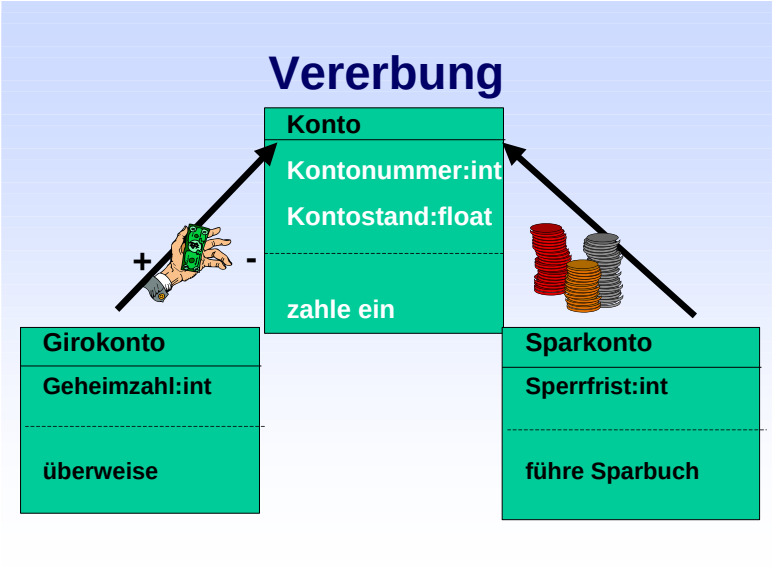


Richtig !
Eine Basisklasse namens Konto hat eben die Attribute „Kontonummer“ und „Kontostand“ und die Methode „zahle ein“.

Die Basisklasse vererbt ihre Attribute und Methoden an alle spezialisierten Klassen und diese wiederum werden durch eigene, lokale Attribute und Methoden ergänzt.

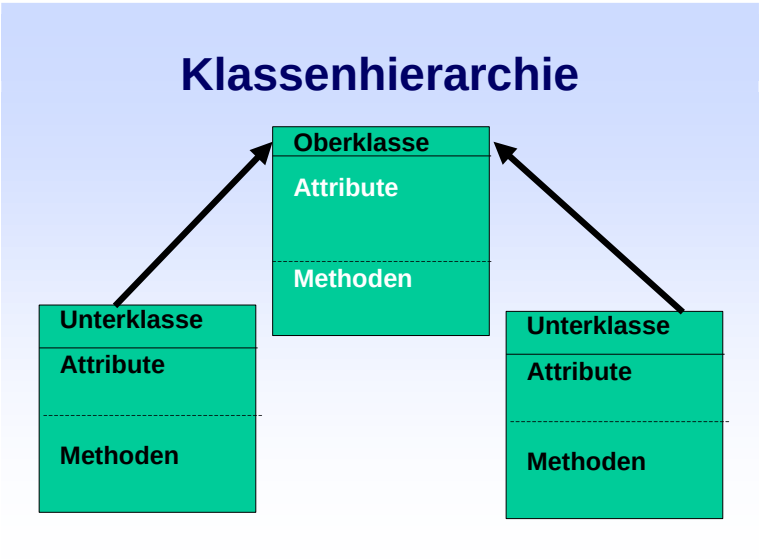
5.6.2 Vererbungshierarchie

Die Darstellung der Vererbung wird auch als **Klassenhierarchie** oder **Vererbungsstruktur** bezeichnet.

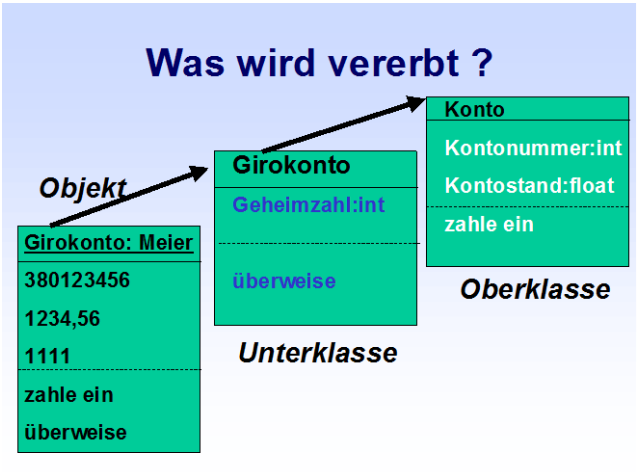


Zusammenfassend kann man sagen:
Die allgemeine Klasse wird auch als **Oberklasse,** die spezialisierte Klasse als **Unterklasse** bezeichnet. Wir sprechen auch von **Generalisierung**, im Falle der Oberklasse und von **Spezialisierung** im Falle der Unterklasse. Eine Klassenhierarchie kann natürlich über mehrere Hierarchiestufen aufgebaut werden. Jedes Objekt einer Unterklasse **ist auch ein** Objekt der Oberklasse.

Beachte:
Die Pfeile zeigen immer in Richtung der Oberklasse



Am Beispiel der Bankkonten lässt sich sehr gut darstellen was an wen vererbt wird!



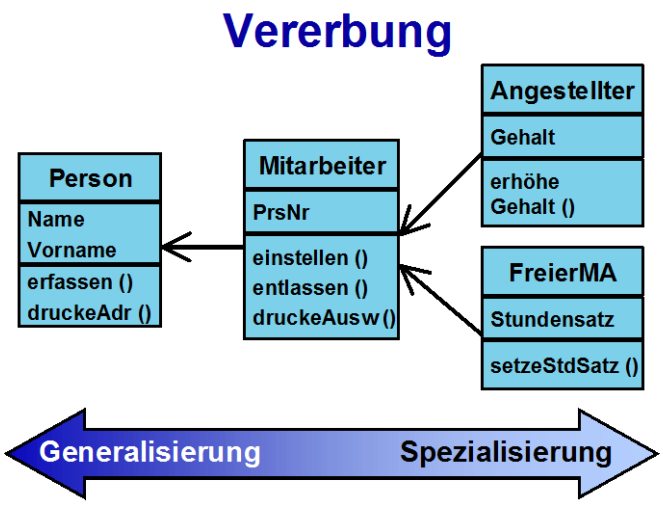
Einem konkreten Objekt der Klasse Girokonto, also einem konkreten Girokonto, werden die Attribute und Methoden der Unterklasse Girokonto und der Oberklasse Konto vererbt.

5.7 Abschließendes Beispiel

Bei einem Projektverwaltungssystem, das bei den Mitarbeitern zwischen Angestellten und freien Mitarbeitern unterscheiden soll, kann das Konzept der Vererbung vorteilhaft angewandt werden. Eine Beziehung zwischen einer allgemeinen Klasse, wie unserem Mitarbeiter und einer spezialisierten Klasse, wie dem Angestellten oder freien Mitarbeiter, wird als Vererbung bezeichnet. Mit Hilfe der Vererbung können wir komplette Klassenhierarchien, ausgehend von einer allgemeinen Basisklasse, bis zu immer weiter spezialisierten Klassen aufbauen.

Beispiel:

- Person
- Mitarbeiter
- Angestellter FreierMitarbeiter
- TarifAngestellter AußerTarifAngestellter



In der UML wird die Verbindung von generellen Klassen und spezialisierten Klassen durch einen Pfeil dargestellt, der in Richtung der Generalisierung zeigt. Eine Generalisierung liegt immer dann vor, wenn sich eine Beziehung zwischen Klassen als "ist ein" beschreiben lässt.
Dieses "ist ein" bedeutet auch, dass eine speziellere Klasse stets wie ihre Basisklasse verwendet werden kann. In unserem Beispiel der Projektverwaltung heißt das, dass auch ein freier Mitarbeiter in einem Projekt mitarbeiten kann, da er auch ein Mitarbeiter ist, der eine Assoziation zum Projekt hat.

Damit haben wir die wichtigsten Grundkonzepte des objektorientierten Softwareentwurfs kennengelernt.

6 Das Klassenkonzept in C++

In Kapitel 6 lernen Sie, wie man das im vorherigen Kapitel 5 in erklärte Klassenkonzept in C++ umsetzt.

6.1 Was ist eine Klasse?

Eine *Klasse* ist eine Gruppe von Datenelementen und Funktionen, die zusammenarbeiten, um eine bestimmte Programmieraufgabe zu verrichten. Man sagt auch, die Klasse *kapselt* die Aufgabe.

Klassen haben die folgenden Merkmale:
- Attribute
- Methoden
- Datenkapselung für Zugriffsbeschränkungen auf die Klassenattribute
- Konstruktoren
- Destruktoren

Eine Klasse ist sozusagen ein abstrakter Bauplan für ein konkretes Objekt (=Instanz).
Jede Instanz einer Klasse ist ein eigenständiges Objekt. Jede Instanz hat ihre eigenen Datenelemente und die Objekte sind unabhängig voneinander. Sie sind alle vom selben Typ, doch im Speicher bilden sie gesonderte Instanzen.

Sehen wir uns noch mal die Basisklasse „Konto“ an, welche wir im letzten Kapitel beispielhaft besprochen haben.

Der **Name** der Klasse ist „Konto“.
Zwei **Attribute** der Klasse wurden mit „Kontonummer“ und „Kontostand“ angegeben, jeweils vom Datentyp *integer*.
Eine **Methode** „zahle ein“ wurde angegeben.

6.2 Attribute einer Klasse in C++

6.2.1 Deklaration, Definition und Zugriff

Eine Klasse muss zuerst **deklariert** werden**.** Nach der Deklaration gilt sie als Datentyp.

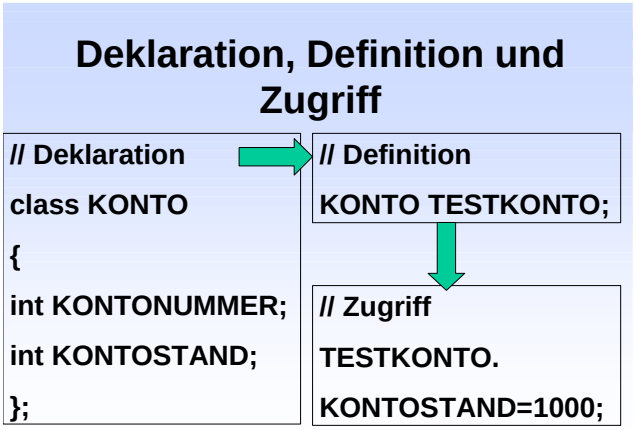
```
class KONTO              // Deklaration der Klasse KONTO
{
public:
    int KONTOSTAND;
};
```

Wir können sie dann wie jeden anderen Datentyp verwenden und im Hauptprogramm Variablen (Instanzen) davon bilden.

```
KONTO TESTKONTO;          // Erzeugen einer Instanz der Klasse KONTO
```

Der Zugriff auf eine Instanz erfolgt durch Angabe ihres Namens zusammen mit dem Namen des Elements getrennt durch den **Punktoperator** (für den Zugriff auf Elemente eines Objektes).

```
TESTKONTO.KONTOSTAND = 1000; // Setzt den KONTOSTAND von TESTKONTO auf 1000
```



Gewöhnlich findet man die Deklaration einer Klasse in einer Header-Datei. In einfachen Fällen können sowohl Deklaration als auch Definition der Klasse in ein und derselben Quelltextdatei stehen.

Normalerweise erzeugen Sie eine Quelltextdatei, deren Name weitgehend dem Namen der Klasse entspricht und die auf .CPP endet. Da Windows 95 und Windows NT lange Dateinamen unterstützen, können Sie, wenn Sie wollen, Ihre Datei auch genauso benennen wie Ihre Klasse. Die Header-Datei für die Klasse benennen Sie am besten nach der Quelltextdatei, nur dass sie hier die Endung .H verwenden.

Beispiel:

```
class KONTO                    // Deklaration in Headerdatei: Abspeichern unter KONTO.H
{
  int KONTONUMMER;
  int KONTOSTAND;
.......
};
```

```
#include "KONTO.H"       // Einbinden der eigendefinierten Headerdatei
.....
                         // Abspeichern unter KONTO.CPP
```

Kann man auch mehrere Konten definieren?

Ja, das ist wieder eine Stärke der Objektorientierung.

Falls Sie neben einem TESTKONTO noch ein GIROKONTO, ein SPARKONTO und ein CASHKONTO für Ihr Programm benötigen, müssen Sie diese nur noch definieren.
Im Beispiel werden 3 Instanzen der Klasse KONTO erzeugt.

Beispiel:

```
KONTO GIROKONTO;
KONTO SPARKONTO;
KONTO CASHKONTO;
```

Diese drei Konten werden auf Basis der Klassendeklaration KONTO definiert und stehen dann schon zur Bearbeitung zur Verfügung.

6.2.2 Geheimnisprinzip in C++

Ein zentrales Paradigma der Objektorientierung ist das **Geheimnisprinzip.**

Der entscheidende Punkt ist, dass die Attributwerte eines Objektes nur über definierte Methoden geändert werden können. Die Attributwerte sind deshalb von der Umwelt abgekapselt.

Sehen wir uns nun die Anwendung des Geheimnisprinzips in der Programmiersprache C++ an.

Ich möchte nochmals auf unser letztes Beispiel zurückkommen.

In der Deklaration der Klasse KONTO wurden die Attribute KONTONUMMER und KONTOSTAND angegeben.

6.2.2.1 private und public

Ohne weitere Angaben wurden diese Attribute im default als „**private“** deklariert.

Damit erreichen wir ein Optimum an Datenkapselung.

Man kann nun „von außen“ nicht mehr auf die Attribute und Attributwerte dieser Klasse zugreifen.

Aber Vorsicht!

```
class KONTO                          // Definition der Klasse Konto
{
int KONTONUMMER;
};

int main()
{
KONTO Girokonto;                     // Girokonto instanziiert

Girokonto.KONTONUMMER=555666;        // Compilerfehler!!!
}
```

Der direkte Zugriff auf die Attributwerte einer Klasse, wie in diesem Beispiel angegeben, führt beim Kompilieren zu einer Fehlermeldung, wie „KONTO::KONTONUMMER is not accessible“ !

Wie kann man nun auf die Daten eines Klassenobjektes, beispielsweise den KONTOSTAND, zugreifen?

Es gibt zwei Möglichkeiten.

Zum einen über Methoden, welche die Daten ändern können.
Wir werden das etwas später kennenlernen.

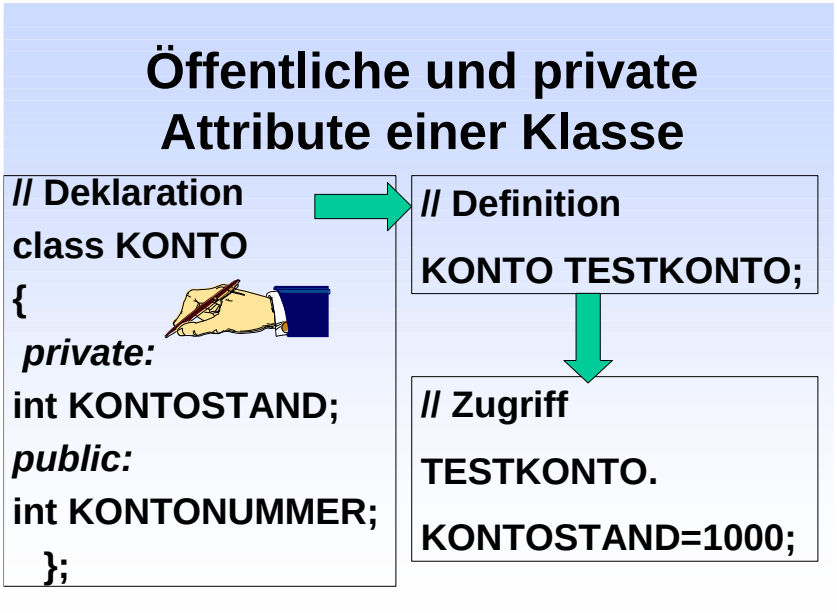
Zum anderen können wir die Datenkapselung lockern, indem wir die entsprechenden Attribute **„public“** deklarieren.

Sehen wir uns dazu nochmals unser Beispiel an.

-
```
class KONTO                          // Definitition der Klasse Konto
{
public: int KONTONUMMER;
};

int main()
{
KONTO Girokonto;                     // Girokonto instanziiert

Girokonto.KONTONUMMER=555666;        // Zugriff auf Instanz
}
```

Der Compiler wird nun keine Fehlermeldung mehr erzeugen.

Sie sollten jedoch die **„public“-Qualifizierung** sehr überlegt einsetzen, sonst wird das Geheimnisprinzip unterlaufen. Die Standardeinstellung für die Deklaration der Klassenattribute sollte deshalb **„private“** sein.



Die Zugriffsbeschränkungen regeln, wie eine Klasse benutzt werden kann. Als C++-Entwickler/-in sind Sie sowohl Schöpfer/-in, als auch Benutzer/-in der Klasse. In einem Team allerdings, werden Sie Ihre Klasse auch den anderen Programmierer/-innen zur Verfügung stellen. Um verstehen zu können, welche Bedeutung die Zugriffsbeschränkungen haben, müssen Sie zuerst lernen, wie Klassen verwendet werden. In jeder Klasse gibt es einen öffentlichen Abschnitt (public), auf den die Außenwelt zugreifen kann. Der private Abschnitt (private) ist für die innere Gestaltung der Klasse. Eine gute Klasse sollte die Dinge, die der Benutzer nicht zu wissen braucht, nicht in die Öffentlichkeit tragen.

Durch die Datenabstraktion erreicht man, dass der Benutzer nur so viel über eine Klasse erfährt, wie zu Ihrer »Bedienung« nötig ist. Sie steigen zum Beispiel in Ihr Auto ein und drehen den Zündschlüssel, um den Motor zu starten. Möchten Sie in allen Einzelheiten wissen, was dabei in Ihrem Auto vorgeht? Natürlich nicht!

Sie möchten nur wissen, wie man es handhabt. In dieser Analogie bilden das Lenkrad, die Pedale, der Tacho und so weiter die Schnittstelle zwischen Auto und Fahrer. Der Fahrer weiß, welche Operationen er wann und wie durchführen muss, um von Ort A nach Ort B zu gelangen. Umgekehrt sind der Motor, das Fahrgestell und die Elektrik des Autos verkleidet und versteckt, so dass Sie kaum etwas damit zu tun haben. Dies sind Details, über die Sie nichts zu wissen brauchen; also werden sie vor Ihnen verborgen – in unserer Terminologie wären sie also privat.

Stellen Sie sich vor, was es bedeuten würde, wenn sie immer alles im Auge behalten müssten, was das Auto macht: Kriegt der Vergaser genug Benzin? Ist das Differential ausreichend geschmiert? Liefert die Lichtmaschine die richtige Spannung? Öffnen die Einlassventile sauber? Wer braucht das?
Auf dieselbe Weise behält eine Klasse ihre interne Arbeitsweise für sich, damit der Benutzer sich nicht darum sorgen muss, was unter der Haube passiert. Die Interna sind privat, und die Benutzerschnittstelle ist öffentlich.

6.2.2.2 protected

Die Zugriffsspezifikation protected *(geschützt*) ist etwas schwieriger zu erklären. Geschützte Klassenelemente können, genau wie private Elemente, nicht vom Benutzer verändert werden. Abgeleitete Klassen können jedoch auf sie zugreifen. *protected* wird im Kapitel 9 beim Thema „Vererbung“ ausführlicher behandelt.

Kehren wir noch einmal zu unserem Auto-Beispiel zurück. Angenommen Sie wollten Ihr Auto verlängern, um es in eine stattliche Limousine zu verwandeln. Dazu müssten Sie etwas von der dem Auto zugrundeliegenden Struktur verstehen. Sie müssten zumindest wissen, wie Sie die Antriebswelle und den Rahmen des Fahrzeugs modifizieren. Sie müssten Ihre Hände schmutzig machen und kämen als Designer mit den Teilen des Fahrzeugs in Berührung, die Sie vorher nicht interessiert hätten (die geschützten Teile). Dabei bleibt Ihnen die eigentliche Funktionsweise des Motors immer noch verschlossen (private), da Sie nicht wissen müssen, wie der Motor funktioniert, um den Autorahmen zu verlängern. Ähnlich ist es mit den »öffentlichen« Teilen des Autos, die Sie zumeist nicht ändern, denen Sie aber einige neue »öffentliche« Elemente hinzufügen können, wie zum Beispiel die Bedienungselemente für eine Sprechanlage.

Damit sind wir jetzt jedoch etwas vom Thema abgewichen und haben einen kleinen Einblick in das Problemfeld Vererbung erhalten. Ein Thema, das ich erst später ausführlicher behandeln werde.

C++ besitzt drei Schlüsselwörter für die Zugriffsspezifikation. Sie lauten public, private und protected (was so viel wie *öffentlic*h, *privat* und *geschützt* bedeutet).

- *public*: Elemente der Klasse sind öffentlich, d.h. sind auch von außerhalb der Klasse zu greifbar
- *private*: Der Zugriff auf private-Elemente ist nur für andere Elemente der Klasse zugelassen.
- *protected*: Abgeleitete Klassen (siehe Kapitel 9) erhalten Zugriff auf die Elemente der Klasse.

Ob, beziehungsweise wie der Zugriff auf ein Klassenelement beschränkt wird, legen Sie bei der Deklaration fest. Die Klasse selbst wird mit dem Schlüsselwort class deklariert. Eine Klassendeklaration gleicht der Deklaration einer Struktur mit zusätzlichen Zugriffsspezifizierern.

6.3 Methoden einer Klasse in C++

Kommen wir nun zur Abbildung von **Methoden** in C++.

Methoden sind ja, das haben wir im letzten Kapitel schon ausführlich besprochen, die zu einer Klasse gehörigen Funktionen.

Methoden sind in der Lage auf **private Attribute** zuzugreifen.

Mit Methoden wird demnach das Geheimnisprinzip sichergestellt.

Kommen wir nochmals auf das Beispiel einer **Basisklasse** zurück.

Neben dem Namen und den Attributen der Klasse werden auch die Methoden der Klasse angegeben.
In unserem Beispiel wurde nur die Methode „zahle ein“ angegeben.

Der Gültigkeitsbereich von Methoden ist die Klasse; außerhalb dieser existieren sie nicht. Sie können nur innerhalb der Klasse oder durch eine Instanz der Klasse aufgerufen werden. Mithilfe der Methoden können Sie auf alle Attribute der Klasse zugreifen, ohne Rücksicht auf deren jeweilige Zugriffsbeschränkung. Methoden können ebenfalls im public-, protected- und private-Modus deklariert werden.

Überlegen Sie sich aber genau, in welchen dieser Abschnitte Sie welche Methoden packen.
Die public-Methoden bilden die Schnittstelle zwischen Klasse und Benutzer. Über Sie kann der Benutzer auf die Klasse zugreifen und mit ihr arbeiten.

Nehmen wir an, Sie haben eine Klasse, die Klangdateien abspielt und aufzeichnet. Unter den public-Methoden der Klasse sollten sich dann Funktionen wie
- Oeffnen(), Abspielen(), Aufnehmen(), Speichern(), Vorspulen()

befinden.

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

class Klang
{
   private:
          int timer;
          void timerSetzen(int t) { timer = t; };

   public:
          void Oeffnen() { timerSetzen(0); };
          void Abspielen() { };
          void Aufnehmen() { };
          void Speichern() { };
          void Vorspulen() { };
};

int main()
{
          Klang k;
          k.Oeffnen(); // möglich
          // k.timerSetzen(0); // nicht möglich

          system("pause");
          return 0;
}
```

Die private-Methoden werden für die interne Arbeit der Klasse verwendet. Sie sind nicht dafür vorgesehen, vom Benutzer aufgerufen zu werden; schließlich sind sie ja privat, um sie vor der Außenwelt zu verbergen.

Ein Beispiel:

Häufig müssen bei der Bildung der Klasseninstanzen umfangreiche, viele Quelltextzeilen umfassende Arbeiten erledigt werden (darunter fallen zum Beispiel die Initialisierungen der Variablen und das Reservieren von Speicherplatz). Um diese Methode übersichtlich zu halten, könnte man diese Anweisungen in eine Init()-Funktion packen, die dann aufgerufen werden kann. Diese Funktion sollen niemals direkt vom Benutzer aufgerufen werden. Wahrscheinlich wäre es geradezu fatal, wenn ein Benutzer dies zur falschen Zeit täte. Eine solche Funktion würde also als private deklariert werden, um die Unversehrtheit sowohl der Klasse als auch des Benutzers zu gewährleisten.

Auch protected-Methoden können nicht vom Benutzer aufgerufen werden. Allerdings können abgeleitete Klassen auf sie zugreifen.

6.3.1 Deklaration einer Methode

In C++ muss eine Methode zunächst **deklariert** und **definiert** werden, bevor sie zur Anwendung kommen kann.

Zunächst sehen wir uns die Deklaration einer Methode an.

Neben den Attributen werden für eine Klasse auch die Methoden deklariert.

Auch Methoden können wie Attribute **public, private** oder **protected** sein**.**

Beispiel:

```
int KONTOSTAND_LESEN();
```

Deklaration einer Methode

```
class KONTO
{ private: // Deklaration eines Attributs
      int KONTOSTAND;
  public: // Deklaration einer öffentl. Methode
      int KONTOSTAND_LESEN();
};
```

6.3.2 Definition einer Methode

Im nächsten Schritt muss eine deklarierte Methode definiert werden.
Es wird dabei festgelegt, wie die Methode im Detail arbeitet.
In unserem Beispiel haben wir eine Methode definiert, welche nur einen Lesevorgang vornimmt.

Wie bei der Definition einer Funktion wird die Methode im Methodenkopf und Methodenkörper festgelegt.

Methoden können innerhalb der Klasse oder außerhalb der Klasse definiert werden.

6.3.2.1 Definition innerhalb der Klasse

```
class KONTO
{
        private:
                int KONTOSTAND;    // Deklaration eines Attributs

        public:                            // Definition der Methode KONTOSTAND_LESEN
                int KONTOSTAND_LESEN()
                {
                        return(KONTOSTAND);
                }
};
```

6.3.2.2 Definition außerhalb der Klasse

Bei umfangreicheren Methoden wird die Definition innerhalb der Klasse zu unübersichtlich und sollte deshalb außerhalb der Klassendeklaration erfolgen.
Bitte beachten Sie, dass hier der **Klassenoperator „::“** verwendet werden muss.
Er wird auch Gültigkeitsbereichoperator genannt und zeigt an für welche Klasse diese Methode definiert wird.

Beispiel:

Methodenkopf: `int KONTO::KONTOSTAND_LESEN()`
Methodenkörper: `{ return (KONTOSTAND); }`

Definition einer Methode

```
int KONTO::KONTOSTAND_LESEN( )
{
// Die Methode KONTOSTAND_LESEN liefert
// den Wert des Attributs KONTOSTAND
// der Klasse KONTO zurück
return(KONTOSTAND);
}
```

6.3.2.3 friend

In C++ gibt es die Möglichkeit durch *private* oder *protected* geschützte Attribute oder Methoden für andere Klassen zugänglich zu machen, indem man sie als „Freund“ (*friend*) deklariert.

Beispiel:

```
class TESTKLASSE
{
friend class FREUNDKLASSE;        // friend-Deklaration

private:                  //Deklaration von Attributen und Methoden von TESTKLASSE
…
};
```

Damit dürfen alle Objekte der Klasse FREUNDKLASSE auf die privaten Attribute und Methoden von TESTKLASSE zugreifen.

Merke: „Freunde“ einer Klasse haben die gleichen Zugriffsrechte wie die Methoden der Klasse selbst.

6.3.3 Anwendung einer Methode

Der Aufruf bzw. die Anwendung einer Methode erfolgt durch Angabe

- **des Objektnamens**
- **des Punktoperators (für den Zugriff auf Elemente eines Objektes)**
- **der Methodennamen und**
- **der Methodenparameter**

Beispiel:

TESTKONTO.KONTOSTAND_LESEN();

Anwendung einer Methode

```
// Aufruf der Methode KONTOSTAND_LESEN
// der Klasse KONTO ohne
// Eingabeparameter
TESTKONTO.KONTOSTAND_LESEN();
```

7 Beispielanwendung: KONTOVERWALTUNG

7.1 Anforderungen

Wir werden in einem Beispiel zur KONTOVERWALTUNG die Umsetzung der OO-Konzepte demonstrieren.
Was soll denn unsere KONTOVERWALTUNG können?
Unser Programm ist recht einfach aufgebaut.
Wir wollen den KONTOSTAND AENDERN und den KONTOSTAND LESEN können.
Das soll natürlich auch mehrmals möglich sein.

Bildschirmausgabe

KONTOVERWALTUNG
1 = KONTOSTAND AENDERN
2 = KONTOSTAND LESEN
0 = ENDE DER AKTION

IHRE WAHL:_

7.2 Analyse

Zunächst müssen wir uns darüber Gedanken machen, welche
- Klassen
- Attribute und
- Methoden

wir für die Lösung unseres Problems benötigen.

Eine erste **Analyse** wird ergeben, dass wir eine Klasse KONTO mit dem Attribut
- KONTOSTAND

und den Methoden
- WERT_ÄNDERUNG und
- KONTOSTAND_LESEN

entwerfen müssen.

7.3 Deklaration einer Klasse

Die Klasse KONTO wird also wie folgt deklariert:

```
class KONTO
{
   private:
          int KONTOSTAND;

   public:
          bool WERT_AENDERUNG(int);
          int KONTOSTAND_LESEN();
};
```

In der Klasse sind die beiden Methoden lediglich deklariert. Es muss nun außerhalb der Klasse noch genau festgelegt werden, wozu sie dienen (Definition). Es wäre auch möglich, die Methoden innerhalb der Klasse zu deklarieren und gleichzeitig zu definieren.
Die Methode Wert_Aenderung erhält als Eingabeparameter einen Wert, der von der Tastatur eingegeben werden soll. Dann erfolgt in der if-Anweisung die Überprüfung, ob der Wert plausibel ist (zwischen 0 und 100000). Falls die Eingabe korrekt ist, wird der Wert des Attributs KONTOSTAND auf den Eingabewert gesetzt und die Methode gibt 1 (true) als Returnwert zurück.
Andernfalls erfolgt keine Wertänderung und es wird 0 (false) zurückgegeben.

```
bool KONTO::WERT_AENDERUNG(int EINGABE)   // Definition der METHODE "WERT_AENDERUNG"
{
          if((EINGABE>=0)&&(EINGABE<=100000))  // Die Eingabe muss zwischen 0 und 100000 liegen
          {
            KONTOSTAND=EINGABE;               // KONTOSTAND wird geändert
            return(1);                        // Änderung erfolgt
          }
          return(0);                          // Änderung hat nicht geklappt (z.B. bei Eingabe eines negativen
                                              // Werts)

int KONTO::KONTOSTAND_LESEN()                 // Definition der METHODE "KONTOSTAND_LESEN"
          { return(KONTOSTAND); }
```

7.4 Hauptprogramm

Im Hauptprogramm wird zunächst eine Instanz der Klasse KONTO erzeugt und die Methode WERT_Aenderung mit 0 initialisiert.

```
int main()                                    // Hauptprogramm
{
   KONTO TESTKONTO;                           // Definition TESTKONTO
```

Dann wird die Methode WERT_AENDERUNG mit 0 initialisiert. Die Methode hat als Eingabeparameter eine Variable. Diese muss auch ebenfalls mit 0 initialisiert werden.

```
    int WERT=0;                               // Initialisierung einer Variablen

   TESTKONTO.WERT_AENDERUNG(WERT);            // Initialisierung der Methode WERT_AENDERUNG mit 0
```

In einer Do-While-Schleife wird das Benutzermenü angezeigt. Hier wird die Hilfsvariablen ALTERNATIVE benötigt, die zunächst mit 0 initialisiert wird
Das Menü wird solange angezeigt, wie der Benutzer 1 oder 2 eingibt. Gibt er 0 ein, wird die Schleife und das Programm beendet.

```
   int ALTERNATIVE=0;                                  // Initialisierung einer Variablen

   cout << "K O N T O V E R W A L T U N G" << endl;    //  Benutzermenü
        do                                             //  ! DO-WHILE-Schleife !
        {
        cout << "1 = KONTOSTAND AENDERN\n";            // Eingabe Benutzerwunsch
        cout << "2 = KONTOSTAND LESEN\n";
        cout << "0 = Ende der Aktion" << endl << endl;
        cout << "Ihre Wahl:";
        cin >> ALTERNATIVE;                            // Tastatureingabe

                                                       // Switch Anweisung hier einbauen

        } while(ALTERNATIVE);                          // Solange ALTERNATIVE gewünscht!
  system("pause");
}                                                      // Programmende!
```

Innerhalb einer Switch-Anweisung wird entsprechend der Benutzereingabe die Methode WERT_AENDERUNG oder KONTOSTAND_LESEN aufgerufen

```
        switch(ALTERNATIVE)                    // Folgefunktionen
        {
        case 1:                                // Eingabe eines neuen WERTES
                cout << "Neuer WERT:";
                cin >> WERT;                           // Wert wird von der Tastatur eingelesen
                if(TESTKONTO.WERT_AENDERUNG(WERT))  // Methodenaufruf und Abfrage: 1 (true) oder 0 (false)
                        cout << "WERT geaendert." << endl << endl;
                else
                        cout << "WERT nicht geaendert." << endl << endl;
                break;

        case 2:                                // Abfrage des aktuellen WERTES
                cout << "Aktueller WERT:";
                cout << TESTKONTO.KONTOSTAND_LESEN() << endl << endl;
                break;
        }
```

7.5 Vollständiges Programm

Das vollständige Programm sieht nun wie folgt aus:

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

class KONTO                                    // Deklaration der KLASSE "KONTO"
{
   private:
          int KONTOSTAND;

   public:
          int WERT_AENDERUNG(int);
          int KONTOSTAND_LESEN();
};

int KONTO::WERT_AENDERUNG(int EINGABE)         // Definition der METHODE "WERT_AENDERUNG"
{
          if((EINGABE>=0)&&(EINGABE<=100000))  // Die Eingabe muss zwischen 0 und 100000 liegen
          {
          KONTOSTAND=EINGABE;                  // KONTOSTAND wird geändert
            return(1);                         // Änderung erfolgt
          }
          return(0);                           // Änderung hat nicht geklappt (z.B. bei Eingabe
}                                              // eines negativen Werts

int KONTO::KONTOSTAND_LESEN()                  // Definition der METHODE "KONTOSTAND_LESEN"
          { return(KONTOSTAND); }

int main()                                     // Hauptprogramm
{
   KONTO TESTKONTO;                            // Definition TESTKONTO
   int ALTERNATIVE=0, WERT=0;                  // Initialisierung von Variablen

   TESTKONTO.WERT_AENDERUNG(0);                // Initialisierung der Methode WERT_AENDERUNG mit 0

   cout << "K O N T O V E R W A L T U N G " << endl;      //  Benutzermenü
          do                                               //  ! DO-WHILE-Schleife !
          {
          cout << "1 = KONTOSTAND AENDERN" << endl;       // Eingabe Benutzerwunsch
          cout << "2 = KONTOSTAND LESEN" << endl;
          cout << "0 = Ende der Aktion" << endl << endl;
          cout << "Ihre Wahl:";
          cin >> ALTERNATIVE;                              // Tastatureingabe

          switch(ALTERNATIVE)                              // Folgefunktionen
          {
          case 1:                                          // Eingabe eines neuen WERTES
          cout << "Neuer WERT:";
          cin >> WERT;                                     // Wert wird von der Tastatur eingelesen
          if(TESTKONTO.WERT_AENDERUNG(WERT))               // Aufruf der Methode
          cout << "WERT geaendert." << endl << endl;
          else
          cout << "WERT nicht geaendert." << endl << endl;
          break;

          case 2:                                          // Abfrage des aktuellen WERTES
          cout << "Aktueller WERT:";
          cout << TESTKONTO.KONTOSTAND_LESEN() << endl << endl;
          break;
          }

          } while(ALTERNATIVE);                            // Solange ALTERNATIVE gewünscht!
  system("pause");
}                                                          // Programmende!
```

8 Spezielle Klasseneigenschaften und –methoden

In Kapitel 8 werden spezielle Methoden erklärt, die für das Erzeugen und Löschen von Objekten benötigt werden.
Am Ende des Kapitels werden Sie außerdem wissen, was man unter „Überladen von Methoden“ und „Klassenvariablen“ versteht.

8.1 Konstruktoren

Der *Konstruktor* ist eine besondere Methode. Er kann bei der Erzeugung einer Instanz, d.h. einem konkreten Exemplar einer Klasse aufgerufen werden. Der Konstruktor hat den Vorteil, dass er die Initialisierung der Attribute, allgemeine Hinweise und Speicher-Ressourcen beim Erzeugen einer Instanz bereitstellen kann. Wird kein Konstruktor angelegt, so wird ein Default-Konstruktor vom Compiler bereitgestellt.

Konstruktor

Deklaration:
```
class KONTO
{ ... KONTO(void); };
```
Definition:
```
KONTO::KONTO(void)
{cout << "Objekt erzeugt !\n\n";}
```
Anwendung:
```
KONTO K1;
KONTO K2;
```

Für eine Klasse kann es nur einen Konstruktor geben. Bevor der Konstruktor in Aktion treten kann, muss er deklariert und definiert werden. **Der Konstruktor besitzt denselben Namen wie die zugehörige Klasse!**
Die Implementierung des Konstruktors kann wie bei anderen Methoden innerhalb oder außerhalb der Klassendeklaration erfolgen.
Der Konstruktor kann überladen werden, er hat jedoch keinen Rückgabewert.

Ein Beispiel hierzu:
- *Sie wollen beispielsweise eine Meldung erhalten, wenn ein neues Konto eröffnet worden ist!*

8.2 Destruktor

Der *Destruktor* wird automatisch vom Compiler aufgerufen, wenn ein konkretes Exemplar einer Klasse gelöscht wird, z.B. bei einem lokalen Objekt am Ende des Blocks, in dem es angelegt wurde. Er wird meistens für die Freigabe von Speicher-Ressourcen eingesetzt.

Destruktor

Deklaration:
```
class KONTO
{ ... ~KONTO(); };
```
Definition:
```
KONTO::~KONTO()
{cout << "Objekt zerstoert !\n\n";}
```
Anwendung:

automatisch beim Löschen

Der vom Compiler eingesetzte Default-Destruktor kann durch einen eigenen Destruktor ersetzt werden, der z.B. eine Meldung über das Löschen des Objektes in eine Protokolldatei schreiben könnte.

Der Destruktor heißt wie die Klasse selbst, hat aber eine Tilde (~) vorangestellt.

Er hat keinen Rückgabewert und keine Eingabeparameter. Nicht einmal „void“ ist erlaubt.

8.3 Elementinitialisierungsliste

Diese Technik wird meist bei Konstruktoren eingesetzt und dient zur Initialisierung einzelner Attribute beim Erzeugen einer Klasseninstanz.

Sie können schon bei der Definition des Konstruktors im Methodenkopf eine Elementinitialisierungsliste angeben.

Bei jeder Generierung einer Instanz werden diese Vorgabewerte automatisch eingestellt.

Auch hierzu ein Beispiel aus unserer KONTOVERWALTUNG:
Sie wollen eventuell bei der Einrichtung eines neuen Kontos ein Startkapital von 100 € berücksichtigen.

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

class KONTO                         // Deklaration der KLASSE "KONTO"
{
private:
      int KONTOSTAND;

public:
      KONTO();                      // Konstruktor
      ~KONTO();                     // Destruktor
};

  KONTO::KONTO():KONTOSTAND(100)          // Konstruktor-Methode für KONTO
  {
   cout << "Neues KONTO erzeugt!\n\n";
  }


  KONTO::~KONTO()                         // Destruktor-Methode für KONTO
  {
   cout << " KONTO aufgeloest!\n\n";
  }

int main()          // Hauptprogramm
{
   KONTO TESTKONTO;                       // Definition TESTKONTO (Aufruf des Konstruktors)
   system("pause");
}                                         // Programmende (impliziter Aufruf des Destruktors)
```

Die Implementierung des Konstruktors kann auch innerhalb der Klassendeklaration erfolgen.

```
class KONTO
{
        KONTO() : KONTOSTAND(100) {}
};
```

Die Elementinitialisierungsliste kann mehrere Parameter enthalten. Diese werden durch ein Komma getrennt:

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

class KONTO                                  // Deklaration der KLASSE "KONTO"
{
private:
int KONTOSTAND;
int KONTONR;

public:
       KONTO(int KST,int KNR                 // Konstruktor mit 2 Eingabeparameter
       ~KONTO();                             // Destruktor
};


   KONTO::KONTO(int KST, int KNR):KONTOSTAND(KST),KONTONR(KNR)     // Konstruktor-Methode
   {
 cout << "Neues KONTO erzeugt !\n\n";
   }


   KONTO::~KONTO()                                                 // Destruktor-Methode
   {
    cout << " KONTO aufgeloest !\n\n";
   }

int main() //Hauptprogramm
{

    int ALTERNATIVE=0, WERT=0;                 // Initialisierung von Variablen
     KONTO TESTKONTO(150,444555);              // Definition und Initialisierung TESTKONTO

   system("pause");
}
```

8.4 Überladen von Funktionen/Methoden

Wir verstehen darunter die Definition mehrerer **gleichnamiger** Methoden (= Funktionen), die sich in ihrer Signatur unterscheiden. Der Compiler sucht dabei anhand der Anzahl und Typen der Parameter, welche Methode, Funktion oder Konstruktor er aufrufen muss.

Überladen von Funktionen

Deklaration:
```
float WERT(float);
float WERT(float,float);
```

Definition:
```
float KONTO::WERT(float EINGABE){...};
float KONTO::WERT(float EINGABE, float FAKTOR){...};
```

Anwendung:
```
TESTKONTO.WERT(BETRAG);
TESTKONTO.WERT(BETRAG,FAKTOR);
```

Auch hierzu ein Beispiel:
Vielleicht wollen Sie die KONTOVERWALTUNG mit zwei Methoden ausstatten.
In einer Methode wollen Sie die WERT_ÄNDERUNG ohne Umrechnungs-Faktor
in der anderen Methode die WERT_ÄNDERUNG mit Umrechnungs-Faktor berücksichtigen.

```
#include<cstdlib>
#include <iostream>

using namespace std;

class KONTO
{
   private:
          float KONTOSTAND;
    public:
          float WERT_AENDERUNG(float);
          float WERT_AENDERUNG(float,float);                    //Überladen der Funktion WERT_AENDERN
          float KONTOSTAND_LESEN(void);};


float KONTO::WERT_AENDERUNG(float EINGABE)
{
          if((EINGABE>=0)&&(EINGABE<=100000))
          {
          KONTOSTAND=KONTOSTAND + EINGABE;

          return(1);
          }
          return(0);
}

float KONTO::WERT_AENDERUNG(float EINGABE, float Faktor)      // Definition mit 2 Eingabeparameter
{
          if((EINGABE>=0)&&(EINGABE<=100000))
          {
          KONTOSTAND=KONTOSTAND + (EINGABE * Faktor);         // Berechnung mit Faktor

          return(1);
          }
          return(0);
}
```

```
int main()
{
   KONTO TESTKONTO;
   TESTKONTO.WERT_AENDERUNG(0);
   int ALTERNATIVE=0;
   float WERT=0, Faktor=0;

   cout << "K O N T O V E R W A L T U N G  \n";

          do
          {
          cout << "1 = KONTOSTAND AENDERN ohne Faktor\n";
          cout << "2 = KONTOSTAND AENDERN mit Faktor\n";
          cout << "3 = KONTOSTAND LESEN\n";
          cout << "0 = Ende der Aktion\n\n";

          cout << "Ihre Wahl:";

          cin >> ALTERNATIVE;

          switch(ALTERNATIVE)
          {
          case 1:
            cout << "Neuer WERT:";
            cin >> WERT;
                  if(TESTKONTO.WERT_AENDERUNG(WERT))
                  cout << "WERT geaendert.\n\n";
                  else
                  cout << "WERT nicht geaendert.\n\n";
            break;

          case 2:                                                    //Berechnung mit Faktor
            cout << "Neuer WERT :";
            cin >> WERT;
            cout << "Faktor :";
            cin >>Faktor;
                  if(TESTKONTO.WERT_AENDERUNG(WERT,Faktor))
                  cout << "WERT geaendert.\n\n";
                  else
                  cout << "WERT nicht geaendert.\n\n";
            break;

          case 3:
            cout << "Aktueller WERT:";
                   cout << TESTKONTO.KONTOSTAND_LESEN() << "\n\n";
            break;
          }

          } while(ALTERNATIVE);
   system("pause");
}
```

8.5 Static

8.5.1 Statische Variablen

Statische Variablen werden beim Verlassen ihres Bezugsrahmens nicht gelöscht, sondern behalten ihren Wert bei.

Beispiel: Statische Variable

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

void count();                          // Prototyp der Funktion count

int main()
{
        count();
        count();
        count();

   system("pause");
}

void count()
{
   static int a=1;                     // statische Variablen-Definition

   cout << a << endl;
   a++;
}
```

Aufgabe: Was gibt das Programm auf dem Bildschirm aus?

Lösung:
1
2
3

Wichtig: Statische Variablen müssen bei ihrer Definition initialisiert werden.

8.5.2 Klassenvariablen

Statische Variablen finden auch in Klassen Verwendung.
Objekte einer Klasse sollten nicht nur ihre eigenen Daten verwalten, sondern auch Daten innerhalb derselben Klasse gemeinsam benutzen können. Solche Daten repräsentieren gemeinsame Eigenschaften aller Objekte einer Klasse. Sie werden in **Klassenvariablen** gespeichert. Das sind Variablen, die für alle Objekte einer Klasse nur einmal im Speicher vorhanden sind.

Einsatzmöglichkeiten für Klassenvariablen sind beispielsweise:

- Statusinformationen, die alle Objekte einer Klasse betreffen
  (wie etwa die Anzahl der Objekte einer Klasse oder bestimmte von Objekten erreichte Extremwerte)
- Kennzahlen, die zu einem Zeitpunkt für alle Objekte den gleichen Wert haben (wie etwa Zinsraten; Rabattsätze oder Umrechnungskurse)
- Zwischenspeicher für temporäre Daten (wie etwa Pipes, Task-Listen oder Priority Queues)

Deklaration von Klassenvariablen

In C++ erfolgt die Deklaration einer Klassenvariablen innerhalb der Klasse mit dem Schlüsselwort static. Deshalb werden sie auch **statische Datenelemente** genannt.

Beispiel:

```
static int anzahl;
```

Die Variable *anzahl* wird als statisches Element der Klasse *Kunde* deklariert. Sie speichert die Kundenzahl, d. h. die Anzahl der bereits existierenden Objekte vom Typ Kunde.

Definition von Klassenvariablen

Statische Datenelemente einer Klasse belegen Speicherplatz, auch wenn noch kein Objekt der Klasse existiert. Sie müssen daher wie die Elementfunktionen (Methoden) außerhalb der Klasse in einer Quelldatei definiert und initialisiert werden. Dabei wird der Klassenbezug durch den Bereichsoperator : : hergestellt.

Beispiel:

```
int Kunde : : anzahl = 0;        // Noch kein Objekt vorhanden
```

Die Definition wird in derselben Quelldatei vorgenommen, die auch die Definition der Methoden der Klasse enthält. Dabei wird static nicht mehr angegeben.
Jeder Konstruktor der Klasse *Kunde* inkrementiert die Variable *anzahl*. Im Gegenzug dekrementiert der Destruktor die Variable *anzahl*, da ja nach dem Aufruf des Destruktors ein Objekt weniger existiert.

Hinweis:
Ein konstantes statisches Datenelement, das einen ganzzahligen elementaren Datentyp besitzt, kann auch direkt bei der Deklaration in der Klasse initialisiert werden.

Beispiel:

```
#include <iostream>
#include <cstdlib>

using namespace std;

class Schaf
 {
    private:
    static int zaehler;         // Deklaration einer Klassenvariablen

    public:
    Schaf(){ zaehler++; };  // Definition Konstruktor; im Methodenkörper wird die Klassenvariable um 1 erhöht

    void zeige(){ cout << "Zeige Anzahl der Schafe: "<< zaehler << endl; } // Definition der Methode zeige
 };

int Schaf::zaehler=0;         // Initialisierung der Klassenvariablen außerhalb der Klasse

int main()
{
   Schaf Schaf1;              // Instanz der Klasse Schaf erzeugt (Konstruktoraufruf)
   Schaf Schaf2;

   Schaf2.zeige();            // Aufruf der Methode zeige für der 2. Instanz

   system("pause");
}
```

Programmausgabe:

Zeige Anzahl der Schafe: 2

9 Vererbung

In Kapitel 9 lernen Sie das Konzept der Vererbung für Klassen.

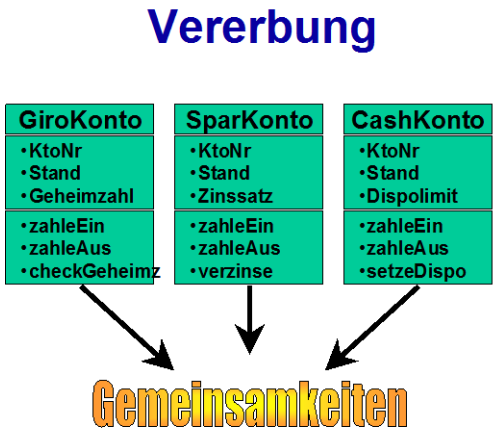
Wir kennen bereits den Begriff der Vererbung in der objektorientierten Softwareentwicklung. Mit Hilfe der Vererbung können wir einmal erstellte Programmteile wiederverwenden, indem wir die Elemente einer Klasse an andere Klassen weiterreichen. Damit muss nicht in jeder Klasse “das Rad neu erfunden werden”. Wenn wir allgemeingültige Konzepte in unseren Objekten erkennen, sollten wir diese auch nur einmalig in sogenannten Basisklassen realisieren.

Wir werden in diesem Kapitel die C++ Mechanismen zur Vererbung kennen lernen und beispielhaft einsetzen. Wir werden folgende Frage beantworten: Wie vererbt man die allgemeinen Eigenschaften eines Kontos an spezialisierte Konten?

9.1 Motivation

Aus Gründen einer wirtschaftlich effizienten Programmentwicklung und höherer Softwarequalität fordert die Softwareindustrie von ihren Programmen oder Programmteilen: **Wiederverwendung, Möglichkeiten zur Anpassung und Erweiterung.**

Deshalb ermöglicht die OO Softwareentwicklung, sogenannte **Klassenhierarchien** zu erstellen. Diese Hierarchie entsteht dadurch, dass neue Klassen gebildet werden, indem auf schon vorhandene, allgemeinere Klassendeklarationen mit Hilfe der **Vererbung** oder **Ableitung** zurückgegriffen wird.

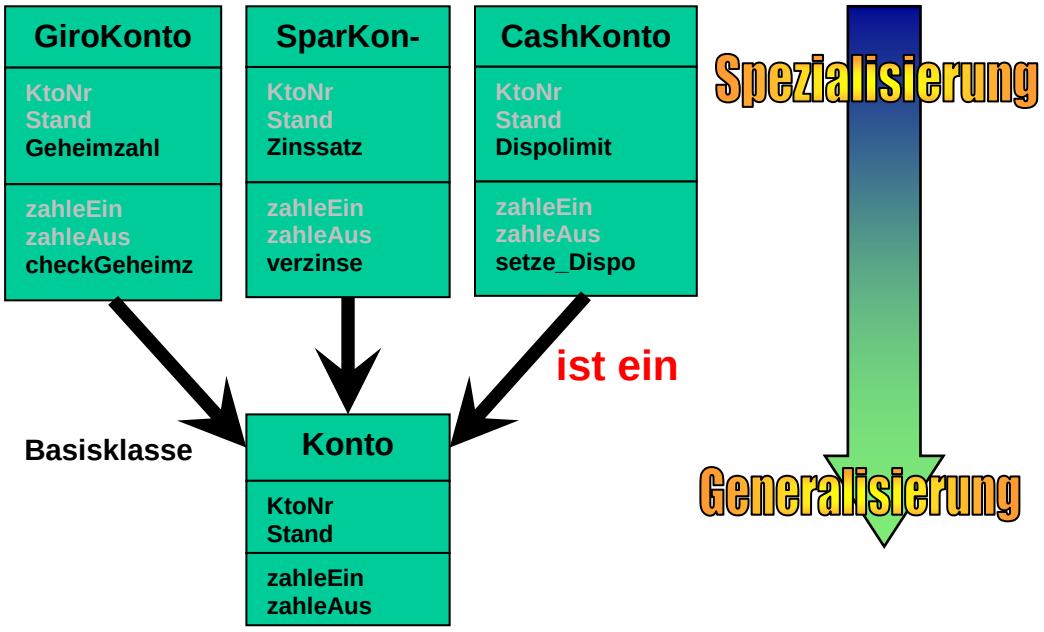


Ein Beispiel, bei dem Vererbung vorteilhaft eingesetzt werden kann, sind die verschiedenartigen Bankkonten. Wir haben es mit Girokonten, Sparkonten, Aktiendepots, Cashkonten – um nur einige zu nennen – zu tun. Alle diese verschiedenartigen Konten haben etwas gemeinsam.

Diese Gemeinsamkeit modellieren wir in einer generellen Kontoklasse und bezeichnen diese Klasse auch als **Basisklasse**. Mit Hilfe der Vererbung wollen wir diese allgemeinen Eigenschaften eines Kontos in spezielleren Kontoklassen wiederverwenden. Dadurch spart man eine Menge Arbeit, da grundlegende Attribute und Methoden, die für alle Konten gelten, nur einmalig entwickelt und getestet werden müssen. Dieses Vorgehen vermeidet auch Fehler, die beim mehrmaligen entwickeln gleicher Funktionalität entstehen können.

Im Klassendiagramm werden die Beziehung zwischen der **Basisklasse** und den **abgeleiteten Klassen** durch Pfeile dargestellt. Die Pfeilspitze zeigt dabei in Richtung der **Generalisierung**. In der anderen Richtung haben wir demnach eine **Spezialisierung**. In den speziellen Konten müssen wir dann nur noch "die Spezialitäten" des jeweiligen Kontos realisieren. Also z.B. für das Girokonto: eine Geheimnummer und eine Methode „prüft", welche die eingegebene Nummer mit der Geheimnummer des Kontos vergleicht. Oder für das Sparkonto: eine Sperrfrist, ...

Durch die Vererbung wird sichergestellt, dass die Attribute und Methoden der Basisklasse auch in den abgeleiteten Klassen verfügbar sind – wir sagen auch Attribute und Methoden **werden vererbt**. Die Beziehung zwischen abgeleiteter Klasse und Basisklasse lässt sich lesen mit "ist ein". Also z.B. Girokonto "ist ein" Konto; ebenso wie Cashkonto "ist ein" Konto oder Sparkonto "ist ein" Konto.



Wann sollte eine Klasse durch Vererbung gebildet werden?

- Eine Klasse ergibt sich als Spezialisierung oder Erweiterung einer vorhandenen (Girokonto).
- Eine Klasse vereint die Eigenschaften mehrerer vorhandener Klassen (Mehrfachvererbung).
- Zwei parallel entwickelte Klassen enthalten einen gemeinsamen Kern, der als Basisklasse verwendet werden kann.

Vorteile:

- Die abgeleitete (neue) Klasse (Unterklasse, Kindklasse) erbt (fast) alle Eigenschaften
  - Attribute
  - Methoden

  von den bestehenden Klassen (Basisklassen, Oberklassen, Elternklassen), ohne dass diese angetastet oder neu übersetzt werden müssen.

- Diese Eigenschaften können in der abgeleiteten Klasse ergänzt oder auch verändert werden:
  - weitere Methoden oder Attribute hinzufügen
  - Methoden modifizieren

  Verschiedene Klassen in einer Hierarchie können eine *gemeinsame, einheitliche Schnittstelle* haben, was den Umgang mit Objekten dieser Klasse vereinfacht.

Einschränkungen:

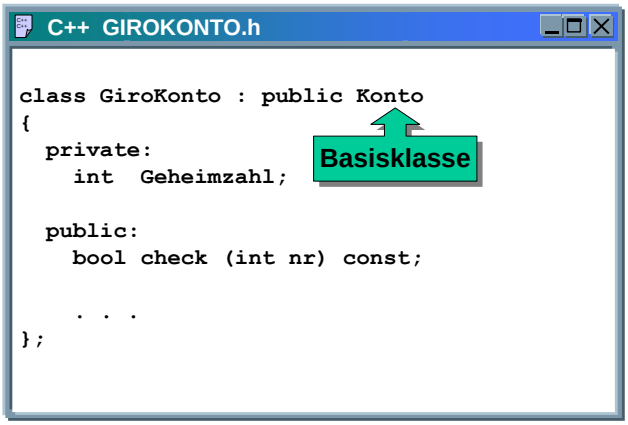
- Nicht vererbt werden:
  - Freundschaft mit einer anderen Klasse friend
  - Ein klassenspezifischer Zuweisungsoperator operator=()
  - Konstruktoren und Destruktoren

9.2 Deklaration und Zugriffsrechte

Allgemeine C++ Syntax um eine Klasse abzuleiten:

```
class AbgelKlasse : [private|public|protected] BasisKlasse, ...
{
        ...
        // Deklarationen für AbgelKlasse
        ...
};
```

Deklaration: Girokonto



Der Name der abgeleiteten Klasse wird durch **:** vom Namen der Basisklasse(n) getrennt.

Zugriffsrechte nach der Ableitung:

Grundsätzlich erbt AbgelKlasse alle Komponenten von BasisKlasse, d.h. jede Instanz von AbgelKlasse enthält ein (anonymes) Objekt vom Typ BasisKlasse. Dieses Subobjekt wird noch vor den zusätzlichen Komponenten von AbgelKlasse durch impliziten Aufruf des Basisklassenkonstruktors erzeugt.

„public“ - Ableitung

Zugriff auf / von	Basis-Klasse	Abgeleitete Klasse	Andere Klassen
private	✓	🔒	🔒
protected	✓	✓	🔒
public	✓	✓	✓

Das Zugriffsrecht in der BasisKlasse und der Zugriffsmodifizierer bei der Ableitung bestimmen das Zugriffsrecht auf Komponenten von BasisKlasse in AbgelKlasse.

private-Komponenten von BasisKlasse sind in AbgelKlasse grundsätzlich nicht zugänglich. Ein Zugriff ist hier weiterhin nur über öffentliche Methoden von BasisKlasse möglich. Durch Ableitung von einer Klasse, kann man sich also nicht den Zugriff auf private Elemente erschleichen.

- **public-Ableitung:** class AbgelKlasse : public BasisKlasse

protected-Komponenten von BasisKlasse sind auch in AbgelKlasse zugänglich, jedoch nicht allgemein.

public-Komponenten der BasisKlasse sind auch in AbgelKlasse public.

- **protected-Ableitung:** class AbgelKlasse : protected BasisKlasse

public-Komponenten von BasisKlasse werden zu protected-Komponenten von AbgelKlasse. Auf sie kann von "außerhalb" nicht mehr zugegriffen werden.

- **private-Ableitung:** class AbgelKlasse : [private] BasisKlasse

Alle public- und protected-Komponenten von BasisKlasse sind in AbgelKlasse private. Diese Form wird dann eingesetzt, wenn eine neue Schnittstelle in AbgelKlasse entstehen soll, d.h. die Implementierung wird vererbt (modelliert nicht "*ist ein*" sondern "***ist implementiert durch***").

Zugriffsattribut in der Basisklasse	Zugriffsmodifizierer bei der Ableitung		
	private	protected	public
private	Kein Zugriff	kein Zugriff	kein Zugriff
protected	private	protected	protected
public	private	protected	public
	modifiziertes Zugriffsrecht auf Basisklassenkomponenten in der abgeleiteten Klasse		

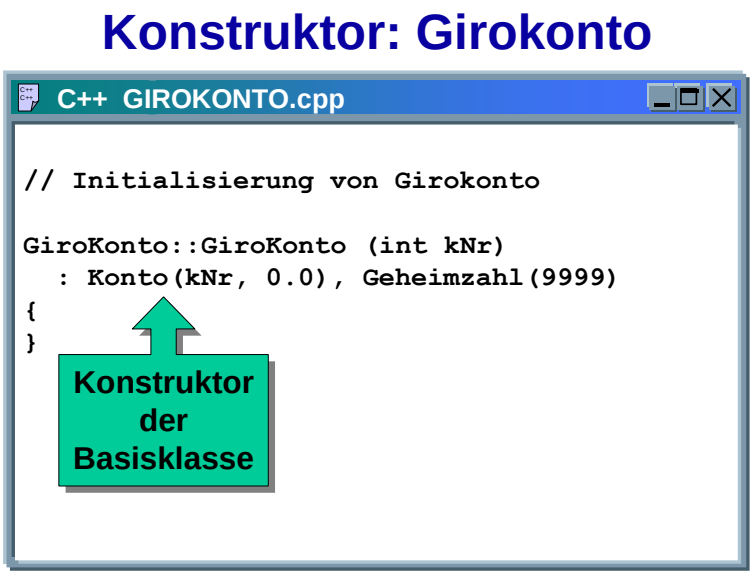
Beachte: Es gilt immer das restriktivere Zugriffsrecht!

Tipp: In der Definition von AbgelKlasse läßt sich das Zugriffsrecht *einzelner Komponenten* von BasisKlasse gezielt beeinflussen:

```
class AbgelKlasse : <mod> BasisKlasse
{
  ...
  [private|public|protected]:
     BasisKlasse::KomponentenName;          // alte
     using BasisKlasse::KomponentenName;    // neue Schreibw.
  ...
};
```

9.3 Initialisierung

Wenn wir nun ein Objekt einer abgeleiteten Klasse – also z.B. Girokonto – anlegen, stellt sich die Frage, wie werden dabei die ererbten Attribute der Klasse Konto initialisiert. Dazu gibt es genaue Regeln für die Konstruktoren abgeleiteter Klassen.



In einem Test wollen wir den Aufruf der Konstruktoren und Destruktoren durch eingebaute Ausgaben während des Programmlaufs sichtbar machen:

```
#include <iostream>
#include <cstdlib>

using namespace std;

class Konto
{
  private:
    int KtoNr;
    //...

  public:
    // Defaultkonstruktor
    Konto () : KtoNr(999)
    { cout << "1: Konto(), KtoNr: " << KtoNr << endl;  }

    // Parameterkonstruktor
    Konto (int kNr) : KtoNr(kNr)
    { cout << "2: Konto(int), KtoNr: " << KtoNr << endl; }

    // Destruktor
    ~Konto()
    { cout << "3: ~Konto(), KtoNr: " << KtoNr << endl; }
};
```

```
class Girokonto : public Konto
{
  private:
    float Dispo;

  public:
    Girokonto () : Dispo(2000)
    { cout << "4: - Girokonto(), Disp.: " << Dispo << endl;  }

    Girokonto (float d) : Dispo(d)
    { cout << "5: - Girokonto(float), Disp.: " << Dispo << endl;  }

    Girokonto (int kNr, float d) : Konto(kNr), Dispo(d)
    { cout << "6: - Girokonto(int,float), Disp.: " << Dispo << endl;  }

    ~Girokonto ()
    { cout << "7: - ~Girokonto(), Disp.: " << Dispo << endl;  }
};

int main()
{
  Girokonto a, b(5500.0), c(123,1000.0);
  cout << "Weiter mit Taste ....." << endl;
  system("pause");
}
```

Ausgabe:

```
1: Konto(), KtoNr: 999
4: - Girokonto(), Disp.: 2000
1: Konto(), KtoNr: 999
5: - Girokonto(float), Disp.: 5500
2: Konto(int), KtoNr: 123
6: - Girokonto(int,float), Disp.: 1000
Weiter mit Taste .....
7: - ~Girokonto(), Disp.: 1000
3: ~Konto(), KtoNr: 123
7: - ~Girokonto(), Disp.: 5500
3: ~Konto(), KtoNr: 999
7: - ~Girokonto(), Disp.: 2000
3: ~Konto(), KtoNr: 999
```

Erkenntnisse:

- Komponenten von Basisklassen können nur über ihre Konstruktoren initialisiert werden.

- Der Konstruktoraufruf erfolgt entweder
  - *implizit*: Defaultkonstruktor (falls vorhanden) oder muss
  - *explizit* erfolgen: in der Initialisierungsliste

- Ein *expliziter* Konstruktoraufruf muss erfolgen, wenn Parameter übergeben werden sollen.

- Eine direkte Initialisierung geerbter Member (= Attribute oder Methoden) über die Initialisierungsliste ist *nicht möglich*!

- Reihenfolge der Konstruktion:
  - *Basisklassen* in der Reihenfolge ihrer Deklaration in der Ableitungsliste (nicht in der Initialisierungsliste), danach die
  - *Member* der abgeleiteten Klasse in der Reihenfolge ihrer Deklaration.

- Reihenfolge der Destruktion: genau umgekehrt