Einführung in die Programmierung mit C++

Joachim Goll, Ulrich Harms,
Dominik Bruhn, Roman Keller, Anton Trojosky

September 2005

Inhaltsverzeichnis

1. Einführung in die Programmierung

1.1. Bits und Bytes

Was ist ein Bit?

- Kunstwort: binary digit (Binärziffer)
- Maß für die Größe bzw. den Umfang von Daten in der Informationstechnik
- Die Informationsmenge 1 Bit entspricht der Information, welche von zwei möglichen Begebenheiten zutrifft.

Beispiele:

- Die Stellung eines Schalters mit zwei Zuständen, zum Beispiel eines Lichtschalters mit den Stellungen EIN oder AUS.
- Das Vorhandensein einer Spannung, die größer oder kleiner als ein vorgegebener Wert ist.
- Eine Variable, welche einen von zwei Werten, zum Beispiel 0 oder 1, die logischen Wahrheitswerte Wahr oder Falsch, high oder low, H oder L enthalten kann.

Mit einem Bit lassen sich also 2 Zustände Speichern (0 oder 1). Wenn man nun 2 Bits nimmt lassen sich schon 4 Zustände Speichern, nämlich 2*2=4 (00; 01; 10; 11).
Mit 4 Bits lassen sich dann 16 Zustände darstellen und mit 8 Bits sind es dann 256 Zustände.

8 Bits werden wiederum zu einem Byte zusammengefasst. Dann gibt es wieder größere Einheiten:

Name	Symbol	Mehrfaches in Byte
Kilobyte	KByte	10^3 (oder 2^{10})
Megabyte	MByte	10^6 (oder 2^{20})
Gigabyte	GByte	10^9 (oder 2^{30})
Terabyte	TByte	10^{12} (oder 2^{40})
Petabyte	PByte	10^{15} (oder 2^{50})

Name	Symbol	Mehrfaches in Bit
Kilobit	KBit	10^3 (oder 2^{10})
Megabit	MBit	10^6 (oder 2^{20})
Gigabit	GBit	10^9 (oder 2^{30})
Terabit	TBit	10^{12} (oder 2^{40})
Petabit	PBit	10^{15} (oder 2^{50})

Problem mit Umrechnung. 10^3 (1000) unterscheidet sich um 24 von 2^{10} (1024). Wegen der nur kleinen Differenz und einem nur geringen Fehler, verzichtet man oft auf eine Unterscheidung.

1.2. Maschinensprache – 1. Generation

Wie schon erwähnt speichert ein Computer alles in zwei Zuständen, 0 und 1. Ein Programm in Maschinensprache ist demnach eine Folge von Binärzahlen wie z.B.:

```
00111010
01100000
00000000
01000111
00111010
01100001
00000000
10000000
00110010
01100010
00000000
```

Das Programmieren in Maschinensprache ist mit etlichen Schwierigkeiten verbunden:

- Die Programme beschreiben die Aufgabe, die der Computer ausführen soll, in einer für den Menschen nur schwer verständlichen Form.
- Der Programmierer macht häufig Fehler, da Binärzahlen einander sehr ähnlich sehen, insbesondere wenn man bereits einige Stunden mit ihnen gearbeitet hat.
- Die Eingabe der Programme ist sehr langwierig, da man jedes Bit individuell bestimmen muss.

Der erste Fortschritt war der Assembler.

1.3. Assembler – 2. Generation

Eine offensichtliche Verbesserung des Programmierens stellt die Zuweisung einer Bezeichnung zu einem Befehl dar. Diese Bezeichnungen, die aus drei oder vier Buchstaben bestehen, sollen in irgendeiner Form beschreiben, was ein Befehl ausführt. In der Tat liefert jeder Hersteller eines Prozessors einen Satz von diesen Bezeichnungen für den Befehlssatz seines Prozessors. Sie sind Standard für einen gegebenen Prozessor.

Ein Programm mit den Standardbezeichnungen von Intel sieht z.B. folgendermaßen aus:

```
MOV AX,800H
AND AX,BX
JNZ MARKE
MOV AX,1000H
```

Diese Darstellung ist wesentlich anschaulicher als ein Maschinencode. Man bezeichnet diese Schreibweise als **Assemblersprache**.

Die Übersetzung eines in Assemblersprache geschrieben Quellprogramms in die Maschinensprache erfolg mit Hilfe eines dafür zuständiger Programms, dem **Assembler**.

Nachteile des Assemblers

Beim Programmieren in Assemblersprache muss man eine sehr detaillierte Kenntnis des verwendeten speziellen Prozessors besitzen. Man muss z.B. wissen, welche Befehle und Speicher der Prozessor hat, wie die Befehle die verschiedenen Register beeinflussen, welche Adressier-Verfahren der Prozessor verwendet. Keine dieser Informationen ist für die Aufgabe, die der Computer letztlich ausführen muss, relevant.

Programme in Assemblersprache sind zwischen Prozessortypen verschiedener Herstellen in der Regel nicht übertragbar, da die Assemblersprache auch durch die Architektur des Prozessors bestimmt wird.

1.4. Programmiersprachen der 3. Generation

Die Lösung vieler der mit Assemblersprache-Programmen verbundenen Schwierigkeiten ist die Verwendung von „höheren" Sprachen der 3. Generation, von so genannten „problemorientierten" Sprachen.

Derartige Sprachen gestatten die Beschreibung von Aufgaben in einer Art und Weise, die eher problemorientiert als computerorientiert ist. Jede Anweisung in einer höheren Programmiersprache führt eine erkennbare Funktion aus. Sie wird im Allgemeinen mehreren Befehlen in Assemblersprache entsprechen.

Ein Programm, das Kompiler genannt wird, übersetzt ein Quellprogramm in einer höheren Sprache in Maschinencode.

Man kann grob gesprochen sagen, dass ein Programmierer ein Programm zehnmal schneller in einer höheren Sprache als in Assemblersprache schreiben kann. Die Programmierer können sich auf ihre eigentliche Aufgabe konzentrieren. Sie brauchen nur noch geringe Hardware-Kenntnisse über den Computer, den sie programmieren.

Dennoch kann es Fälle geben, wo der Einsatz von Assembler gegenüber höheren Programmiersprachen von Vorteil ist, wie z.B. bei der Ansteuerung von Hardware-Schnittstellen in Geräten.

Generell sollte man den Einsatz von Assembler auf kleine Programme beschränken, die entweder sehr zeitkritisch sind und eine hohe Effizienz erfordern, die bei höheren Programmiersprachen in diesem Maße nicht gegeben ist, oder bei der Ansteuerung von Hardware-Bausteinen in Geräten.

1.5. Debugger

Ein Debugger dient zur Fehlersuche. Mit ihm kann man ein Programm in Einzelschritten durchforsten.

1.6. Zusammenfassung

- Ein Prozessor versteht nur Maschinenbefehle
- Maschinenbefehle sind Folgen von Nullen und Einsen z.B. 11110010
- Befehle in der Programmiersprache Assembler enthalten Bezeichnungen wie z.B. MOV, AND.
- Jeder Prozessor hat seinen speziellen Befehlssatz.
- Eine höhere Programmiersprache ist unabhängig vom speziellen Prozessor.
- Ein in einer höheren Programmiersprache geschriebenes Programm wird durch einen Kompiler und Linker in Maschinencode übersetzt und in ein ausführbares Programm umgewandelt.

2. Einführung in Visual Studio 2003 .NET

2.1. Die integrierte Entwicklungsumgebung

Die integrierte Entwicklungsumgebung (zu Englisch: integrated development environment, Abkürzung DIE) hilft dem Entwickler bei seiner täglichen Arbeit. Sie bietet Abkürzungen an und vereint viele verschiedene Funktionen in einem Programm. So findet sich in der IDE "Visual Studio .NET 2003" von Microsoft ein Compiler, ein Linker, ein Debugger, eine Quellcodeverwaltung, eine sehr ausführliche Hilfe und diverse weitere Funktionen, die das Programmieren vereinfachen und beschleunigen können.

2.2. Grundbegriffe von Visual Studio

- **Quellcodedateien** (**Source Files**): Enthalten C++ Code, der mit dem Befehl Kompilieren in Maschinen-Code umgewandelt wird.
- **Projekt** (**Project**): Enthält ein oder mehrere Quelldateien. Aus diesen Dateien entsteht beim Kompilieren und Linken eine Anwendung. Pro Projekt kann also nur eine Anwendung erstellt werden.
- **Projektmappe** (**Solution**): Enthält ein oder mehrere Projekte zum selben Thema. Alle Projekte der Projektmappe werden im selben Ordner gespeichert.

2.3. Ordner für Dateien anlegen

- Mit dem Windows-Explorer oder dem Arbeitsplatz anlegen. Der Order darf keine Sonderzeichen wie Leerzeichen oder ö, ü, ä, ß enthalten.

2.4. Visual Studio starten

- Visual Studio aus dem Startmenü starten.

2.5. Neue Projektmappe anlegen

- Sollten noch Projekte geöffnet sein (Arbeitsfläche ist nicht grau), dann klickt man im Menü auf **File** (**Datei**), **Close Solution** (**Projektmappe schließen**).
- Um eine neue Projektmappe anzulegen, klickt man auf **File** (**Datei**), **New** (**Neu**) und auf **Blank Solution** (**Leere Projektmappe**).
- Bei Name wird ein Name für die Projektmappe festgelegt. Außerdem wird der Speicherordner mit **Location** (**Speicherort**) auf den neu angelegten Ordner gesetzt.
- Nach einem Klick auf **OK** wird die Projektmappe angelegt.

2.6. Projektmappe anzeigen

- Um den Inhalt einer Projektmappe anzuzeigen und deren Inhalt zu bearbeiten klickt man auf **Solution Explorer** (**Projektmappen Explorer**). Danach sollte der Projektmappen-Explorer erscheinen.

2.7. Ein Projekt einrichten

- Zum Anlegen eines neuen Projekts klickt man mit der rechten Maustaste auf die **Solution** (**Projektmappe**) Eintrag im Projektmappen-Explorer. Danach auf **Add** (**Hinzufügen**) und auf **New Project** (**Neues Projekt**).
- Im neuen Fenster klickt man unterhalb von **Project Types:** (**Projekttypen:**) auf **Visual C++ Projects** (**Visual C++ Projekte**) und auf **Win32**. Rechts unterhalb von **Templates** (**Vorlagen**) wählt man nun **Win32 Console Project** (**Win32 Konsolen Projekt**) aus.
- Bei Name wird ein Name für das Projekt festgelegt. **Location** (**Speicherort**) wird nicht verändert.

- Im neuen Fenster auf **Application Settings** (**Anwendungseinstellungen**) klicken. Dort setzt man den Haken bei **Empty Project** (**Leeres Projekt**). **Dies ist ein wichtiger Schritt, sonst funktionieren alle Quellcodes in diesem Skript nicht.**
- Nach einem Klick auf **Finish** (**Fertigstellen**) wird das Projekt angelegt.
- Im **Projektmappen-Explorer** erscheint nun ein neuer Punkt für das Projekt mit den Unterpunkten **References** (**Verweise**), **Source Files** (**Quelldateien**), **Header Files** (**Headerdateien**) und **Resource Files** (**Ressourcendateien**).

2.8. Eine neue Quelldatei anlegen

- Um eine neue Quelldatei anzulegen klickt man mit der rechten Maustaste auf das Projekt. Im Kontext-Menü wählt man nun **Add** (**Hinzufügen**) und dann auf **Add New Item** (**Neues Element hinzufügen**).
- Im darauf folgenden Fenster wählt man **C++ File (.cpp)** (**C++ Datei (.cpp)**). Danach wird ein Name festgelegt. Auch hier bleibt der Speicherort unverändert. Der Name darf dabei keinerlei Sonderzeichen wie ö, ü, ä oder Leerzeichen enthalten.
- Nach einem Klick auf **Open** (**Öffnen**) legt Visual Studio die Datei an.
- Anschließend zeigt Visual Studio die gerade angelegte Datei an.
- In dem Fenster kann nun der Code getippt werden.

2.9. Ausführbares Programm erstellen

- Zuerst muss der Code kompiliert werden. Hierzu klickt man auf den Menüpunkt **Build** (**Erstellen**) und dann auf **Compile** (**Kompilieren**). Der Tastaturkürzel hierzu lautet *STRG + F7.*
- Im **Output**-Fenster (untere Bildschirm-Rand) muss *„Build: 1 succeeded, 0 failed, 0 skipped*" erscheinen.
- Die kompilierten Dateien müssen nun noch gelinkt werden. Hierzu wählt man im Menü **Build** (**Erstellen**) den Punkt **Build [projektname]**(**[projektname] erstellen**) aus. Auch hier muss im **Ausgabe**-Fenster die oben angegebene Bestätigung erscheinen.

2.10. Programm starten

- Um das Programm nun zu starten klickt man im Menü **Debug** (**Debuggen**) auf **Start without Debugging** (**Starten ohne Debuggen**; damit sich das Programmfenster nicht automatisch schließt). Der Tastaturkürzel hierzu lautet *STRG + F5*.
- Nach dem Ende des Programms muss mit einem beliebigen Tastendruck das Fenster geschlossen werden.
- Danach kann Visual Studio wie beschrieben benutzt werden.

2.11. Ein zweites Projekt in der Projektmappe anlegen

- Vorgehensweise identisch mit 2.6.-2.8.
- Beim Starten des Projekts wie in 2.10. wird aber noch das zuerst angelegte Projekt gestartet und nicht das zweite.
- Um dies zu ändern klickt man im Projektmappen-Explorer mit der rechten Maustaste auf das zweite Projekt (jenes das gestartet werden soll) und wählt den Menüpunkt **Set as StartUp Project** (**Als Startprojekt festlegen**)
- Nun wird beim Starten des Projekts nach 2.10. das neu angelegte Projekt gestartet.

2.12. Anpassen der Symbolleiste von Visual Studio

- Da die 3 Befehle (Kompilieren, Linken, Starten) mehrere Mausklicks benötigen ist es möglich eine Abkürzung anzulegen, in dem man die Befehle als Symbole in die Symbolleiste einfügt.
- Hierzu klickt man im Menü auf **Tools** (**Extras**) und anschließend auf **Customize** (**Anpassen**).

- In dem Dialogfeld wählt man unter Kategorie den Punkt **Build** (**Erstellen**) an und zieht mit gedrückter Maustaste die Befehle **Compile** (**Kompilieren**) und **Build Selection** (**Auswahl erstellen**) auf die Symbolleiste rechts neben das Auswahlmenü mit dem Eintrag **Debug**. Zusätzlich wird aus der Kategorie **Debug** (**Debuggen**) der Punkt **Start without Debugging** (**Starten ohne Debugge**n) neben die beiden schon erstellen Symbole gezogen.
- Nach einem klick auf **Close** (**Schließen**) können die Symbole in der Symbolleiste wie die Einträge aus den Menüs benutzt werden.

2.13. Shortcut

Zum gleichzeitigen Kompilieren, Linken und Ausführen kann man auch einfach *STRG+F5* drücken, Visual Studio fragt dann nach ob man vor dem Ausführen kompilieren will. Diese Nachfrage sollte mit ja beantwortet werden

3. Unser erste Programm

Unser erste Programm hallo.cpp schreibt einen Satz auf den Bildschirm.

```
// halloWelt.cpp – 16.8.2005 – Roman Keller
// Schreibt einen Satz auf den Bildschirm aus.

#include <iostream>                        // Einbindung von iostream

using namespace std;                       // Definition von std für cout

int main()                                 // Funktionskopf
{                                          // Funktionsrumpf Anfang
      cout << "Hallo Welt!" << endl;       // Ausgabe des Textes auf den
                                           // Bildschirm
      return 0;                            // Rückgabewert
}                                          // Funktionsrumpf Ende
```

3.1. Allgemeiner Programmaufbau

`include`

Bestimmte Befehle sind in C++ in Bibliotheken hinterlegt und zusammengefasst. Diese Bibliotheken müssen - wenn man ihre Befehle benutzen will – eingebunden werden. So ist z.B. `cout` in der Bibliothek `iostream` (Input-Output-Stream) hinterlegt. Da dies eine Anweisung an den vorgeschalteten Kompiler (Präprozessor) ist, muss man hier kein Semikolon ans Ende schreiben.

`namespace`

Um Verwechslungen und mehrfache Namensgebung zu vermeiden, werden die Befehle in C++ in sog. Namespaces (Namensräume) zusammengefasst. Normalerweise müsste man `std::cout` schreiben, aber um sich diesen erhöhten Schreibaufwand zu ersparen, sind mit `using namespace std;` alle Befehle aus dem Namesraum std dem Compiler bekannt.

`main()`

main() die Hauptfunktion des Projekts. Hier fängt der Programmablauf an. main() wird von dem Betriebssystem aufgerufen.

`Klammernblock { }`

Die geschweiften Klammern `{` und `}` begrenzen den Funktionsrumpf und einen logischen Block. Alle Befehle innerhalb eines Blocks werden zwecks Übersichtlichkeit eingerückt. Dies erledigt Visual Studio in der Regel von selbst. Falls nicht kann dies mit TAB geschehen. Dies ist notwendig um die Übersichtlichkeit des Quelltextes zu wahren.

`return`

Meldet einen Wert, in diesem Fall 0, an das Betriebssystem zurück. Damit kann auf Seiten des Betriebssystems überprüft werden, ob das Programm erfolgreich beendet wurde (Rückgabewert gleich 0).

`Kommentare`

Kommentare dienen zur vereinfachten Lesbarkeit des Programmcodes. Hier können z.B. Gedankengänge oder Erläuterungen niedergeschrieben werden.
Kommentare werden vom Kompiler ignoriert.

- Einzeilige Kommentare (`// ...`)
  Ein einzeiliger Kommentar wird mit `//` eingeleitet. Alles was dahinter in derselben Zeile steht wird von Kompiler ignoriert.

- Blockkommentare (`/* .... */`)
  Ein Blockkommentar beginnt mit /* und endet mit */. Alles dazwischen wird von Kompiler ignoriert.

`cout`

Mithilfe von `cout` können Zeichen an den Bildschirm ausgegeben werden.
Dazu später mehr.

In C++ werden alle Anweisungen mit einem Semikolon (Strichpunkt) ; abgeschlossen, außer wenn es sich dabei um eine Präprozessoranweisung handelt. Diese beginnen mit einem #.

Alle Wörter die vom Visual Studio blau gefärbt werden, sind so genannte reservierte Wörter. Diese haben eine feste Bedeutung in C++ und können nicht anderweitig verwendet werden.

Grundsätzlicher Aufbau eines C++-Programms:
- Kommentare: Dateiname – Datum – Autor; Aufgabe/Funktionsweise
- Includes
- Namespaces (Namensräume öffnen)
- Funktionskopf
- Blockklammer auf {
- Anweisungen
- Rückgabeanweisung
- Blockklammer zu }

3.2. Die Bildschirmausgabe: `cout`

Wie schon erwähnt, kann mit `cout` etwas an den Bildschirm ausgegeben werden.
Alle einzelnen Argumente werden mit **<<** angeführt. Die Argumente werden auch symbolisch `cout` übergeben.

Texte werden mit Anführungszeichen beginnend und abschließend ausgegeben.

Um einen Zeilenumbruch zu erzwingen, benützt man `endl` (EndLine).

```
cout << "Die erste Zeile" << endl << "Die zweite Zeile";
```

4. Datentypen und Variablen

4.1. Grundbegriff Variable

In Variablen können Inhalte gespeichert und wieder ausgelesen werden. Man kann die Inhalte verändern oder mit ihnen mathematische Operationen durchführen. Sie repräsentieren eine bestimmte Stelle im Arbeitsspeicher. Dieser kann mit Hilfe der Variablen angesprochen werden. Jede Variable hat immer einen **Namen**, einen **Typ** und einen **Inhalt**.

4.2. Mögliche Namen

Bei der Definition von Variablen sollte man sich an einige Regeln halten.

Mögliche Zeichen
a .. z; A .. Z; 0..9; Also keine Umlaute und ß.
Am Anfang darf keine Ziffer stehen.

Sinnvolle Namensgebung
Der Name sollte selbsterklärend sein, so dass es für Dritte nachvollziehbar ist. Beim späteren Benutzen der Variablen sollte man den Namen logisch herleiten können und wissen welchen Zweck sie hat.

Übersichtlichkeit
Variablennamen werden der Übersichtlichkeit halber mit kleinem Anfangsbuchstaben geschrieben. Weitere Wörter dahinter werden dann mit großem Anfangsbuchstaben geschrieben (z.B.: timeCount).

Beispiel:
isNew, input1, meinName, deinNummernSchild

4.3. Mögliche Typen

Variable können unterschiedliche Aufgaben haben. Manchmal muss mit ihnen gerechnet werden oder sie sollen einen Text speichern.
Hierfür gibt es verschiedene Typen, die eine Variable annehmen kann.
Welchen Typ eine Variable für das Programm hat, muss schon bei ihrer Erstellung festgelegt werden. Einige dieser Typen werden im Folgenden kurz erklärt.

Name	**Typ**	**Größe**	**Beschreibung**
`int`	Ganzzahl	4 Byte 32 Bit	Ganzzahlen zwischen -2147483648 und 2147483647
`char`	Zeichen	1 Byte 8 Bit	Hier kann nur ein Zeichen gespeichert werden
`string`	Zeichenkette	1 Byte/8 Bit pro Zeichen	Hier kann eine Zeichenkette gespeichert werden. Es muss noch die Bibliothek `string` und der `namespace std` eingebunden werden.
`bool`	Boolesche Variable	1 Byte 8 Bit	Bool kann zwei Zustände annehmen `true` (wahr) und `false` (falsch)
`float`	Fließkommazahl	4 Byte 32 Bit	Rationale Zahlen mit oder ohne Punkt, 7 Stellen

Name	Typ	Größe	Beschreibung
`double`	Fließkommazahl	8 Byte 64 Bit	Rationale Zahlen mit oder ohne Punkt, 15 Stellen

4.4. Definition von Variablen

In C++ werden Variablen wie folgt definiert:

```
[VARTYPE] [VARNAME];
```

Variablen können in C++ überall im Quelltext definiert werden. Allerdings sollte man auf Grund der Übersichtlichkeit darauf verzichten und sie jeweils am Anfang eines Blocks definieren (Hinter den {-Klammern), am Besten am Anfang einer Funktion.

```
int meineZahl;
```

Man kann auch mehrere Variable des gleichen Typs auf einmal definieren.

```
int meineZahl, meineZweiteZahl;
```

4.5. Wertzuweisung an Variablen

Um mit den Variablen arbeiten zu können, muss man ihnen Werte zuweisen. Dies erfolgt mithilfe des = Operators.

Die Wertzuweisung kann auch gleich mit der Definition erfolgen um z.B. Anfangswerte zu setzen oder einfach Schreibarbeit zu sparen.

Beispiel:

```
int meineZahl = 10;
```

Solange man einer Variablen keinen Wert zugewiesen hat, besitzt sie einen zufälligen Wert, der beim Aufruf der Variable zu Fehlern im Programmablauf führen kann. Normalerweise würde man erwarten, dass ein `int` automatisch 0 als Wert erhält. Dies ist aber nicht richtig. Deshalb sollten wir darauf achten, dass wir einer Variablen immer einen Wert zuweisen bevor wir sie benutzen.

- Ganzzahl **`int`**

Einem int können Ganzzahlen zugewiesen werden.

```
int meineZahl = 312;
```

- Zeichen **`char`**

Einem `char` können einzelne Zeichen zugewiesen werden.
Die Zeichen werden mit Apostrophe ' geschrieben.

```
char meinZeichen;
meinZeichen = 'z';
```

- Zeichenkette **`string`**

Einem `string` kann eine beliebig lange Zeichenkette zugewiesen werden.
Diese Zeichenkette muss von Anführungszeichen " begrenzt werden, damit der Kompiler das Ende und den Anfang erkennt.

```
string meineZeichenkette = "Das ist meine Zeichenkette";
```

Um in einem `string` ein doppeltes Anführungszeichen zu speichern ohne das der Kompiler dies als das Ende des `strings` ansieht, muss man stattdessen ein `\"` an Stelle von " schreiben.

```
string meineZeichenkette;
meineZeichenkette = "Das ist ein Anführungszeichen: \" ";
```

Außerdem sollte auf die Verwendung von Umlauten und ß verzichtet werden.

Da `string` nicht zu den fundamentalen Datentypen in C++ gehört, sondern in der Bibliothek `string` im `namespace std` hinterlegt ist, muss bei der Verwendung von Zeichenkette die Bibliothek `string` und der `namespace std` eingebunden werden.

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

- Boolean **`bool`**

Einem `bool` kann entweder `true` (wahr) oder `false` (falsch) zugewiesen werden.

```
bool meineBool;
meineBool = true;
```

- Fließkommazahl **`float`**

Einem `float` kann eine Zahl mit oder ohne „Komma" zugewiesen werden. Die Zahl kann 6 Stellen haben, wobei die Stelle des Kommas beliebig ist. Außerdem ist das Vorzeichen frei wählbar.

Achtung: Im Englischen werden der Punkt und das Komma anders verwendet. Bruchzahlen werden im Englischen mit einem Punkt getrennt. (1.5) (deutsch: 1 320,5 => englisch: 1 320.5)

```
float meineFKZ = 34.45436f;
```

Das `f` hinter der Zahl sagt dem Kompiler, dass es sich bei der Zahl um einen float und nicht um einen double handelt.

- Fließkommazahl **`double`**

Einem double kann eine 15-stellige Zahl mit oder ohne Komma zugewiesen werden. Wobei dabei die Stelle des Punktes frei wählbar ist. Das Vorzeichen ist ebenfalls frei wählbar.

```
double meineFKZ = 343.45643534543;
```

4.6. Operationen mit Variablen

Um mit Variablen zu rechnen, brauchen wir Operatoren. Rechnen kann man nur mit Variablen die vom Kompiler auch als Zahlen interpretiert werden. Also z.B. int oder float.

Operator	**Funktion**
+	Addition
-	Subtraktion
*	Multiplikation
/	Division
%	Modulo (Rest der Division) `10 % 3 = 1 ( 3 * 3 = 9; 10 – 9 = 1)`
++	Inkrementieren, um eins erhöhen: `++z=> z = z + 1;`
--	Dekrementieren, um eins verringern: `--z => z = z – 1;`

Operator	**Funktion**
+=	Plus-Gleich-Operator: `z += y` entspricht `z = z + y;`
-=	Minus-Gleich-Operator: `z -= y` entspricht `z = z - y;`
*=	Mal-Gleich-Operator: `z *= y` entspricht `z = z * y;`
/=	Geteilt-Gleich-Operator: `z /= y` entspricht `z = z / y;`
%=	Modulo-Gleich-Operator: `z %= y` entspricht `z = z % y;`

Beispiel:

```
ergebnis = zahl1 * zahl2;
++zahl1;                          // zahl1 = zahl1 + 1; zahl 1 += 1;
```

Wenn man Zeichenketten (`strings`) miteinander verbinden will, so geht dies mit dem + Operator.

Beispiel:

```
string zeichenKette = "Das ist meine Zeichenkette";
string dasKommtDazu = " und das kommt dazu!";
string ergebnis = zeichenKette + dasKommtDazu;
// ergebnis = "Das ist meine Zeichenkette und das kommt dazu!"
```

Oder mit dem **+=** Operator:

```
string zeichenKette = "Das ist meine Zeichenkette";
string dasKommtDazu = " und das kommt dazu!";
zeichenKette += dasKommtDazu;
// zeichenKette = "Das ist meine Zeichenkette und das kommt dazu!"
```

4.7. Gültigkeit von Variablen

Variablen gelten nur unterhalb der Stelle und in dem Block in dem sie definiert wurden. Also nur zwischen den engsten geschweiften Klammern.

```
...
{
    int test = 4;
    cout << test;
}
cout << test;           // FEHLER: test gibt es nicht mehr
```

So ist es auch innerhalb einer Funktion. Eine Variable, die in einer Funktion definiert wurde, ist außerhalb der Funktion nicht mehr gültig.
Um die Übersicht zu behalten sollte man seine Variablen am Anfang definieren und nicht über den Code verstreut. Es kann sonst zu Fehlern kommen, wenn die Variable außerhalb des Blocks aufgerufen wird.

4.8. Die Tastatureingabe: `cin`

Nun wollen wir in unserem Programm Eingaben vom Benutzer bearbeiten. Dazu müssen wir diese von der Tastatur lesen. Dies erfolgt mithilfe des Befehls `cin`. Dieser funktioniert ähnlich wie `cout`. Allerdings zeigen die **>>** auf die Variable nicht auf `cin`, da in die Variable geschrieben wird.
`cin` kann Zahlen, Zeichen und Zeichenketten von der Tastatur lesen.

```
int myZahl;
cin >> myZahl;
```

Vor dem Befehl `cin` wird gewartet, bis der Benutzer etwas auf der Tastatur eingibt. Wenn er die Enter-Taste drückt wird die Zahl die er eingegeben hat in der Variable `myZahl` gespeichert.
Beim Einlesen von `strings` beendet `cin` den Lesevorgang allerdings schon ab dem ersten Leerzeichen. Die Eingabe am Bildschirm geht zwar weiter, wird aber nachher nicht in die `string` Variable gespeichert.

```
string myZK;
cin >> myZK;            //Eingabe z.B. von: "Roman Keller"
cout << myZK;           //Ausgabe ist "Roman"
```

Die Eingabe muss typengerecht erfolgen. Wenn bei der Eingabe eine Zahl erwartet wird, aber ein Zeichen bzw. eine Zeichenkette eingegeben wird, so wird `cin` einfach übergangen (solange man keine Fehlerbehandlung).

```
int myZahl = 0;
cin >> myZahl;          //Eingabe von einem Zeichen z.B.: a
cout << myZahl;   //Ausgabe ist 0, also der Anfangswert
```

4.9. Zusammenfassung

- In C++ muss man Bibliotheken importieren um vorgefertigte Funktionen nutzen zu können. z.B. `<iostream>` für `cout` und `cin`
- Alle Anweisungen werden mit einem Semikolon (`;`) abgeschlossen.
- Mit `cout` kann an den Bildschirm ausgegeben werden.
- Mit `cin` kann man von der Tastatur lesen.

- Variablennamen dürfen nur Buchstaben und Ziffern enthalten, wobei keine Ziffer am Anfang stehen darf.
- Bei der Definition einer Variablen kommt erst der Variablentyp und dann der Variablenname.
- Man kann Variablen gleich bei der Definition einen Wert zuweisen.
- In C++ werden Gleitkommazahlen mit einem Punkt anstelle des Kommas geschrieben (engl. Schreibweise)
- Mit Variablen, die eine Zahl repräsentieren (`int, float, double`), kann man normal rechnen. Zeichenketten (`string`) können „nur" aneinander gefügt werden.
- Variablen sind nur innerhalb eines Logischen Blocks gültig (`{ ... }`).

5. Praktikum 1

5.1. Aufgabe 1 – Text ausgeben

Geben Sie einen beliebigen Text auf den Bildschirm aus.

5.2. Aufgabe 2 - Name

Lesen Sie den Namen des Benutzers ein (Achtung: Keine Leerzeichen) und geben sie einen Begrüßungstext aus.

5.3. Aufgabe 3 - Rechnen

Lesen Sie zwei Zahlen ein und wenden sie alle Grundrechenarten auf die beiden Zahlen an und geben sie die Ergebnisse auf den Bildschirm aus.

5.4. Aufgabe 4 – Sekunden seid Geburt

Lesen Sie das Geburtsdatum und das aktuelle Datum ein und berechnen sie näherungsweise die Sekunden seit der Geburt, also Monate mit 30 Tagen und Jahre mit 365 Tagen.

6. Kontrollstrukturen

Kontrollstrukturen werden, wie der Name schon sagt, zur Steuerung bzw. Kontolle des Programmablaufs verwendet.

6.1. Verzweigungen

Was ist eine Bedingung?

Beispiel: Baum

- Baum braucht Wasser zum Wachsen.
- Baum braucht Sonne oder künstliches Licht zum Wachsen.

Man kann nun verschiedene Bedingungen und deren Folgen formulieren:
- Wenn Wasser vorhanden ist, dann kann ein Baum wachsen.
- Wenn Wasser und Sonne vorhanden sind, dann kann ein Baum wachsen.

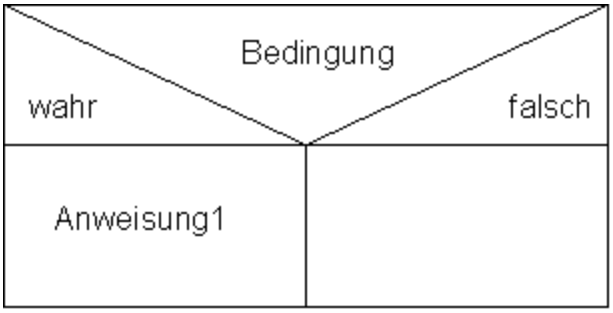
Auf diese Bedingung, ob ein Baum Wasser hat kann man immer mit ja oder nein antworten. Genauso ist es in C++. Die Bedingung ist entweder erfüllt (`true`) oder nicht erfüllt (`false`).

6.1.1. Einfache Alternative – `if` und `else`

Wenn man das nun in die Syntax eines Programms übersetzt, ergibt sich folgendes.

6.1.1.1. Allgemeine Syntax

```
if (Ausdruck)
{
      Anweisung(en);
}
```



Wenn die Bedingung erfüllt (`true`) ist oder zutrifft, wird der Anweisungsblock ausgeführt.

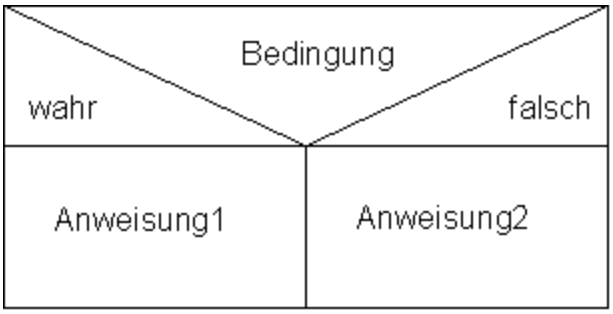
Bedingung: Wenn Wasser vorhanden ist, dann kann ein Baum wachsen.
Übersetzung:
wenn -> `if`

```
if (Wasser)
{
      baumWachse();
}
```

`baumWachse()` wird ausgeführt, wenn `(Wasser)` erfüllt ist. Wenn `(Wasser) false` (falsch) ist dann ist die Bedingung nicht erfüllt und `baumWachse()` wird nicht ausgeführt.

Bedingung: Wenn Wasser vorhanden ist, dann kann ein Baum wachsen, *sonst trocknet er aus.*
Übersetzung:
sonst -> `else`

```
if (Wasser)
{
      baumWachse();
}
else
{
      baumTrockneAus();
}
```



`baumTrockneAus()` wird ausgeführt, wenn die angegebene Bedingung nicht erfüllt (`false`) ist.

6.1.1.2. Vergleichsoperatoren

- Gleichheitsoperator ==
- Ungleichheitsoperator !=
- Größeroperator >
- Kleineroperator <
- Größergleichoperator >=
- Kleinergleichoperator <=

Bedingung: Wenn Wasser und Sonne vorhanden sind, dann kann ein Baum wachsen.

```
if ( (Wasser) && (Sonne) )
{
      baumWachse();
}
```

Beide angeführten Bedingungen müssen wahr sein, damit `baumWachse()` ausgeführt wird. Die zwei Einzelbedingungen werden dazu mit einer weiteren Klammer zusammengefasst (fett gedruckt).

6.1.1.3. Logische Operatoren

- Logischer UND-Operator &&
- Logischer ODER-Operator ||
- Logischer Negationsoperator !

Wenn Wasser und Sonne oder künstliches_Licht anstelle der Sonne vorhanden sind, dann kann ein Baum wachsen.

```
if ( (Wasser) && ((Sonne) || (künstliches_Licht)) )
{
      baumWachse();
}
```

Damit baumWachse() ausgeführt wird, muss (Wasser == true) sein und gleichzeitig entweder Sonne oder künstliches_Licht. Auch hier werden die Einzelbedingungen mit Klammern gegliedert. Einmal sind Sonne und künstliches_Licht zusammengefasst und dann nochmals diese beiden Bedingungen mit Wasser in einem Klammernpaar. Zwischen diesen Bedingungen stehen dann die Operatoren mit denen sie logisch verknüpft werden.

Nun noch etwas zum Einsatz des logischen Negationsoperators.
Folgende Bedingungen sind alle gleichwertig:

1. (Wasser == false)
2. (Wasser != true)
3. (!(Wasser))
4. (!(Wasser == true))
5. (!(Wasser != false))

Alle diese Bedingungen sind true, wenn kein Wasser vorhanden ist. Die einfachste und kürzeste ist hierbei aber die dritte. Das Ausrufezeichen macht also aus if-Anweisung eine so genannte if-nicht-Anweisung (Wenn-nicht-Anweisung).

Beispiele für Bedingungen in C++

Boolesche Werte

(eingabe) oder (eingabe == true)	
(eingabe == false)	true, wenn eingabe false ist

Zeichenketten

(eingabe == "Hallo")	true, wenn eingabe gleich „Hallo" ist.
(eingabe != "Hallo")	true, wenn eingabe nicht „Hallo" ist.

Zahlen

(eingabe > 5)	true, wenn eingabe größer als 5 ist.
(eingabe <= 10)	true, wenn eingabe kleiner oder gleich 10 ist.

Beispiel:

Wurzelberechnung

Der Befehl, mit dem man die Wurzel einer Zahl ausrechnet, heißt sqrt() und ist in der Bibliothek <cmath> hinterlegt.

```
// wurzel.cpp - 15.08.2005 - Anton Trojosky
// Wurzelberechnung

# include <iostream>
# include <cmath>

using namespace std;
```

```
int main()
{
      float eingabe, wurzel;

      cout << "Programm zur Wurzelberechnung" << endl;
      cout << endl;
      cout << "Geben sie bitte eine Zahl ein: ";
      cin >> eingabe;

      if (eingabe >= 0)
      {
            wurzel = sqrt(eingabe);
            cout << "Wurzel von " << eingabe << " = " << wurzel << endl;
      }
      else
      {
            cout << "Du musst eine positive Zahl eingeben." << endl;
      }

      cout << endl;
      cout << "Programm wird beendet." << endl;

      return 0;
}
```

Man kann nur von positiven Zahlen bzw. Zahlen größer oder gleich 0 eine Wurzel ziehen. Dazu wird hier die `if`-Anweisung genutzt. Die Wurzel von der eingegebenen Zahl wird nur gezogen, wenn die Zahl größer oder gleich 0 ist.

6.1.2. Mehrfache Alternative – `else if`

Die `else if`-Anweisung ist die allgemeinste Möglichkeit für eine Mehrfach-Selektion, mit der eine Auswahl unter verschiedenen Alternativen getroffen wird.

6.1.2.1. Allgemeine Syntax

```
if (Ausdruck_1)
{
      Anweisung_1;
}
else if (Ausdruck_2)
{
      Anweisung_2;
}
else if (Anweisung_3)
{
      Anweisung_3;
}
else if (Anweisung_4)
{
      Anweisung_4;
}
...
else
{
      Anweisung_else;
}
```

In der angegebenen Reihenfolge wird ein Vergleich nach dem anderen durchgeführt. Bei der ersten Bedingung, die wahr ist, wird die zugehörige Anweisung abgearbeitet und die Mehrfachselektion abgebrochen.

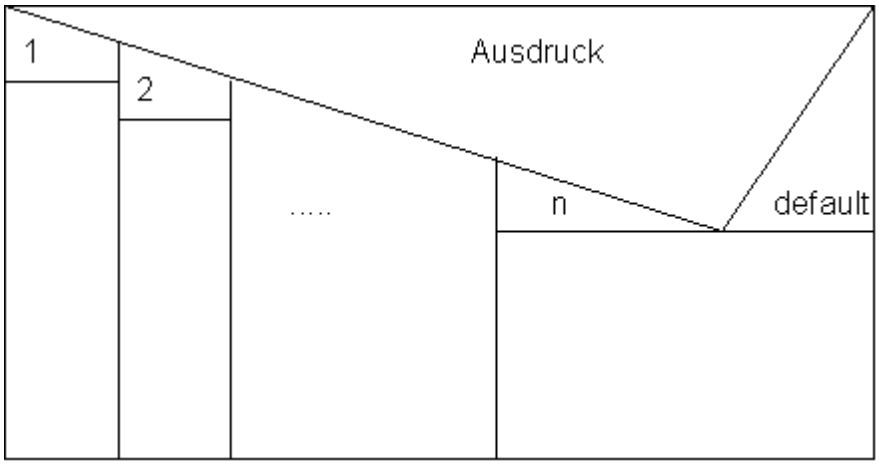
Der letzte else-Zweig ist optional. Hier können alle anderen Fälle behandelt werden, die in der `if-else if`-Kette nicht explizit aufgeführt sind.

6.1.3. `switch`-Verzweigung

Für eine Mehrfach-Selektion, d.h. eine Selektion unter mehreren Alternativen, kann auch die `switch`-Anweisung verwendet werden.

6.1.3.1. Allgemeine Syntax

```
switch (Ausdruck)
{
      case Konstante1:
                  Anweisung(en)1;
                  break;
      case Konstante2:
                  Anweisung(en)2;
                  break;
      case Konstante3:
                  Anweisung(en)3;
                  break;
      case Konstante4:
                  Anweisung(en)4;
                  break;
      ...
      default:
                  Anweisung(en)_default;
}
```



Das Ganze könnte auch durch mehrere `if`-Verzweigungen ersetzt werden. Allerdings ist es so viel übersichtlicher.

Der Ausdruck, der im Kopf der Anweisung steht, wird mit den Konstanten 1 bis 4 verglichen. Wenn `(Ausdruck == Konstante)` erfüllt ist, wird die entsprechende Anweisung, die mit einem Doppelpunkt auf die jeweilige Konstante folgt, ausgeführt.

Eine wichtige Bedingung für die `switch`-Anweisung ist, dass – eigentlich selbstverständlich – alle case-Marken unterschiedlich sein müssen. Vor einer einzelnen Befehlsfolge können jedoch auch mehrere verschiedene `case`-Marken stehen.

```
// Switch.cpp – 18.08.2005 – Anton Trojosky
// Verdeutlichung von switch

#include <iostream>

using namespace std;

int main ()
{
      int zahl;

      cout << "Eingabe: ";
      cin >> zahl;

      switch (zahl)
      {
            case 2:
            case 4:
                  cout << "Es war eine gerade Zahl zwischen 1 und 5";
                  break;
            case 1:
            case 3:
            case 5:
                  cout << "Es war eine ungerade Zahl zwischen 1 und 5";
                  break;
            default:
                  cout << "Es war keine Zahl zwischen 1 und 5";
      }
      cout << endl;
      return 0;
}
```

Wird durch die `switch`-Anweisung eine passende `case`-Marke gefunden, werden die anschließenden Anweisungen bis zum `break` ausgeführt. `break` springt dann an das Ende der `switch`-Anweisung.

Die Anweisung kann auch mehrere Zeilen haben. Allerdings werden diese nicht, wie bei einer `if`-Anweisung, mit geschweiften Klammern in Blöcke zusammengefasst.

Mit `break` wird die Schleife abgebrochen. Wenn die `break`-Anweisung fehlt, werden die nach der nächsten `case`-Marke folgenden Anweisungen abgearbeitet. Dies geht so lange weiter, bis ein `break` gefunden wird oder bis das Ende der `switch`-Anweisung erreicht ist.

Statt umfangreicher `else if`-Konstruktionen sollte – falls möglich – bevorzugt die mehrfache Alternative `switch` benutzt werden.

6.2. Schleifen

Schleifen dienen dazu, einen Anweisungsblock mehrfach hintereinander auszuführen.

6.2.1. `for`-Schleife

6.2.1.1. Allgemeine Syntax

```
for (Initialisierung; Bedingung; Veränderung)
{
      Anweisung(en);
}
```

`Initialisierung`: Der Code wird beim ersten Eintritt in die Schleife ausgeführt. Bei den nachfolgenden Schleifendurchgängen wird die Anweisung nicht mehr ausgeführt. Hier wird der Laufvariablen ein Startwert zugewiesen.

`Bedingung`: Solange die Bedingung erfüllt (`true`) ist, wird die Schleife ausgeführt, andernfalls wird sie abgebrochen.

`Veränderung`: Die Anweisung dieses Teils wird nach jedem Schleifendurchlauf bzw. vor der neuerlichen Prüfung der Schleifenbedingung ausgeführt.

Diese drei Teile bilden den Schleifenkopf. Sie enthalten den Code, der festlegt, wie oft die Schleife ausgeführt wird.

Beispiel:

```
int zahl = 0;

for (int loop = 0; loop <= 5; ++loop)
{
      zahl += loop;
}

cout << zahl;
```

Als erstes wird die Variable `zahl` definiert und gleich mit dem Wert 0 initialisiert.

Beim Eintritt in die Schleife wird die Laufvariable `loop` definiert und mit dem Wert 0 initialisiert. Dann wird die Schleifenbedingung überprüft. Da sie erfüllt ist (`loop` ist kleiner als 5) wird der Anweisungsblock ausgeführt. Zu der Variable Zahl wird `loop` addiert und das Ergebnis in `zahl` gespeichert. Nach Ausführung des Anweisungsblocks wird der dritte Teil des Schleifenkopfs, `++loop`, ausgeführt.

Die weitere Ausführung der Schleife folgt immer dem gleichen Schema:

1. Die Schleifenbedingung wird überprüft.
2. Wenn die Bedingung erfüllt ist, wird der Anweisungsblock der Schleife ausgeführt.
3. Nach Abarbeitung des Anweisungsblocks wird die Laufvariable `loop` inkrementiert (um 1 erhöht).

So nimmt `loop` nacheinander die Werte 0, 1, 2, 3, 4, 5 und 6 an, und in `zahl` werden nacheinander die Werte 1, 3, 6, 10, 15 gespeichert.

Nach dem fünften Durchlauf der Schleife, wird die Laufvariable `loop` ein weiteres Mal inkrementiert. Sie hat jetzt der Wert 6. Wenn nun die Bedingung (`loop <= 5`) überprüft wird, ist sie nicht mehr wahr (`false`). Die Schleife wird nun verlassen und das Programm wird fortgesetzt.

Man kann die Laufvariable auch dekrementieren (um 1 verringern).

```
int zahl = 0;

for (int loop = 5; loop >= 0; --loop)
{
      zahl += loop;
}

cout << zahl;
```

Die Variable wird mit 5 initialisiert und jedes Mal um 1 dekrementiert. Die Schleife wird durchlaufen, so lange `loop` größer oder gleich 0 ist. `Zahl` hat die Werte 5, 9, 12, 14, 15.

Weitere Beispiele:

```
// BeispieleFor.cpp - Anton Trojosky - 18.08.2005
// Verdeutlichung von for

#include <iostream>

using namespace std;

int main ()
{
      for (int loop=12; loop >= 0; loop = loop - 2)
      {
            cout << loop;
            if (loop > 0)
            {
                  cout << ", ";
            }
      }
      cout << endl;
      for (int loop=-1; loop >= -13; loop = loop - 2)
      {
            cout << loop;
            if (loop > -12)
            {
                  cout << ", ";
            }
      }
      cout << endl;
      for (loop = 0; loop <= 6; ++loop)
      {
            cout << loop*loop;
            if (loop * loop < 36)
            {
                  cout << ", ";
            }
      }
      cout << endl;

      return 0;
}
```

Die Ausgabe am Bildschirm ist:

12, 10, 8, 6, 4 , 2, 0
-1, -3, -5, -7, -9, -11, -13
0, 1, 4, 9, 16, 25, 36

6.2.2. `while`-Schleife

Die zweite wichtige Schleifenvariante ist die `while`-Scheife. Bei dieser Schleife sind `Initialisierung`, `Bedingung` und `Veränderung` nicht im Schleifenkopf zusammengefasst, sondern werden über den Code verteilt. Sie wird oft verwendet, wenn der Programmierer nicht genau weiß, wie oft die Schleife durchlaufen werden soll.

6.2.2.1. Allgemeine Syntax

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

Als erstes sollten die verwendeten Variablen initialisiert werden. Dann wird die Bedingung der Schleife überprüft. Wenn sie `true` ist, also zutrifft, werden die Anweisungen ausgeführt, sonst werden diese übersprungen.

Beispiel:

Der Benutzer gibt so lange Zahlen auf der Tastatur ein, bis die Summe dieser Zahlen 100 übersteigt.

```
// summe.cpp - 15.08.2005 - Anton Trojosky
// Eingabeabfrage und Berechnung der Summe daraus

#include <iostream>

using namespace std;

int main()
{
      int eingabe=100, summe=0;

      cout << "Summenprogramm" << endl;
      cout << "Bitte geben sie Zahlen ein, "
            << "bis die Summe groesser als 100 ist." << endl;

      while (summe <= 100)
      {
            cin >> eingabe;
            summe += eingabe;
            cout << "Die Summe ist " << summe << endl;
      }
      cout << "Programm beenden" << endl;

      return 0;
}
```

6.2.3. `do-while` Schleife

6.2.3.1. Allgemeine Syntax

```
Initialisierung;
do
{
      Anweisung(en);
} while (Bedingung);
```

Der einzige Unterschied zur While-Schleife besteht darin, dass die Bedingung erst überprüft wird, nachdem die Schleife schon einmal durchlaufen wurde. Sonst sind die beiden Schleifen identisch.

Beispiel:

Wenn die Summe in unserem letzten Beispiel 100 überstiegen hat, wird das Programm beendet. Was ist aber, wenn der Benutzer das Programm nochmals durchlaufen lassen will bzw. noch mal eine neue Summe von neuen Zahlen berechnen will?

Dann müsste man ihn danach fragen. Also erweitern wir hier unser Beispiel. Am Ende wird eine Abfrage eingeblendet, bei der der Benutzer wählen kann, ob er noch mal eine Summe berechnen oder das Programm beendet will.

```
// summe2.cpp - 15.08.2005 - Anton Trojosky
// Eingabeabfrage und Berechnung der Summe daraus

#include <iostream>

using namespace std;

int main()
{
      int eingabe=100, summe=0;

      cout << "Summenprogramm" << endl;
      do
      {
            cout << "Bitte geben sie Zahlen ein, "
                  << "bis die Summe groesser als 100 ist." << endl;

            while (summe <= 100)
            {
                  cin >> eingabe;
                  summe += eingabe;
                  cout << "Die Summe ist " << summe << endl;
            }

            summe = 0;
            cout << endl;
            cout << "(Menu)Was wollen sie tun?" << endl;
            cout << "Summe berechnen:     1" << endl;
            cout << "Programm beenden:    0" << endl;
            cin >> eingabe;

      } while (eingabe != 0);

      return 0;
}
```

Ein weiteres Beispiel:

```
//Genauigkeit.cpp - 26.08.2005 - Anton Trojosky
//Genauigkeit des Datentyps float berechnen

#include <iostream>

using namespace std;

int main ()
{
      int zahl = 1;
      float a = 1.0f, a_alt, teiler = 2.0f, s;

      do
      {
            a_alt = a;
            a /= teiler;
            s = zahl + a;
      } while (s > zahl);

      cout << a_alt << endl;

      return 0;
}
```

Die ausgegebene Genauigkeit mit dem Teiler 2 beträgt:
1.19209e-7 (entspricht: 0.000 000 119 209)

Die Ausgabe ist aber nicht die exakte Genauigkeit. Da immer durch 2 geteilt wird, sind die Schritte bei denen verglichen wird noch recht groß. Wenn wir die Variable `teiler` nun auf 1.5 setzen ist die ausgegebene Genauigkeit schon besser:
6.02918e-8 (entspricht: 0.000 000 060 291 8)

Ganz genau wird es, wenn man die Variable Teiler immer kleiner macht. Sie darf aber nicht kleiner werden als die Genauigkeit selbst, weil sonst nichts mehr verändert wird. Die wirkliche Genauigkeit beträgt:
5.96047e-8 (entspricht: 0.000 000 059 604 7)

7. Praktikum 2

7.1. Aufgabe 1 - Größe

Schreibe ein Programm, das von zwei Benutzern die Namen und die zugehörige Körpergröße in cm einliest. Dann soll es ausgeben, wer von den beiden größer ist.

7.2. Aufgabe 2 - Umwandlung

Schreiben sie ein Programm, das eine Zahl von 0 bis 5 einliest und als ausgeschriebene Zahl wieder ausgibt. (z.B. Eingabe: 5; Ausgabe: „Fuenf") Verwende dazu als erstes `else-if` und dann im zweiten Schritt `switch`. Welche Mehrfach-Selektion ist sinnvoller.
Als Erweiterung kann man den Benutzer entscheiden lassen, ob er diese Umwandlung einmal oder öfter durchführen will. (`while` oder `do-while`)

7.3. Aufgabe 3 - Quadratzahlen

Berechnung der ersten zehn Quadratzahlen mit einer `for`-Schleife. (1, 4, 9, ...)

7.4. Aufgabe 4 – Vierstellige Zahlen

Wenn man eine vierstellige Zahl hat und sie in zwei Zahlen zerlegt, indem man die vorderen beiden und die hinteren beiden Stellen wegnimmt, kann man diese beiden Zahlen quadrieren und wieder addieren. Nun gibt es einige wenige Zahlen, bei denen diese Summe gleich der Zahl aus der sie berechnet wurde ist. (z.B. 1233 = 12*12 + 33*33)
Schreiben sie ein Programm, das mit Hilfe einer `for`-Schleife weitere Zahlen berechnet, bei denen dies zutrifft.

7.5. Aufgabe 5 - Schaltjahrberechnung

Von Papst Gregor XIII wurde 1582 folgende Regelung eingeführt (Gregorianischer Kalender):

- Glatt durch 4 teilbare Jahre sind Schaltjahre
- Glatt durch 100 teilbare Jahre sind keine Schaltjahre
- Glatt durch 400 teilbare Jahre sind aber wieder Schaltjahre

Schreibe ein Programm, dass überprüft ob ein eingegebenes Jahr ein Schaltjahr ist. Beachte, dass diese Berechnung nur für die Jahre nach 1582 gilt.

Tipp: `if`-Verzweigungen können auch verschachtelt verwendet werden. Das heißt man benutzt eine oder mehrere weitere `if`-Verzweigungen innerhalb einer anderen `if`-Verzweigung.

7.6. Aufgabe 6 - Wochentagsberechnung

Zellers Kongruenz ist der mathematische Weg, um den Wochentag eines gegebenen Datums zu ermitteln. Dieser Weg wurde von dem Geistlichen Christian Zeller 1882 in einer Formel zusammengefasst. Diese Vorgehensweise wird oft in der Programmierung verwendet und trägt auch häufig den Namen *Ewiger Kalender*.

Die Formel, um einen Wochentag im Gregorianischen Kalender zu einem gegebenen Datum zu ermitteln, lautet:

$$Wt = \left(T + \left\lfloor \frac{(M+1)*26}{10} \right\rfloor + J + \left\lfloor \frac{J}{4} \right\rfloor + \left\lfloor \frac{Jh}{4} \right\rfloor - 2*Jh \right) \bmod 7$$

Jh	Jahrhundert (die ersten beiden Stellen des Jahres)
J	Jahreszahl innerhalb des Jahrhunderts
M	Monat Die Monate Januar und Februar werden als 13. bzw. 14. Monat des Vorjahres betrachtet. Das Jahr *J* ist dann auch um Eins zu reduzieren.
T	Tag
Wt	Wochentag (1 = Sonntag, 2 = Montag, 3 = Dienstag, 4 = Mittwoch, 5 = Donnerstag, 6 = Freitag, 0 = Samstag) *Möchte man, wie heutzutage üblich, dass die Woche mit dem Montag beginnt, so muss man einfach Eins von der Formel subtrahieren.*

Das mod 7 (ausgesprochen Modulo 7) am Ende bedeutet, dass der ermittelte Wert durch 7 geteilt und der Rest, der bei dieser ganzahligen Division durch 7 übrig bleibt, bestimmt wird.

Ist das Ergebnis negativ (je nach verwendeter Modulo-Funktion), so addiert man 7 hinzu, so dass eine positive Zahl entsteht. Diese Zahl entspricht dann dem Wochentag.

Beispiel

Nehmen wir an, der Tag wäre der 9. November 1989. Für den Monat November ist *M* = 11 in die Formel einzusetzen. Der Tag *T* = 9 und *J* = 89. Beim Wert für *Jh* werden nur die ersten beiden Ziffern des Jahres verwendet, also 19 für 1989.

Die Berechnung würde sich also folgendermaßen darstellen:

$$Wt = (9 + \lfloor((11+1)*26)/10\rfloor + 89 + \lfloor 89/4\rfloor + \lfloor 19/4\rfloor - 2*19) \bmod 7$$

$$Wt = (9 + 31 + 89 + 22 + 4 - 38) \bmod 7$$

$$Wt = (117) \bmod 7 = 5$$

Im gegebenen Beispiel ist dieser Restwert 5. Dieser Restwert repräsentiert dann den Wochentag. In unserem Beispiel also den Donnerstag. Der Tag des Mauerfalls war an einem Donnerstag.

Aufgabe

Schreiben sie ein Programm, das ein beliebiges Datum einliest und den Wochentag ausgibt. Tipp: Beim Einlesen des Datums sollten Tag, Monat, Jahr einzeln in Variablen vom Typ `int` gespeichert werden. Das Jahr kann dann mit dem Modulo(%)- und dem Divisions(/)-Operator in Jahrhundert und Jahreszahl zerlegt werden.

7.7. Aufgabe 7 – Berechnung von Pi

Die Kreiszahl Pi kann man auf folgende Art und Weise berechnen:

Pi = (1 – 1/3 + 1/5 – 1/7 + 1/9 – 1/11 + 1/13 - ...) * 4

Schreiben Sie ein Programm, das nach dieser Formel Pi berechnet. Fragen Sie beim Programmstart, wie viele Schritte durchgeführt werden sollen.
Achtung: Beim Rechnen mit Kommazahlen müssen Konstanten mit Punkt und einem f dahinter, wenn es sich um ein float handelt, geschrieben werden. Also 1.0f und nicht 1, sonst werden sie als `int` interpretiert und die Nachkommastellen gehen verloren.

8. Arrays

8.1. Was ist ein Array?

Im praktischen Einsatz reicht es meist nicht aus nur einzelne Variable zu speichern, sondern es müssen sehr viele Variable erstellt werden. Mit den bekannten Mitteln wäre es möglich, dies über eine einfache Variable zu lösen. Nehmen wir an, man bräuchte 100 `int`-Variablen. Diese explizit anzulegen, wäre eine mühselige und stupide Aufgabe:

```
int i1;
int i2;
int i3;
.....
int i100;
```

Stattdessen bietet C++ den Datentyp Feld (oder Englisch Array) an. Ein Feld kann dabei sehr viele Elemente speichern, wobei alle vom gleichen Typ sein müssen. Die Anzahl der Elemente muss beim Erstellen des Feldes bekannt sein. Es ist also nicht möglich später noch weitere Elemente anzufügen.

8.2. Ein Array definieren

Um ein Feld anzulegen, verwendet man in C++ eine ähnliche Syntax zu jener, die zur Definition von Variablen genutzt wird:

```
TYPNAME ARRAYNAME[GROESSE];
```

Wobei Typname der Typ der einzelnen Elemente und `GROESSE` die Anzahl der Elemente in diesem Array ist. Die Anzahl der Elemente kann nur eine ganze, positive Zahl sein.

```
int alpha[5];           //Array aus 5 Elementen von Typ int
char beta[10];          //Array aus 10 Elementen von Typ char
```

Die Elemente des Arrays `alpha` werden vom Kompiler direkt hintereinander anlegt. Man kann sich das Array also wie folgt vorstellen:

1	2	3	4	5
int	int	int	int	int

Der Kompiler erkennt Arrays an den eckigen Klammern.

8.3. Auf Elemente eines Arrays zugreifen

Der Zugriff auf die einzelnen Elemente erfolgt über den so genannten Array-Index. Syntax hierbei ist:

```
FELDNAME[INDEX]
```

Für den Wert des Index gilt es zu beachten:

- Wenn ein Feld mit der Größe n angelegt wird, so kann der Index die Werte 0 bis (n-1) annehmen. Wenn also ein Feld mit der Größe 5 angelegt wird, so kann der Index Werte von 0 bis 4 annehmen.
- C++ verhindert nicht, das auf Elemente zugegriffen wird, die nicht existieren. Wenn man also ein Array mit der Größe 5 anlegt, verhindert C++ den Zugriff auf das 6. Element nicht. Jede Schreib- und Leseoperation auf nicht existente Elemente hat jedoch kein voraussagbares Ergebnis und sollte deshalb vermieden werden.

Der Index kann sowohl eine feste Zahl als auch eine Variable sein. Dies ist auch einer der Vorteile von Feldern gegenüber einzelnen Variablen, denn auf diese Art können tausende von Variablen einfach mit einer Schleife geändert werden.

Dieses Beispiel zeigt ein Array mit 5 `int` Zahlen:

```
int alpha[5];     //Array definieren

alpha[0] = 5;     //das 1.Element mit dem Index 0 auf den Wert 5 setzten
alpha[1] = 14;    //das 2.Element mit dem Index 1 auf den Wert 14 setzten
alpha[2] = 24;    //das 3.Element mit dem Index 2 auf den Wert 24 setzten
alpha[3] = 63;    //das 4.Element mit dem Index 3 auf den Wert 63 setzten
alpha[4] = 8;     //das letzte Element mit dem Index 4 auf den Wert 8 setzten
```

alpha[0] int	alpha[1] int	alpha[2] int	alpha[3] int	alpha[4] int
5	14	24	63	8

8.4. Ausgabe aller Elemente eins Arrays

Möchte man nun alle Elemente des oben angelegten Arrays auf den Bildschirm ausgeben, verwendet man eine for-Schleife. Diese muss die Werte 0 bis 4 annehmen und das jeweilige Element ausgeben.

```
/* .... Code von oben ....*/
for (int index = 0; index<5; ++index)
{
      cout << "Das Element Nr.  " << index << " hat den Wert "
            << alpha[index] << endl;
}
```

Hier zeigt sich auch der Vorteil eines Arrays, denn man kann auf viele Elemente über eine Variable zugreifen. Dies ist mit einer einzelnen Variable nicht möglich.

8.5. Zusammenfassung Arrays

- Arrays dienen zum Speichern von vielen Variablen vom gleichen Typ
- Anlegen von Arrays: `elemententyp arrayname[anzahlElemente]` (Beispiel: `int zahlen[10]`)
- Zugriff auf Elemente eines Arrays: `arrayname[index]` (Beispiel: `zahlen[3]`)
- Index kann Werte zwischen 0 und `anzahlElemente` minus eins annehmen

8.6. Sortieren von Arrays

Eine Herausforderung beim Programmieren stellt das Sortieren von vielen Zahlen da. Hier hat sich eine Wissenschaft gebildet, die immer auf der Suche nach der schnellsten Methode zum Sortieren von vielen Zahlen ist. Je schneller die Methoden werden, desto komplexer werden die Vorgehensweisen. Wir werden jetzt eine Methode durchsprechen, die die einfachste aber gleichzeitig auch die langsamste ist.
Um die Zahlen sortieren zu können, werden diese in einem Array gespeichert. Wir testen unsere Methode an Ganzzahlen (`int`). Sie ist aber auch einfach auf Kommazahlen zu übertragen. Die Sortiermethode heißt „Bubble-Sort" (Bubble zu Deutsch: Blase). Wieso werden wir später sehen.

Wir wollen die Methode anhand dieses Beispiels testen:

1	*2*	*3*	*4*	*5*
5	2	3	1	4

Dieses Arrays soll aufsteigend sortiert werden. Also so, dass das kleinste Element am Anfang und das größte am Ende steht. Danach soll das Array also so aussehen:

1	*2*	*3*	*4*	*5*
1	2	3	4	5

Beim Bubble-Sort werden immer zwei nebeneinander liegende Elemente verglichen. Man vergleicht also als erstes die ersten beiden Elemente. Nun werden die beiden Elemente so vertauscht, dass beim aufsteigenden Sortieren die kleinere Zahl an erster Stelle steht oder beim absteigenden Sortieren an zweiter Stelle.
In unserem Beispiel vergleicht man also 5 und 2 und vertauscht dann die beiden, so dass die 2 an erster und die 5 an zweiter Stelle steht. Das Array sieht dann also so aus:

1	*2*	*3*	*4*	*5*
2	5	3	1	4

Im nächsten Schritt werden die Elemente 2 und 3 verglichen, also die Werte 5 und 3. Auch hier werden die Elemente vertauscht, so dass das Array dann so aussieht:

1	*2*	*3*	*4*	*5*
2	3	5	1	4

Dies wird solange wiederholt bis die letzten beiden Elemente verglichen und evtl. vertauscht wurden. Das Array sieht dann so aus:

1	*2*	*3*	*4*	*5*
2	3	1	4	5

Sobald alle Elemente einmal betrachtet wurden ist der erste Durchlauf abgeschlossen. Das Vergleichen und Vertauschen muss jetzt von neuem beginnen und es müssen wieder die ersten beiden Elemente verglichen werden. Nur der letzte Vergleich kann weggelassen werden, da dieser schon richtig sortiert ist. So werden also im ersten Durchlauf 4 Vergleiche durchgeführt, im zweiten 3, im dritten 2, im vierten 1 und damit ist das Array richtig sortiert.

Die Umsetzung in eine Programmiersprache ist recht einfach: Man benötigt eine `for`-Schleife zum Vergleichen der Elemente und eine weitere `for`-Schleife die diese Durchläufe wiederholt. Wenn ein Array mit x Elementen existiert, müssen x Durchläufe (von 0 bis x-1) gemacht werden, wobei jeder Durchlauf (x-Durchlaufzähler) Vergleichsvorgänge machen muss, also von 0 bis (x-Durchlaufzähler-1). Mit diesen Angaben lassen sich schon zwei `for`-Schleifen konstruieren.

```
// Schleife zum wiederholen der Durchläufe
for (int i=0;i<count-1; ++i) {
      // Schleife für die Durchläufe
      for (int j=0;j<count-i-1; ++j) {
```

Nun muss immer das j. Element des Arrays mit dem Element Nummer (j+1) verglichen werden und bei Bedarf getauscht werden.

Es existiert noch ein Problem beim Tauschen: Es muss eine temporäre Variable benutzt werden, da sonst beide Elemente den gleichen Wert haben. Daraus folgt dann folgender Code:

```
//Schleife zum wiederholen der Durchläufe
for (int i=0; i<count-1; ++i)
{
      //Schleife für die Durchläufe
      for (int j=0; j<count-i-1; ++j)
      {
            //Wenn kleiner, dann tauschen
            if ( z[j] < z[j+1] )
            {
                  int temp = z[j];
                  z[j] = z[j+1];
                  z[j+1] = temp;
            }
      }
}
```

Das Array z hat die Länge count und muss vorher mit Elementen gefüllt werden. Diese können entweder am Anfang festgelegt oder durch Zufallszahlen gesetzt werden.

9. Praktikum 3

9.1. Aufgabe 1 – Array - Mittelwert

Lesen Sie 10 Zahlen vom Benutzer ein. Speichern Sie diese in einem Array. Gebe Sie dann alle 10 Zahlen aus und berechen Sie den Mittelwert.

9.2. Aufgabe 2 – Array - Index

Lesen Sie 10 Zahlen vom Benutzer ein, speichern Sie diese in einem Array. Fragen Sie nun den Benutzer nach einem Index und geben Sie die entsprechende Zahl aus. Wiederholen Sie dies bis ein ungültiger Index angefordert wird.

9.3. Aufgabe 3 – Sortieren von Arrays

Erstellen Sie ein Programm das 5 vom Programmierer festgelegte Zahlen sortiert und jeden Schritt während der Durchläufe ausgibt.

9.4. Aufgabe 4 – Sortieren von Arrays

Schreiben Sie ein Programm das ein Array mit 100 Zufallszahlen erstellt und ausgibt und dieses dann absteigend sortiert und noch einmal ausgibt.
Um Zufallszahlen zu erzeugen zu können, muss `<ctime>` eingebunden werden. Der Befehl `rand()` gibt eine Zufallszahl zwischen 0 und 32767 zurück. Vorher muss jedoch der Zufallsgenerator einmal mit folgendem Befehl initialisiert werden:

```
srand((unsigned)time(NULL));
```

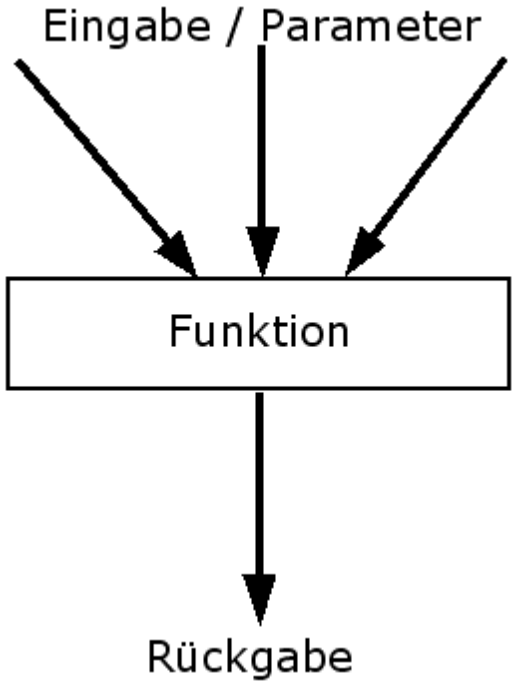
10. Funktionen

10.1. Wozu dienen Funktionen?

Oft werden bestimmte Operationen mehrfach benötigt. Eine Lösung für dieses Problem wäre es, die Code-Zeilen einfach zu kopieren. Dies führt jedoch zu Problemen, wenn später der Code geändert werden soll. Außerdem bläht es den Code unnötig auf. Deshalb bietet C++ (wie die meisten anderen Programmiersprachen auch) die Möglichkeit, Code in so genannte Funktionen auszulagern. Eine Funktion wird dabei einmal definiert und kann dann an jeder Stelle im Code aufgerufen werden. Soll nun die Funktionsweise geändert werden, so muss man nur noch die Funktion ändern und nicht stattdessen den gleichen Code sehr oft umändern.

10.2. Eigenschaften einer Funktion

Eine Funktion kann man sich am einfachsten mit folgendem Diagramm vorstellen:



Jede Funktion hat einen Namen mit dem sie aufgerufen werden kann.
Funktionsnamen werden dabei nach den gleichen Regeln wie Variablennamen gewählt. Der Name sollte dabei Auskunft geben, was die Funktion tut.

Außer einem Namen benötigt man zur Deklaration einer Funktion keine weiteren Angaben. Um allerdings eine sinnvolle Funktion zu erzeugen, ist es meistens sinnvoll entweder so genannte Parameter oder einen Rückgabe-Wert festzulegen, oder sogar beides.

Parameter sind Werte, die der Funktion bei jedem Aufruf zur Verwendung übergeben werden. Eine Funktion kann keine, einen oder mehrere Parameter haben. Es müssen bei der Deklaration die Anzahl und der Typ der Parameter festgelegt werden und diese müssen dann auch bei jedem Aufruf übergeben werden. Da mehrere Parameter vorhanden sein können, muss für jeden Parameter sowohl ein Name zur Identifizierung als auch der Typ festgelegt werden.

Eine Funktion kann einen Wert zurückgeben, welcher dann nach dem Aufruf weiterverwendet werden kann. Eine Funktion kann keinen oder einen Rückgabe-Wert besitzen. Auch für den Rückgabe-Wert muss bei der Deklaration ein Typ festgelegt werden. Wenn ein Rückgabe-Wert festgelegt wird, dann muss dieser auf jeden Fall in der Funktion zurückgegeben werden. Er wird also zur Pflicht.

10.3. Definition von Funktionen in C++

```
[rückgabetyp] [funktionsname](parameterliste)
{
      Anweisung(en);

      return [rückgabewert];
}
```

Rückgabetyp:
Gibt den Typ des Rückgabe-Wertes an, so zum Beispiel `int`, `string` oder `bool`.

Funktionsname:
Name der Funktion, unter dem sie später aufgerufen wird. Regeln für den Namen sind identisch mit denen für Variablen, also nur Buchstaben und Zahlen, wobei Groß- und Kleinschreibung wichtig sind, sie müssen bei der Definition und dem Aufruf identisch sein.

Parameterliste:
Enthält für jeden Parameter ein Paar aus Variablen-Typ und Name. Jedes dieser Paare wird durch Komma getrennt. Die Parameterliste hat also immer folgendes Format: „`parametertyp parametername, parametertyp parametername`" (siehe Beispiele weiter unten)

Anweisungen:
Hier stehen alle Befehle, die abgearbeitet werden sollen, wenn die Funktion aufgerufen wird.

Return:
Gibt einen Wert zurück und beendet die Funktion. Return steht also immer dann als Befehl, wenn die Abarbeitung der Funktion abgeschlossen werden soll und ein Rückgabe-Wert verwendet werden soll.

Wenn kein Rückgabe-Wert verwendet werden soll, steht statt der Rückgabetyp-Angabe das Wort „`void`", zu Deutsch „leer".

Dasselbe gilt für die Parameterliste: Wenn keine Parameter übergeben werden sollen, wird anstatt der Parameterliste auch hier das Wort void benutzt oder man lässt die Klammern ganz leer. Wir bevorzugen hier die Schreibweise mit einem void in der Klammer.

Stellt sich nur noch die Frage wohin man den Funktions-Code schreibt. Da die Funktion dem Kompiler vor dem Aufruf bekannt sein muss, ist es notwendig, den Code der Funktion oberhalb des Aufrufbefehls zu setzten. In den meisten Fällen ist es sinnvoll den Funktionscode oberhalb der `main()`-Funktion zu schreiben.

Betrachten wir also die einfachste Funktion, eine mit keinem Parameter und keinem Rückgabe-Wert. Wie oben bereits angegeben, kommt dabei zweimal das Wort void zum Einsatz:

```
void unsereErsteFunktion(void)
{
      Anweisung(en);
}
```

Der Befehl return kann weggelassen werden, da kein Rückgabe-Wert zurückgegeben werden muss.

Mit diesen Angaben können wir eine erste Funktion schreiben, sie soll bei jedem Aufruf einen einfachen Text ausgeben. Diese Funktion benötigt weder einen Parameter, noch einen Rückgabewert, weshalb wir den Aufbau von oben übernehmen können:

```
// FunktionBeispiel1.cpp – 18.07.2005 – Dominik Bruhn
// Simple Funktion die eine Nachricht auf den Bildschirm ausgibt

#include <iostream>

using namespace std;

void ausgeben(void)
{
      cout << "Das hier ist unsere Funktion" << endl;
}

int main()
{
      ausgeben();       //Aufruf der Funktion
      return 0;
}
```

Bei jedem Aufruf dieses Programms wird die Funktion `ausgeben` aufgerufen. Diese gibt dann den Text auf den Bildschirm aus.

Die nächste Funktion soll eine übergebene Zahl mit 2 multiplizieren und diese dann zurückgeben. Die Funktion hat also einen Parameter vom Typ `int` und hat einen Rückgabe-Wert vom Typ `int`.

Der Funktionskopf für diese Funktion heißt:

```
int malZwei(int zahl)
```

Innerhalb der Funktion hat `zahl` den Wert des übergebenen Parameters. Wird die Funktion also im Hauptprogramm mit `malZwei(4);` aufgerufen, dann hat „`zahl`“ den Wert 4. Innerhalb der Funktion erstellt man eine neue Variable in der man `zahl` mal zwei abspeichert. Diese Variable muss zurückgegeben werden, deshalb schreibt man:

```
// FunktionBeispiel2.cpp - 16.08.2005 - Dominik Bruhn
// Eine Funktion, die eine Zahl mit 2 multipliziert und deren Aufruf

#include <iostream>

using namespace std;

int malZwei(int zahl)
{
      int i = zahl * 2;
      return i;
}

int main()
{
      int erg = malZwei(25);
      cout << "Ergebnis der Funktion ist " << erg << endl;

      return 0;
}
```

Die Funktion gibt hierbei einen Wert vom Typ `int` zurück. Das Ergebnis der Funktion kann wie ein Wert behandelt werden, es kann also in einer weiteren Variable gespeichert werden oder mit Operatoren verarbeitet werden.

Nun möchten wir die Funktion so erweitern, dass anstatt die Zahlen immer mit 2 zu multiplizieren, diese mit einer zweiten übergebenen Zahl multipliziert werden. Auch die zweite Zahl soll vom Typ `int` sein, deshalb lautet der Funktionskopf:

```
int mal (int zahl1, int zahl2)
```

Der weitere Aufbau ist fast identisch zum obigen, außer das statt `*2` jetzt `*zahl2` stehen muss und der Funktion ein zweiter Parameter übergeben wird.

```
// FunktionBeispiel3.cpp - 16.08.2005 - Dominik Bruhn
// Funktion, die zwei Zahlen multipliziert und deren Ergebnis zurückgibt

#include <iostream>

using namespace std;

int mal(int zahl1, int zahl2)
{
      int i = zahl1 * zahl2;
      return i;
}

int main()
{
      int erg = mal(25,3);
      cout << "Ergebnis der Funktion ist " << erg << endl;

      return 0;
}
```

10.4. Parameter sind Kopien

Parameter, die einer Funktion beim Aufruf übergeben werden, sind immer Kopien.
Folgendes Programm funktioniert deshalb nicht:

```
// FunktionBeispiel6.cpp - 16.08.2005 - Dominik Bruhn
// Zeigt, dass Parameter Kopien sind

#include <iostream>

using namespace std;

void aendern(int zahl)
{
      zahl=zahl*2;
}

int main()
{
      int test=5;
      cout << "Zahl vorher: "<<test<< endl;
      aendern(test);
      cout << "Zahl nachher: "<<test<< endl;

return 0;
}
```

Der Wert der Zahl kann durch die Funktion nicht verändert werden, da der übergebene Wert nur eine Kopie ist und deshalb nur der Wert der Kopie verändert wird, nicht aber der Wert des Originals.

10.5. Default – Parameter

Weiter oben wurde erwähnt, dass alle definierten Parameter der Funktion auch übergeben werden müssen. Es gibt hiervon eine Ausnahme: Es können auch Funktionen definiert werden wo bestimmte Parameter nicht übergeben werden müssen. So könnte man die obige Funktion so verändern, dass Sie folgendes tut:

- Wenn zwei Parameter übergeben werden, so soll sie beide multiplizieren und zurückgeben.
- Wenn nur ein Parameter übergeben wird, so soll sie die Zahl mit 2 multiplizieren und dann zurückgeben.

Der zweite Parameter ist also nicht unbedingt notwendig. Wenn er nicht übergeben wird, soll als Voreinstellung (oder Englisch `default`) die Zahl 2 angenommen werden. Um dies festzulegen muss der Funktionskopf wie folgt aussehen:

```
int mal (int zahl1, int zahl2 = 2)
```

Dies legt fest, dass die Funktion `mal` mit einem oder zwei Parametern aufgerufen werden kann, wobei der zweite Parameter automatisch den Wert 2 annimmt wenn er nicht übergeben wird. Der weitere Code bleibt gleich. Zusätzlich kann die Funktion jetzt im Hauptprogramm noch mit nur einem Parameter aufgerufen werden:

```
// FunktionBeispiel4.cpp - 16.08.2005 - Dominik Bruhn
// Funktion, die zwei Zahlen multipliziert und deren Ergebnis zurückgibt.
// Der Default-Wert für zweiten Parameter ist 2

#include <iostream>

using namespace std;

int mal(int zahl1, int zahl2=2)
{
      int i = zahl1 * zahl2;
      return i;
}

int main()
{
      int erg1 = mal(25,3);
      cout << "Ergebnis der Funktion mit zwei Parameter ist " << erg1
            << endl;

      int erg2 = mal(25);
      cout << "Ergebnis der Funktion mit einem Parameter ist " << erg2
            << endl;

      return 0;
}
```

Es ist zu beachten, dass sobald ein Parameter einen Default-Wert hat, alle darauf folgenden Parameter auch Default-Werte haben müssen. Sollen also zwei Parameter definiert werden, von denen einer einen vorher eingestellten Wert hat (also ein Defaultparameter), so muss der Defaultparameter immer an zweiter Stelle stehen und der „normale" Parameter an erster Stelle.

10.6. Funktionen überladen

In C++ existiert die Möglichkeit mehrere Funktionen mit dem gleichen Namen anzulegen, die sich nur durch die Anzahl und den Typ ihrer Parameter unterscheiden. Der Kompiler sucht beim Erstellen des Programms automatisch die passende Funktion heraus und verwendet diese.
Will man zum Beispiel eine Funktion `output()` schreiben, die entweder einen `int` oder einen `string` annimmt und diesen dann auf den Bildschirm ausgibt, dann kann man folgendes programmieren:

```
// FunktionBeispiel5.cpp - 18.08.2005 - Dominik Bruhn
// Zwei überladene Funktionen, die entweder einen String oder einen int auf
// den Bildschirm ausgeben.

#include <iostream>
#include <string>

using namespace std;

void ausgeben(int zahl)
{
      cout << "Ich habe die Zahl " << zahl << " übergeben bekommen " <<
endl;
}

void ausgeben(string zeichen)
{
      cout  << "Ich habe die Zeichenkette '" << zeichen
            << "' übergeben bekommen " << endl;
}

int main()
{
      ausgeben(25);
      ausgeben("Meine Zeichenkette");

      return 0;
}
```

Beim ersten Aufruf wird die erste Funktion verwendet, da der Parameter ein `int` ist. Beim zweiten Aufruf wird die zweite Funktion verwendet da der Parameter ein `string` ist. Es sollte darauf geachtet werden, dass überladene Funktionen immer das gleiche machen.

10.7. Zusammenfassung

- Funktionen dienen zum Auslagern von Code
- Eine Funktion kann Parameter und einen Rückgabe-Wert haben
- Parameter müssen, wenn sie definiert sind, auch benutzt werden (Ausnahme: Default-Werte)
- Definition einer Funktion mit `[rückgabetyp] [funktionsname]([parameterliste])` `(Beispiel: int mal(int zahl1,int zahl2) )`
- Funktionen werden vor dem Hauptprogramm (also vor `int main()`) eingefügt
- Funktionsname nach gleichen Regeln wie Variablennamen

- Rückgabetyp kann jeder Variablentyp sein. Außerdem `void`, wenn kein Wert zurückgegeben werden soll

- Parameterliste enthält für jeden Parameter dessen Definition im Format `[parametertyp] [parametername]` (Beispiel: `int zahl1`). Mehrere Parameter werden durch Kommas getrennt.

- Wenn keine Parameter erwartet werden sollen, enthält die Klammer entweder das Wort void oder ist ganz leer (beides gleichwertig)

- Parameter sind immer Kopien, deshalb kann die Funktion nicht die Werte der übergebenen Variablen verändern.

- Soll ein Parameter nicht Pflicht sein, so kann stattdessen ein Default-Parameter festgelegt werden, der angenommen wird, wenn der Parameter nicht übergeben wird. Wenn ein Parameter einen Default-Wert hat, dann müssen auch alle folgenden Parameter einen Default-Wert haben.

- Rückgabe eines Wertes (beendet die Funktion automatisch) mit `return [wert];` (Beispiel: `return 0;`)

- Aufruf einer Funktion mit `[funktionsname]([Liste der Parameterwerte]);` (Beispiel: `mal(3,4);`)

11. Einführung in Zeiger/Pointer

Wie wir bei den Variablen gelernt haben, sind diese im Arbeitsspeicher hinterlegt. Sie haben eine bestimmte Adresse im Arbeitsspeicher.
Pointer sind Zeiger, die die Adressen der Variablen speichern können und auf ihre Werte zeigen. Sie haben also keinen „echten" Wert, sondern sie speichern nur die Speicheradresse einer bestimmten Variablen.

In C++ dienen zur Verwaltung von Pointern die Operatoren * und &.

- Der Operator & liefert die Adresse einer Variable.
- Der Operator * liefert den Wert der Variable bzw. den Wert auf was der Pointer zeigt.

```
z = &zahl; // z wird die Adresse von zahl übergeben
zahl = *z; // zahl wird der Wert der Variablen übergeben,
           // auf die z zeigt (hier zahl)
```

11.1. Definition eines Pointers

Ein Pointer zeigt immer auf den gleichen Variablentyp, also z.B. `int` oder `string`. Dieser Typ wird bei der Definition festgelegt und kann zu einem späteren Zeitpunkt nicht verändert werden. Ein Pointer auf einen `int` kann also immer nur auf ein `int` zeigen und niemals auf z.B. einen `string`.
In C++ wir der Pointer mit einem * deklariert.

```
[Variablentyp] * [Variablenname];
```

```
int * z;
```

Dieser Pointer z zeigt also auf eine Variable vom Typ `int`.

Dem Pointer kann nun die Adresse einer `int`-Variablen zugewiesen werden.

```
z = &zahl;
```

Und nun kann der Variablen, auf die der Pointer zeigt, ein Wert über den Pointer zugewiesen werden.

```
*z = 18 ;              // => zahl = 18;
```

```
int zahl;
int * z = &zahl;
string testString = "Das ist ein Test";
z = &testString;         // FEHLER: z ist ein Pointer auf ein int
                         // und kann nicht auf einen string zeigen
```

```cpp
// pointer.cpp – 16.8.2005 - Roman Keller
// Operationen mit Pointern

#include <iostream>

using namespace std;

int main()
{
     int * pZahl;
      int zahl1 = 128;
      int zahl2 = 256;

      pZahl = &zahl1;
      cout << *pZahl << " | " << pZahl << endl;

      pZahl = &zahl2;
      cout << *pZahl << " | " << pZahl << endl;

      *pZahl = 512;
      cout << zahl2  << " | " << pZahl << endl;

      return 0;
}
```

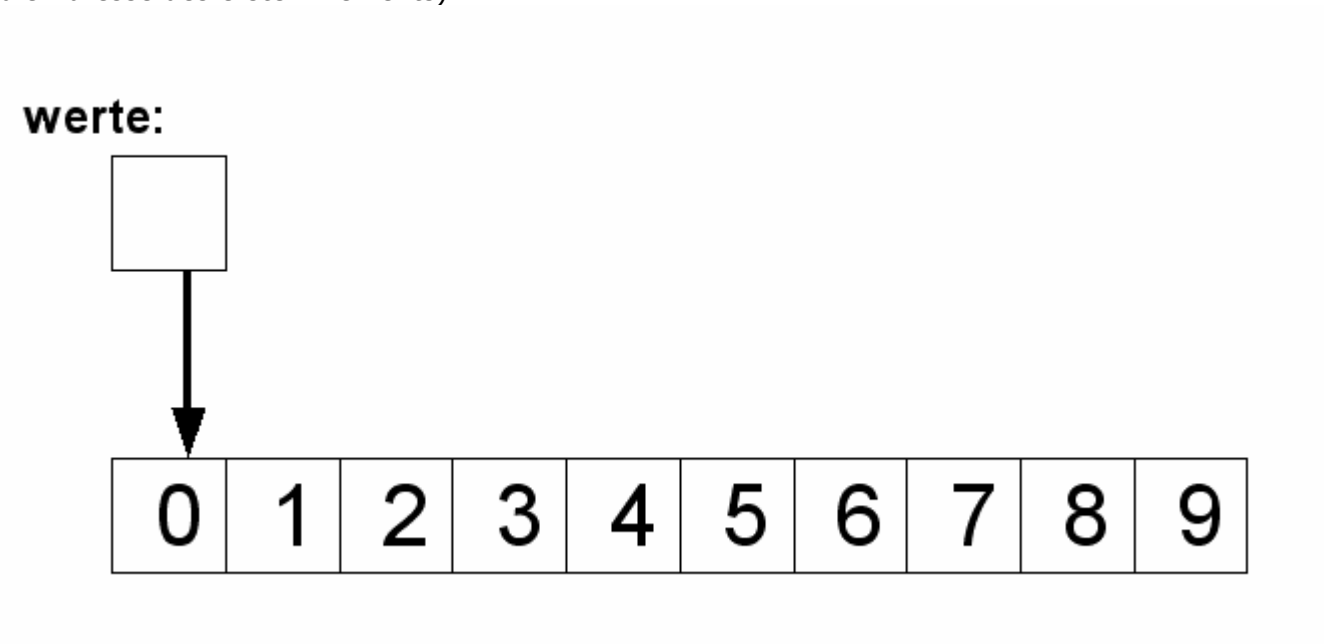
Beispielausgabe:
Beachte, dass die Adressen der Variablen in jedem Programmaufruf variieren können, da nicht feststeht, welcher Platz im Arbeitsspeicher dem Programm zugewiesen wird.

```
128 | 0012FEE4
256 | 0012FEE0
512 | 0012FEE0
```

Als erstes referiert der Pointer `pZahl` auf `zahl1`. Es wird der Wert der Variable ausgeben auf die pZahl referiert. Außerdem wird die Adresse der Variablen, auf die der Pointer zeigt ausgegeben.

11.2. Arrays und Pointer

Mit Pointern kann man auf Arrays zugreifen. Dabei ist zu beachten, dass programmintern eine Array-Variable auch ein Zeiger auf das erste Element eines Arrays ist (Es enthält also die Adresse des ersten Elements).



Insofern kann man auf Felder auch mit Zeigerschreibweise zurückgreifen.
Wenn man die eckigen Klammern weglässt wird eine Arrayvariable als Pointer interpretiert. Mit Klammer kann direkt das Array angesprochen werden bzw. das angegebenen Element.

```
int * pw = werte;       // => pw = &werte[0];
*werte = 88;                  // verändert werte[0] auf 88;
*pw = 89;                     // verändert werte[0] auf 89;
```

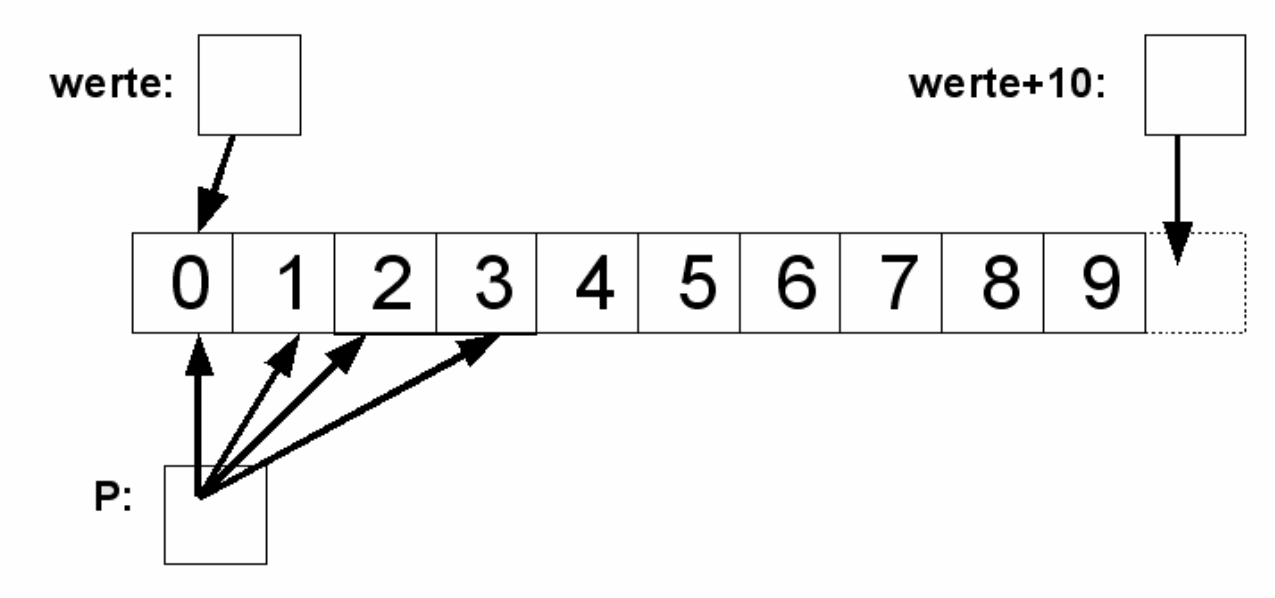
11.3. Pointerarithmetik

Die Analogie zwischen Arrays und Pointer geht sogar so weit, dass man Zeiger über alle Elemente eines Feldes wandern lassen kann. Wenn man den Zeiger ums eins erhöht, dann zeig er auf das nächste Element in Array.

Somit kann man auch auf diese Weise alle Elemente eines Feldes ausgeben.

```
int arOfInt[10];
for (int * p = arOfInt; p < arOfInt+10; ++p)
{
    cout << *p << p << endl;
}
```

Am Anfang zeigt `p` auf `arOfInt[0]`, da `arOfInt` darauf zeigt. Im Verlauf der for-Schleife springt der Pointer immer auf das nächste Element `(++p)`.
Mit cout wird der Wert auf den `p` zeigt `(*p)` und die Adresse, die `p` speichert `(p)`, ausgegeben.



11.4. Zusammenfassung

- Zeiger zeigen auf andere Variablen. Über sie können diese angesprochen werden.
- Der einzige Wert, den ein Zeiger speichert, ist die Speicheradresse der Variablen auf die er zeigt.
- Arrayvariable sind Zeiger auf das erste Element eines Arrays.
- Durch Inkrementierung eines Arraypointers springt dieser auf das nächste Element im Array.

12. Praktikum 4

12.1. Aufgabe 1 – Funktionen - Durchschnitt

Schreiben Sie ein Programm, das zwei Zahlen einliest, diese einer Funktion übergibt, welche dann den Mittelwert zurückgibt.

12.2. Aufgabe 2 – Funktionen - Teiler

Schreiben Sie eine Funktion, die prüft ob der zweite übergebene Parameter ein Teiler des ersten ist. Wenn der Teiler nicht übergeben wird, so soll die Funktion den Wert 2 als gewünschten Teiler annehmen. (Tipp: Benutze den % Operator)

12.3. Aufgabe 3 – Funktionen - Quersumme

Schreiben Sie eine Funktion, die die Quersumme einer übergebenen Zahl ermittelt und zurückgibt. (Tipp: Benutze den % Operator)

13. Objektorientierung

13.1. Was ist ein Objekt

Im Alltag findet man verschiedenste Objekte. Beispiele für Objekte wären ein Telefon, ein Kühlschrank oder auch ein Auto.

Schauen wir uns das Auto näher an. Ein Auto unterscheidet sich von einem Kühlschrank, weil es einfach anders aufgebaut ist und man andere Sachen mit ihm anstellen kann. Wenn man versucht ein Auto zu beschreiben, würde man vielleicht wie folgt anfangen. Ein Auto hat vier Räder, einen Motor und einen Benzintank. Diese Dinge nennt man Eigenschaften. Ein Auto hat noch viel mehr Eigenschaften. Farbe, Anzahl der Sitze, usw.

Die genannten Eigenschaften sind fest. Sie können nicht so einfach geändert werden. Man kann aber auch die aktuelle Geschwindigkeit eines Autos als Eigenschaft bezeichnen. Sie ist im Gegensatz zu den festen Eigenschaften leichter veränderlich. Eine weitere solcher Eigenschaften wäre der Kilometerstand.

Jedes Auto besitzt diese Eigenschaften. Nehmen wir das Beispiel Farbe. Jedes Auto hat eine Farbe, allerdings unterscheiden sich die Farben von vielen Autos. Manche sind grün andere rot und wieder andere vielleicht blau. Jedes Auto hat eine Farbe, aber diese Farben können sehr unterschiedlich sein. So ist es mit allen Eigenschaften. Jedes Auto hat bestimmte Eigenschaften. Die Ausprägungen dieser, machen aber auch gerade den Unterschied verschiedener Autos aus.

Unsere erste Erkenntnis ist also, dass jedes Auto Eigenschaften hat.

Um eine Eigenschaft eines Autos zu ändern, muss man aktiv etwas tun. Wenn man die Farbe ändern will, muss man das Auto lackieren. Wenn man die Eigenschaft Geschwindigkeit ändern will, muss man auf das Gaspedal drücken. Man kann also Eigenschaften nicht direkt verändern, sondern muss immer etwas tun damit sie sich ändern. Dafür stehen in der Programmierung Methoden zur Verfügung.

Unsere zweite Erkenntnis ist, dass man Methoden braucht um Eigenschaften zu ändern.

13.2. Was ist eine Klasse

Unser Auto wurde anhand eines Bauplans gebaut. In diesem Bauplan sind alle Eigenschaften und Methoden beschrieben. Anhand dieses Bauplans wird jetzt aber nicht nur ein Auto gebaut, sondern es werden tausende von Autos produziert, die alle diese Eigenschaften haben. Sie müssen aber nicht alle blau sein. Sie dürfen auch rot, grün, lila usw. sein. Sie müssen aber auf jeden Fall eine Farbe haben. Diesen Bauplan nennt man in der Programmierung Klasse. Von dieser Klasse können dann genau wie bei unseren Autos, viele Objekte mit unterschiedlichen Ausprägungen der Eigenschaften gebaut oder erstellt werden.

13.3. Programmierung

Beim Programmieren braucht man nicht immer alle Eigenschaften und Methoden der realen Objekte zu berücksichtigen. In eine Klasse, die man programmiert, nimmt man nur die Eigenschaften und Methoden auf, die man verwenden will. In unserem Beispiel haben wir uns für die Geschwindigkeit und den Tankinhalt entschieden.

```
class MeinAuto
{
	// Variablen -> Eigenschaften:
	int geschwindigkeit;
	int tankinhalt;

	// Funktionen -> Methoden:
	void starten(void)
	{
		geschwindigkeit = 0;
	}
	void beschleunigen(void)
	{
		++geschwindigkeit;
	}
	void abbremsen(void)
	{
		--geschwindigkeit;
	}
	void tacho(void)
	{
		cout << "Aktuelle Geschwindigkeit: " << geschwindigkeit <<
endl;
	}
};
```

Eine Klassendefinition in C++ wird immer durch das Wort `class` eingeleitet, gefolgt vom Namen der Klasse. Auch für den Namen gelten die Regeln wie für Funktions- und Variablennamen, nur dass man den ersten Buchstaben groß schreibt. Die gesamten weiteren Definitionen werden von zwei geschweiften Klammern umschlossen. Achtung: Hinter der abschließenden geschweiften Klammer muss ein Semikolon stehen.

In der Klasse werden Variablen definiert, die die Eigenschaften der Klassen darstellen. Außerdem werden Funktionen definiert, welche die Methoden der Klassen sind.

Die Deklaration der Klasse muss vor ihrer Benutzung erfolgen. Deshalb wird der Code an der gleichen Stelle, wie für eine Funktionsdefinition geschrieben, d.h. vor die `main()`-Funktion.

Da der Variablen `geschwindigkeit` bei der Definition kein Wert zugewiesen werden kann, (dies ist eine neue Tatsache!) müssen wir dies vor dem ersten benutzen machen, da `int` sonst einem zufälligen Wert hat. Dies erledigt die Methode `starten()`.

13.4. Datenkapselung

Bei einem realen Auto wird die Geschwindigkeit über das Gaspedal und die Bremse geregelt und verändert. Genauso ist das auch bei unserer Klasse bzw. dem Objekt, das daraus entsteht. Die Eigenschaft/Variable `geschwindigkeit` wird mit der Methode/Funktion `beschleunigen()` und `abbremsen()` verändert. Man greift also nie auf die Variable direkt zu. Sie wird nur innerhalb des Objekts von den Methoden angesprochen.

In Klassen gibt es deshalb eine Datenkapselung. Variablen und Funktionen die nur intern genutzt werden, nennt man `private`. Funktionen auf die von extern zugegriffen werden sollen, werden als `public` bezeichnet. Unsere Klasse würde dann also so aussehen:

```
class MeinAuto
{
private:
	// Variablen -> Eigenschaften:
	int geschwindigkeit;
	int tankinhalt;

public:
	// Funktionen -> Methoden:
	void starten(void)
	{
		geschwindigkeit = 0;
	}
	void beschleunigen(void)
	{
		++geschwindigkeit;
	}
	void abbremsen(void)
	{
		--geschwindigkeit;
	}
	void tacho(void)
	{
		cout << "Aktuelle Geschwindigkeit: " << geschwindigkeit << endl;
	}
};
```

Jede Eigenschaft und jede Methode kann entweder öffentlich (Englisch: `public`) oder nicht öffentlich (Englisch: `private`) sein. Ohne Angabe von `public` und `private` werden alle Eigenschaften und Methoden automatisch als `private` eingestuft. Die Zugriffsrechte beziehen sich auf lesen sowie auf schreiben. Um die Geschwindigkeit in unserem Beispiel auszugeben gibt es die Methode `tacho()`.

Obwohl es in unserem Beispiel nicht vorkommen wird, ist es auch möglich Variablen `public` und Funktionen `private` zu definieren. Dies ist in den meisten Fällen aber nicht angebracht, da vor allem bei `public` Variablen das objektorientierte Konzept umgangen wird. Stattdessen sollte man lieber eine Methode schreiben, die den Wert setzt und eine die den Wert zurückgibt und diese beiden dann als `public` definieren. In den Methoden können die Variablen des eigenen Objekts ganz normal mit ihrem Namen angesprochen werden.

13.5. Einbinden der Klassen in das Programm

Jetzt haben wir unseren Bauplan bzw. Klasse für "unser Auto". Nun muss anhand von diesem ein wirkliches Auto (Objekt) erstellt werden. Genau das gleiche muss in unserem Programmcode passieren. Dazu erstellen wir ein Objekt der Klasse `MeinAuto`.

```
#include <iostream>

using namespace std;

class MeinAuto
{
...
};

int main()
{
      MeinAuto meinAuto1;
      meinAuto1.starten();

      meinAuto1.tacho();
      meinAuto1.beschleunigen();
      meinAuto1.beschleunigen();
      meinAuto1.tacho();

      return 0;
}
```

Mit `MeinAuto meinAuto1;` wir ein Objekt mit dem Namen `meinAuto1` von der Klasse `MeinAuto` erstellt. Die Erstellung eines Objekts folgt der gleichen Syntax, wie das Definieren von Variablen. (`int zahl;`).

`meinAuto1.starten();` ruft die Methode `starten()` des Objekts `meinAuto1` auf. Mit dem Punkt-Seperator hinter einem Objekt, `meinAuto1.` , kann man auf die Eigenschaften und Methoden von der Klasse `MeinAuto` zugreifen, sofern diese öffentlich (`public`) sind.

Nachdem die Funktion `starten()` von `meinAuto1` aufgerufen wurde, wird mit der Methode `tacho()` die Variable `geschwindigkeit` ausgegeben und dann zweimal mit `beschleunigen()` erhöht. Danach wird `geschwindigkeit` noch einmal ausgegeben.

Die Ausgabe auf den Bildschirm sieht folgendermaßen aus:

```
Aktuelle Geschwindigkeit: 0
Aktuelle Geschwindigkeit: 2
```

13.6. Konstruktor

Wie wir bei unserem Beispiel bemerkt haben, müssen wir den Variablen eines Objektes Anfangswerte zuweisen, was bei uns mit der Methode `starten()` geschah. Sonst kann es passieren, dass manche Variablen zufällige Werte haben, die sich unter Umständen auf den Programmverlauf auswirken können.

Daher ist es sinnvoll schon bei der Erstellung des Objektes `meinAuto1` der Variablen `geschwindigkeit` einen Wert zuzuweisen. Dies geschieht mit Hilfe eines Konstruktors. Er wird bei der Erstellung von `meinAuto1` automatisch aufgerufen. Dadurch können wir auf die Methode `starten()` mit ihrer jetzigen Aufgabe verzichten.

Definition:
Der Konstruktor hat immer den gleichen Namen wie die Klasse, in unserem Fall also `MeinAuto`.
Er wird fast wie eine normale Funktion definiert, außer dass er keinen Rückgabewert hat. Man darf auch kein `void` als Rückgabewert schreiben. Außerdem muss er `public` sein.

```
class MeinAuto
{
private:
	// Variable -> Eigenschaften:
	int geschwindigkeit;
	int tankinhalt;

public:
	// Funktionen -> Methoden:
	MeinAuto(void)
	{
		geschwindigkeit = 0;
	}
...
};
```

Dem Konstruktor können, wie einer normalen Methode auch, Parameter übergeben werden. Allerdings muss dann auch bei der Erstellung des Objekts ein Parameter übergeben werden.

```
class MeinAuto
{
private:
	// Variable -> Eigenschaften:
	int geschwindigkeit;
	int tankinhalt;

public:
	//Funktionen -> Methoden:
	MeinAuto(int newGeschwindigkeit)
	{
		geschwindigkeit = newGeschwindigkeit;
	}
	MeinAuto(void)
	{
		geschwindigkeit = 0;
	}
...
};

int main()
{
	MeinAuto meinAuto1(2);
...
}
```

Beim Erstellen sucht sich C++ einen Konstruktor passend zu den Übergabeparametern aus, wie wir es auch beim Überladen von Funktionen kennen gelernt haben. Die Ausgabe in diesem Fall wäre 2 und 4.

14. Praktikum 5

14.1. Aufgabe 1 - Bruch

Schreiben sie eine Klasse Bruch mit den Eigenschaften Zaehler und Nenner. Die Klasse Bruch soll drei Konstruktoren haben, außerdem eine Methode zur Ausgabe des Bruchs auf den Bildschirm, zwei zum Setzten vom Zaehler und Nenner und vielleicht eine zur Überprüfung, ob der Bruch eine Ganzzahl ist.

14.2. Aufagbe 2 – Erweiterung Bruch

Eine Erweiterung für unsere Bruchklasse wäre eine Methode die, den Bruch gegebenenfalls kürzen kann. Dies stellt aber eine sehr hohe Schwierigkeitsstufe dar.
Außerdem kann man eine Methode zur Addition von Brüchen schreiben, wozu man allerdings die Funktion zum Kürzen braucht.

15. Fortsetzung Objektorientierung

15.1. Statische Eigenschaften

In C++ existiert die Möglichkeit Eigenschaften als `static` zu definieren. Dies wirkt sich entscheidend auf den Einsatz dieser Eigenschaften aus. Eine als `static` definierte Eigenschaft hat für alle Objekte einer Klasse den gleichen Wert. Deshalb nennt man diese Eigenschaften auch Klassenvariablen. Erstellt man also mehrere Objekte einer Klasse so können alle Objekte auf die `static`-Eigenschaften zugreifen und der Wert der Eigenschaft ist in allen Objekten identisch.

Beispiel:

```
// OOStaticBeispiel.cpp - Dominik Bruhn - 24.08.2005
// Zeigt den unterschied zwischen statischen und nicht statischen
Variablen

#include <iostream>

using namespace std;

// Definition der Test-Klasse
class Statictest
{
public:
	// Normale Variable
	int variable;
	// Statische Variable
	static int statischeVariable;

	// Konstruktor zum Initialisieren der normalen Variablen
	Statictest()
	{
		variable=0;
	}

	// erhöht beide Variablen
	void inc()
	{
		++variable;
		++statischeVariable;
	}
};
// Initialisierung der statischen Variablen
int Statictest::statischeVariable = 0;

int main()
{
	// Objekt eins der Klasse
	Statictest eins;
	// Variablen dreimal erhöhen
	eins.inc();
	eins.inc();
	eins.inc();

	// Objekt zwei der Klasse
	Statictest zwei;
	// Variablen zweimal erhöhen
	zwei.inc();
	zwei.inc();
```

```
	// Objekt drei der Klasse
	Statictest drei;
	// Variablen erhöhen
	drei.inc();
	// Werte für jede der Klassen ausgeben
	cout << "Objekt eins: Variable=" <<eins.variable
	     << " Statische Variable=" << eins.statischeVariable << endl;
	cout << "Objekt eins: Variable=" <<zwei.variable
	     << " Statische Variable=" << zwei.statischeVariable << endl;
	cout << "Objekt eins: Variable=" <<drei.variable
	     << " Statische Variable=" << drei.statischeVariable << endl;
}
```

Die Ausgabe dieses Programms ist:

```
Objekt eins: Variable=3 Statische Variable=6
Objekt eins: Variable=2 Statische Variable=6
Objekt eins: Variable=1 Statische Variable=6
```

Erklärung: Die Eigenschaft mit dem Namen „variable" ist nich statisch, hat also für jedes Objekt einen anderen Wert (Im Unterschied zu Klassenvariablen nennt man die Variablen Instanzvariablen). Deshalb unterscheiden sich bei ihr auch die Ausgaben der einzelnen Objekte. Die Eigenschaft `statischeVariable` dagegen ist statisch und hat deshalb nur einen Wert für alle Objekte. Sobald ein Objekt den Wert erhöht gilt das gleichzeitig für alle anderen Objekte dieser Klasse.

Statische Variablen müssen in der Klassendefinition mit dem Schlüsselwort `static` markiert werden. Dieses muss noch vor dem Variablen-Typ geschrieben werden, also im Format `static [variablentyp] [variablenname]`. Außerdem muss wie bei normalen Variablen auch der statischen Eigenschaft ein Startwert zugewiesen werden. Dies kann jedoch nicht im Konstruktor erfolgen, da sonst die Variable bei jedem neuen Objekt wieder auf Null gesetzt würde, sondern muss mit einer neuen Syntax direkt nach der Klassendefinition erfolgen. Dieser Syntax lautet: `[variablentyp] [Klassenname]::[variablenname] = [startwert]`.
Es ist also zu beachten, dass der Typ der statischen Variable zweimal festgelegt werden muss, einmal in der Klasse und einmal beim Zuweisen des Startwerts.

Aufgabe:

- Schreiben Sie eine Klasse, die mitzählt, wie viele Objekte einer Klasse erstellt wurden und die beim Erstellen ausgibt, das wievielte Objekt vorliegt.

15.2. Trennung von Deklaration und Definition

In C++ gibt es die Möglichkeit bei Klassen, die Deklaration von der Definition zu trennen. Wegen der Übersichtlichkeit verwendet man hierzu zwei unterschiedliche Dateien. Für die Deklaration so genannte Headerdateien (*.h) und für die Definition die uns schon bekannten Quellcodedateien (*.cpp). Die Dateinamen sollten wie die Klassen lauten.

Deklaration

Hier werden nur die Klasse, Methodenköpfe und die Variablen deklariert. Es wird nur das Erscheinen der Methoden nach außen deklariert (die Schnittstelle der Methoden). Nicht was die Methoden eigentlich machen. Hier wird auch deklariert, welche Methoden und Eigenschaften `public` oder `private` sind.

Definition

Hier wird nur der eigentliche Programmcode der Funktionen geschrieben. Also alles was die Funktion macht. Die Initialisierung der Klassenvariablen kommt ggf. hinzu.

Nun unser Beispiel von oben auf drei Dateien verteilt:

```
// OOBeispiel3.cpp - 25.8.2005 - Roman Keller & Dominik Bruhn
// Demonstration für eine ausgelagerte Klasse
#include <iostream>
#include "MeinAuto.h"

using namespace std;

int main()
{
	MeinAuto meinAuto1;

	meinAuto1.tacho();
	meinAuto1.beschleunigen();
	meinAuto1.beschleunigen();
	meinAuto1.tacho();

	return 0;
}
```

In der Hauptdatei steht nun nur noch der Quellcode für die `main()`-Funktion. Außerdem muss die Headerdatei unserer neuen Klasse (`MeinAuto.h`) inkludiert werden. Damit ist dem Kompiler unsere neue Klasse bekannt. Wichtig ist dabei, dass die Headerdatei in Anführungszeichen und nicht wie gewohnt in Kleiner-, Größerzeichen steht. Der Unterschied hierbei ist, dass bei Kleiner-, Größerzeichen nur im Systempfad gesucht wird, wo die vorgefertigten Bibliotheken und Klassen liegen. Bei Anführungszeichen wird zusätzlich auch noch der lokale Ordner des Projekts durchsucht. Eben dort wo unsere selbst angelegte Klasse gespeichert ist.

```
// MeinAuto.h - 25.8.2005 - Roman Keller & Dominik Bruhn
// Headerdatei von Klasse MeinAuto.
#ifndef MeinAuto_H
#define MeinAuto_H

class MeinAuto
{
private:
	// Variablen -> Eigenschaften:
	int geschwindigkeit;
	int tankinhalt;

public:
	// Funktionen -> Methoden:
	MeinAuto(int newGeschwindigkeit);

	MeinAuto(void);
	void starten(void);
	void beschleunigen(void);
	void abbremsen(void);
	void tacho(void);
};
#endif // MeinAuto_H
```

In der Headerdatei für unsere `MeinAuto`-Klasse stehen nur der Aufbau der Klasse, die Variablen Deklarationen und die Methodenköpfe. Die Variablennamen der Parameter für die Methoden sind optional. Es ist aber zu empfehlen, sie aufzunehmen, da so bei späterer Verwendung die Reihenfolge der Parameter anhand der Namen ersichtlich ist.

Das sich am Anfang befindliche `#ifndef MeinAuto_H` ist eine Präprozessor-Direktive, die dafür sorgt, dass die Headerdatei nur einmal inkludiert wird.
Sie überprüft, ob die Kompilervariable `MeinAuto_H` definiert ist (`ifndef` (if Not Defined = Wenn nicht definiert).
Wenn dies der Fall ist, wird der Code nicht noch einmal inkludiert. Wenn `MeinAuto_H` nicht definiert ist, wird sie mit `#define MeinAuto_H` definiert und sorgt daher dafür, dass bei einem zweiten Aufruf der Headerdatei sie nicht noch einmal eingearbeitet wird.

Die Variable kann beliebig gewählt werden. Idealerweise sollte sie dem Dateinamen ähneln. Zum Beispiel [KLASSENNAME]_H

Ab `#endif` wird der Quellcode wieder ganz normal durchgearbeitet. Hier endet die bedingte Codeausführung. Man kann sich dies wie eine schließende geschweifte Klammer vorstellen, die einen `if`-Block begrenzt.

Die Präprozessordirektive verhindert also, dass es zu Doppeldeklarationen kommt, wenn die Datei zweimal inkludiert wird.

```
// MeinAuto.cpp - 25.8.2005 - Roman Keller & Dominik Bruhn
// Quellcodedatei für MeinAuto

#include "MeinAuto.h"

#include <iostream>

using namespace std;

MeinAuto::MeinAuto(void)
{
      geschwindigkeit = 0;
}
MeinAuto::MeinAuto(int newGeschwindigkeit)
{
      geschwindigkeit = newGeschwindigkeit;
}
void MeinAuto::beschleunigen(void)
{
      ++geschwindigkeit;
}

void MeinAuto::abbremsen(void)
{
      --geschwindigkeit;
}
void MeinAuto::tacho(void)
{
      cout << "Aktuelle Geschwindigkeit: " ;
      cout << geschwindigkeit << endl;
}
```

Die Quellcodedatei unserer Klasse ist auch etwas verändert. Hier fehlt jegliche Strukturdefinition. Es wird nur der Quellcode der Methoden geschrieben. Man sagt, dass die Methoden implementiert werden. In der Header-Datei wurden sie nur deklariert.

Am Anfang muss die Headerdatei der Klasse inkludiert werden, in unserem Fall `MeinAuto.h`, da dort die Struktur der Klasse hinterlegt ist.

Das `MeinAuto::` bedeutet, dass die Funktion eine Methode der Klasse `MeinAuto` ist, wie sie in der Headerdatei beschrieben wurde.

Allgemeiner Aufbau bei Deklarations- und Definitionstrennung

Deklaration Headerdatei (*.h):

```
#ifndef [KOMPILERVARIABLE]
#define [KOMPILERVARIABLE]

[INCLUDES]

class [CLASSNAME]
{
private:
      [VARTYPE] [VARNAME];
      ...
public:
      [KONSTRUKTORNAME]([PARAMTER m. NAME]);
      ...
      [RÜCKGABEWERT] [METHODENNAME] ( [PARAMETER m. NAME] );
      ...
};
#endif
```

Definition Quellcodedatei (*.cpp):

```
#include "[CLASSNAME].h"

[INCLUDES]

[CLASSNAME]::[KONSTRUKTORNAME] ( [PARAMETER m. NAME] )
{
[ANWEISUNGEN];
}
...

[RÜCKGABEWERT] [CLASSNAME]::[METHODENNAME] ( [PARAMETER m. NAME] )
{
[ANWEISUNGEN];
}
```

15.3. Die Graphik-Klasse

Damit wir graphische Ausgaben machen können, benutzen wir eine Graphik-Klasse. Diese wurde von uns modifiziert und gehört nicht zur Standardausstattung einer C++ Entwicklungsumgebung.

Einbinden der Graphik-Klasse:
Erst müssen wir die beiden Dateien der Klasse in unser Projektverzeichnis kopieren. Die Dateien befinden sich auf der beigefügten DVD.

Dann müssen die Quelldateien unserem Projekt hinzugefügt werden. Hierzu müssen wir mit einem Rechtsklick auf **Source Files** (Quelldateien). Dann **Add** (Hinzufügen) und **Add existing Item** (Vorhandenes Element hinzufügen). Dann die *graphclass.cpp* auswählen und dann auf **Open** (Öffnen) klicken. Dasselbe für die Headerdatei (graphclass.h) durchführen, allerdings mit einem Rechtsklick auf **Header Files** (Headerdateien).

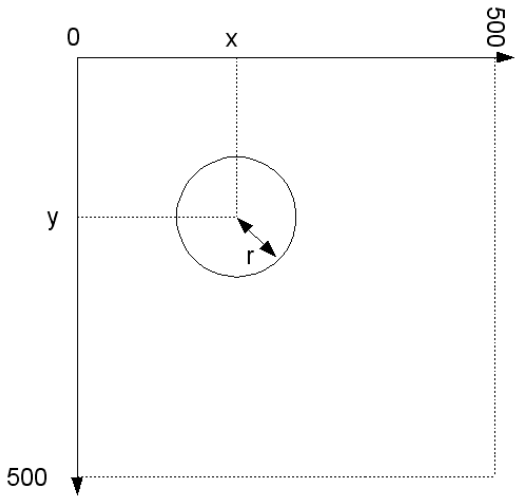
Danach sollten wir die Datei inkludieren:

```
#include "graphclass.h"
```

Um Graphik ausgeben zu können müssen wir als erstes ein graphisches Ausgabefenster erstellen. Dieses wird bei der Erstellung eines Objekts der Graphik-Klasse automatisch vom Konstruktor erstellt:

```
graphclass graph(500,500);
```

Nun haben wir ein Fenster, auf dem wir unsere graphischen Elemente ausgeben können mit 500 auf 500 Pixeln erstellt.



Ein Pixel ist ein Bildpunkt auf dem Bildschirm, den wir einfärben können. Dieser Begriff sollte von der Bildschirmauflösung bekannt sein.

Um graphische Elemente auf diesem Fenster ausgeben zu können, muss man sich dieses Fenster wie ein Koordinatensystem vorstellen. Allerdings ist der Nullpunkt in der linken oberen Ecke.

Jetzt können wir auf dieses graphische Fenster ausgeben.

Einen Kreis ausgeben:

```
graph.circle(color, xCord, yCord, rad);
```

Somit ersteht ein Kreis mit dem Mittelpunkt an der x-Koordinate `xCord` und der y-Koordinate `yCord`, mit dem Radius von `rad` - Pixel und der Farbe `color`.

Einen gefüllten Kreis ausgeben:

```
graph.fillcircle(color, xCord, yCord, rad);
```

Diese Methode erstellt genau dasselbe, nur dass der Kreis mit der angegebenen Farbe `color` gefüllt ist.

Ein Rechteck ausgeben:

```
graph.rectangle(color, xCord1, yCord1, xCord2, yCord2);
```

Diese Methode malt ein Recheck mit der Farbe `color` auf das Fenster. Die Größe des Rechtecks ist angegeben durch die Koordinaten der linken oberen (`xCord1/yCord1`) und der rechten unteren Ecke (`xCord2/yCord2`).

Ein gefülltes Rechteck ausgeben:

```
graph.fillrectangle(color, xCord1, yCord1, xCord2, yCord2);
```

Diese Methode macht wiederum dasselbe wie `graph.rectangle` außer dass das gemalte Rechteck mit der Farbe `color` gefüllt ist.

Einen Text an einer bestimmten Stelle ausgeben:

```
graph.outtextxy(color, xCord, yCord, meinText);
```

Gibt einen Text `meinText` mit der Schriftfarbe `color` an den Koordinaten `xCord` und `yCord` auf das graphische Fenster aus. Die Koordinaten sind der linke obere Punkt des Textes.

Die Schriftgröße ändern:

```
graph.settextsize(size);
```

Setzt die Schriftgröße auf `size` für Texte, die auf das graphische Fenster ausgegeben werden.

Löschen des graphischen Fensters:

```
graph.clear();
```

Löscht den kompletten Inhalt des graphischen Fensters, also alle Elemente die bis dahin darauf gezeichnet wurden. Damit wir wieder auf eine leere Fläche zeichnen können.

Die Wartenmethode:

Da bei einer graphischen Ausgabe meist eine Aktion passiert ohne dass der Benutzer dabei Eingaben tätigen muss, wird der Computer das Programm sehr schnell ausführen können, da er nicht unterbrochen wird. Dadurch würde die Ausgabe so schnell erfolgen, dass wir dies nicht wahrnehmen.

Deshalb benötigen wir eine Methode die den Computer pausieren lässt.

```
graph.delay(200);
```

Bei dieser Methode wartet der Computer für 200 Millisekunden, bevor er den nächsten Befehl in Angriff nimmt.

Mögliche Farben:

Für `color` können folgende Werte eingesetzt werden:

- BLACK
- BLUE
- GREEN
- CYAN
- RED
- MAGENT
- BROWN
- LIGHTGRAY
- DARKGRAY
- LIGHTBLUE
- LIGHTBREEN
- LIGHTCYAN
- LIGHTRED
- LIGHTMAGENTA
- YELLOW
- WHITE

Man muss beachten, dass diese Farben immer in Großbuchstaben geschrieben sein müssen.

15.4. Pointer auf Objekte

Weiter vorne im Skript haben wir Pointer kennen gelernt. Diese Pointer können auch auf Objekte zeigen, die ja ebenfalls im Speicher hinterlegt sind.

Wenn wir in einer Methode auf ein anderes Objekt zugreifen wollen, können wir ja nicht die Objektvariable übergeben, da dabei automatisch eine Kopie des Objekts erstellt wird. Da wir aber direkt auf das Graphik-Objekt, mit Hilfe einer Zeichenfunktion, zugreifen wollen, können wir keine Kopie übergeben. Bei einer Veränderung der Kopie wird das Orginal-objekt nicht verändert.

Als Beispiel nehmen wir ein Graphik-Objekt. Dieses soll in der Hauptfunktion erstellt werden. Nun haben wir mehrere Klassen, die unterschiedliche geometrische Objekte darstellen. Eine davon stellt einen Kreis da. Sie besitzt x und y-Koordinaten des Mittelpunkts und einen Radius.

Ein Objekt dieser Klasse soll sich mit Hilfe der Methode `draw()` selbst zeichen. Dies bedeutet, dass wir auf das Graphik-Objekt zugreifen müssen um mittels dessen `circle`-Methode einen Kreis auszugeben.

Um nun von unserem Objekt aus auf ein anderes Objekt zugreifen zu können, müssen wir unserem Objekt einen Pointer auf das andere Objekt übergeben.

Die Methoden und Eigenschaften, sofern diese `public` sind, können dann im Objekt, dem der Pointer übergeben wurde, aufgerufen werden.

Wie wir gelernt haben, kann man mit dem Punkt-Selektor auf die Methoden eines Objekts zugreifen.
Wenn wir über den Pointer auf die Methoden des Objekts zugreifen wollen, so müssen wir einen Pfeil verwenden. Dieser ist zusammengesetzt aus einem Bindestrich und einem Größerzeichen.

```
OBJEKT.METHODE() wird zu ZeigerAufOBJEKT -> METHODE()
```

Als Beispiel nehmen wir wieder unser Kreis-Objekt, dessen `draw`-Methode dem Zeiger auf das Graphik-Objekt übergeben wird.
In der Methode `draw()` wird dann auf das Graphik-Fenster gezeichnet.

```
void Kreis::draw(graphclass * pGraph)
{
      pGraph->circle(color,x,y,radius);
}
```

Diese Methode wird von einem Objekt `kreis1` der Klasse Kreis aufgerufen mittels:

```
graphclass graph;

kreis1.draw( &graph);
```

15.5. Zusammenfassung

- Klassen werden durch Klassenstrukturen mit dem Schlüsselwort `class` deklariert.
- Klassennamen fangen mit einem Großbuchstaben an.
- Eigenschaften und Methoden einer Klasse besitzen Zugriffrechte, die durch die Schlüsselwörter `private` und `public` bestimmt werden.

- Eigenschaften sollten `private` sein und sich nur über `public` Methoden verändern lassen.

- Es gibt Konstruktoren, die bei der Erstellung eines Objekts von einer Klasse automatisch aufgerufen werden. Sie tragen den Namen der Klasse und haben keinen Rückgabewert.

- Man kann die Deklaration und die Definition einer Klasse voneinander trennen. Also in verschiedene Dateien schreiben (*.h und *.cpp).

- Headerdateien dürfen von Kompiler nur einmal durchgearbeitet werden.

- Man kann mit Pointern auch auf Objekte zeigen

- Um mittels Pointer auf Methoden von Objekten zuzugreifen, benutzt man einen Pfeil ->

16. Praktikum 6

16.1. Aufgabe 1 – Graphikklassen

Schreiben sie drei Klassen. Jede dieser Klassen steht für eine Figur, nämlich Rechteck, Kreis und Quadrat. Dabei speichert jede Klasse die x und y Koordinaten. Außerdem hat die Klasse Rechteck die Eigenschaften für Breite und Höhe, der Kreis Eigenschaften für den Radius und das Quadrat für die Seitenlänge. Diese Parameter (also x, y und die weiteren Eigenschaften) werden dem Konstruktor der einzelnen Klassen übergeben. Es existiert eine Methode in jeder Klasse mit der die Figur gezeichnet wird. Dieser Methode wird ein Zeiger auf ein Objekt der Graphikklasse übergeben. eine Grafikklasse übergeben. Alle gleichen Figuren haben die gleiche Farbe.
Schreiben Sie außerdem eine Anwendung, die als erstes ein Fenster für die Grafikausgabe anzeigt, dann zwei Quadrate, zwei Rechtecke und zwei Kreise erstellt und diese danach zeichnet.

17. Vererbung

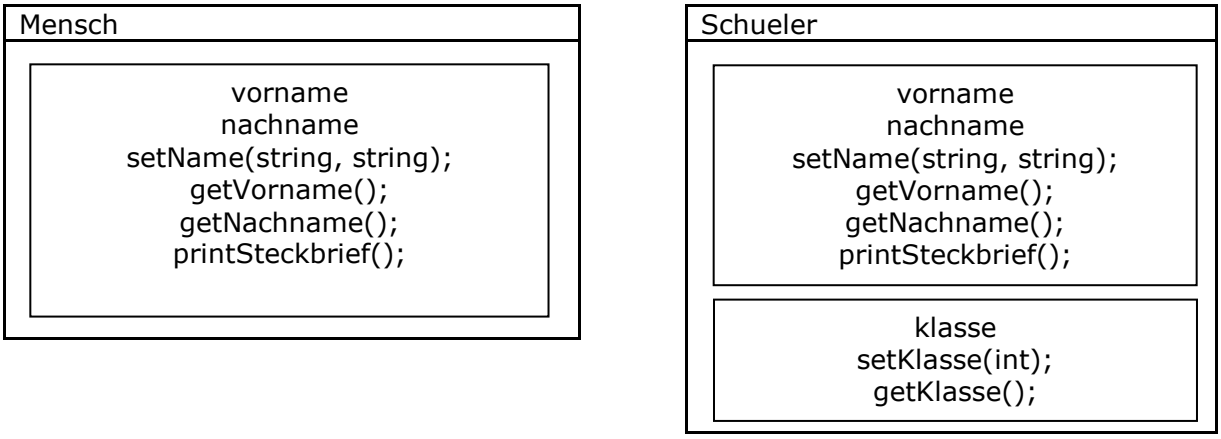
Wir haben nun einige der wichtigsten Aspekte der Objektorientierung vorgestellt. Nun kommen wir zu einem weiteren wichtigem Prinzip, nämlich der Vererbung.

Vererbung bedeutet, dass eine neue Klasse erstellt wird, welches von einer anderen Klasse alle Eigenschaften und Methoden (bis auf die Konstruktoren) erbt und diese selbst benutzen und auch überschreiben kann.

Die Klasse, die erbt, ist die so genannte Kindklasse. Die Klasse, die ihr Eigenschaften und Methoden weitergibt, wird üblicherweise Vaterklasse genannt. Dies ist vor allem dann sinnvoll, wenn man mehrere verschiedene Klassen hat, die sehr viele Gemeinsamkeiten besitzen und nur geringe Abweichungen voneinander haben. Zum Beispiel: Schüler und Lehrer.
Es sind beides Menschen, die ein Alter, einen Vor- und Zunamen, ein Geburtsort und eine Körpergröße besitzen. Der Schüler hat noch Noten und geht in eine Klasse, währen der Lehrer Fächer hat, die er unterrichtet.

Den kompletten Unterbau für beide separat zu erstellen wäre sehr umständlich, da alles zweimal durchgeführt werden müsste.

Deshalb ist es sinnvoll eine eigene Klasse `Mensch` zu erstellen, welche die oben genannten Eigenschaften und dazugehörigen Methoden besitzt. Und die Klasse `Lehrer` und `Schueler` dann abgeleitet. Wenn man dann etwas an der Klasse `mensch` verändert, wird dies automatisch auf die beiden Kinder `Lehrer` und `Schueler` übertragen.



Die Vaterklasse `Mensch`:

Die Vaterklasse ist eine ganz normale Klasse, wie wir sie schon kennen gelernt haben. Sie hat die Eigenschaften `vorname` und `nachname` und die jeweiligen Methoden, um diese zu setzen oder auszulesen. Sie besitzt einen Konstruktor und eine Methode zum Ausgeben eines Steckbriefs.
Ein Unterschied ist jedoch vorhanden. Anstatt `private` verwenden wir nun `protected` für unsere geschützten Variablen.
Bei `private` haben die Kinderklassen keinen Zugriff. `protected` verhält sich genauso wie `private`, nur das die Kinderklassen Zugriff haben, was sinnvollerweise notwenig ist.

```
// Mensch.h - 29.8.2005 - Roman Keller
// Headerdatei fuer Mensch

#ifndef Mensch_H
#define Mensch_H
```

```
#include <string>
#include <iostream>

using namespace std;

class Mensch
{
protected:
      string vorname;
    string nachname;
public:
      Mensch(void);

      Mensch(string newVorname, string newNachname);

      void setName(string newVorname, string newNachname);

      string getVorname(void);

      string getNachname(void);

      virtual void printSteckbrief(void);
};
#endif // Mensch_H

// Mensch.cpp - 29.8.2005 - Roman Keller
// Quellcodedatei fuer Mensch

#include "Mensch.h"

Mensch::Mensch(void)
{
    vorname ="";
      nachname="";
}
Mensch::Mensch(string newVorname, string newNachname)
{
    vorname = newVorname;
      nachname = newNachname;
}
void Mensch::setName(string newVorname, string newNachname)
{
    vorname = newVorname;
      nachname = newNachname;
}
string Mensch::getVorname(void)
{
      return vorname;
}
string Mensch::getNachname(void)
{
      return nachname;
}
void Mensch::printSteckbrief (void)
{
      cout << endl;
      cout << "\t\t******** Steckbrief ********" << endl;
      // \t macht einen Einschub nach rechts
      cout << endl;
      //dadurch werden die Daten schöner ausgegeben
      cout << "\tName: \t\t" << vorname << " " << nachname << endl;
}
```

Die Kindklasse `Schueler`:

Um eine Klasse erben zu lassen, muss man bei der Definition schreiben:

```
class [CLASSNAME] : public [VATERCLASSNAME]
```

Wie oben beschrieben wird der Konstruktor der Vaterklassen nicht mit vererbt. Da wir die Funktionalität des Konstruktors nicht noch einmal implementieren wollen, rufen wir ihn in unserem neuen Konstruktor auf.

Dies erfolgt direkt nach dem Funktionskopf des Konstruktors mit:

```
[KINDCLASSNAME]::[KINDCLASSNAME] ( [PARAMETER] )
:[VATERKOSNTRUKTOR] ( [PARAMETER] )

Kind(string name)
:Vater(name)
{
      ...;
}
```

Wenn wir nun ableiten, ist es sinnvoll einige Methoden zu verändern, also überschreiben wir im Kind einige Methoden des Vaters. So muss zum Beispiel `printSteckbrief()` verändert werden.
Wir wollen nämlich im Steckbrief auch noch die weiteren Eigenschaften unserer Kinder ausgeben. Es wäre umständlich die ganze `printSteckbrief()` Methode neu zu schreiben. Deshalb rufen wir in unserer neuen Funktion `printSteckbrief()` die Funktion der Vaterklasse einfach noch einmal auf.

Dies geht ähnlich wie beim Konstruktor:

```
[VATERCLASSNAME]::[VATERMETHODE] ( [PARAMETER] );
```

```
// Schueler.h - 29.8.2005 - Roman Keller
// Headerdatei fuer Schueler

#ifndef Schueler_H
#define Schueler_H

#include "Mensch.h"

class Schueler : public Mensch
{
protected:
      int klasse;

public:
      Schueler(void);

      Schueler(string newVorname, string newNachname, int newKlasse);

      void setKlasse(int newKlasse);

      int getKlasse(void);

      void printSteckbrief(void);
};
#endif // Schueler_H
```

```
// Schueler.cpp - 29.8.2005 - Roman Keller
// Quellcodedatei fuer Schueler

#include "Schueler.h"

Schueler::Schueler(void)
:Mensch()
{
      klasse = 1;
}
Schueler::Schueler(string newVorname, string newNachname, int newKlasse)
:Mensch(newVorname, newNachname)
{
      klasse = newKlasse;
}
void Schueler::setKlasse(int newKlasse)
{
      klasse = newKlasse;
}
int Schueler::getKlasse(void)
{
      return klasse;
}
void Schueler::printSteckbrief (void)
{
      Mensch::printSteckbrief();
      cout <<"\tKlasse: \t" << klasse << endl;
}
```

Die Kindklasse Lehrer:

```
// Lehrer.h - 29.8.2005 - Roman Keller
// Headerdatei fuer Lehrer

#ifndef Lehrer_H
#define Lehrer_H

#include "Mensch.h"

class Lehrer : public Mensch
{
protected:
      string fach;

public:
      Lehrer(void);

      Lehrer(string newVorname, string newNachname,string newFach);

      void setFach(string newFach);

      string getFach(void);

      virtual void printSteckbrief(void);
};
#endif // Lehrer_H
```

```
// Lehrer.cpp - 29.8.2005 - Roman Keller
// Quellcodedatei fuer Lehrer

#include "Lehrer.h"

Lehrer::Lehrer (void)
:Mensch()
{
    fach = "";
}
Lehrer::Lehrer(string newVorname, string newNachname,string newFach)
:Mensch(newVorname, newNachname)
{
      fach = newFach;
}
void Lehrer::setFach(string newFach)
{
      fach = newFach;
}
string Lehrer::getFach(void)
{
      return fach;
}
void Lehrer::printSteckbrief(void)
{
      Mensch::printSteckbrief();
      cout << "\tFach:\t\t" <<fach << endl;
}

// main.cpp - 29.8.2005 - Roman Keller
// Hauptdatei des Vererbungsprojekts

#include "Mensch.h"
#include "Schueler.h"
#include "Lehrer.h"

using namespace std;

int main()
{
      Mensch roman("Roman", "Keller");

      roman.printSteckbrief();

      Schueler dominik("Dominik", "Bruhn",13);

      dominik.printSteckbrief();

    Lehrer anton("Anton", "Trojosky","Mathe");

      anton.printSteckbrief();

      return 0;
}
```

18. Praktikum 7

18.1. Aufgabe 1 – Drei Klassen Vererbung

Erstellen Sie analog zu Aufgabe 16.1. drei Klassen, Kreis, Rechteck und Quadrat. Entwerfen sie einen Erbfolge mit einer Vaterklasse für alle der drei Klassen. Diese sollte alle gemeinsamen Eigenschaften und Methoden enthalten.

19. Ausblick

Einige Themen konnten wir wegen der beschränkten Zeit und ihrer Komplexität leider im Script und im Vortrag nicht behandeln. Weil dies wichtige Themen in der Objektorientierung sind, werden wir sie kurz anführen und erläutern.

Polymorphie	Die Eigenschaft von Objekten abgeleiteter Klassen, auch wie ihr Vater auftreten zu können. Dabei können sie aber nur Eigenschaften verwenden, die sie vom Vater geerbt haben. Ein Kreis, abgeleitet von GraphObj, kann als GraphObj auftreten und dann mittels dessen Methode move() der Grafikklasse bewegt werden. Man kann alle Ableitungen von GraphObj mit move() bewegen, ohne wissen zu müssen, was es wirklich ist (Kreis, Rechteck). Kinder können also vielgestaltig (polymorph) auftreten. Zu diesem Thema befindet sich auf der beigefügten CD ein Beispiel.
Virtuelle Methoden	Durch die Polymorphie können Objekte einer abgeleiteten Klasse als Objekte ihrer Vaterklasse angesprochen werden. Wird eine Vatermethode als virtuell deklariert, dann verhält sich die überschriebene Methode in der Kindklasse plötzlich anders. Übergibt man einem Zeiger auf ein Vaterobjekt die Adresse eines Kindobjektes dann verwendet das Vaterobjekt jetzt nicht die Vatermethode sondern die überschrieben Kindmethode.
Mehrfachvererbung	Die Möglichkeit einer Klasse, die Methoden und Eigenschaften von mehr als nur einer Klasse zu erhalten. Vergleich hierzu: Mutter Vater \ / Kind Das bedeutet, das Kind hat zwei Klassen von denn es abgeleitet wurde. Dies ist typisch für C++ und kann in dieser Form nur in sehr wenigen anderen Programmiersprachen benutzt werden.
Destruktoren	Gegenteil von Konstruktoren. Sie werden nicht bei der Erstellung des Objekts, sondern bei ihrer Zerstörung aufgerufen. Das dient meist zu Freigabe von Speicher.

20. Buchempfehlungen

- C/C++, Das komplette Programmierwissen für Studium und Job – Dirk Louis – Markt+Technik – 1782 Seiten – 29,95€

Einsteiger		*Fortgeschrittene*		*Profi*
+	+	+	+	

- C++, Einführung und Professionelle Programmierung – Ulrich Breymann – Hanser – 538 Seiten – 39,90€

Einsteiger		*Fortgeschrittene*		*Profi*
	+	+	+	

- Objektorientiertes Programmieren in C++ – Nicolai Josuttis – Addison-Wesley – 613 Seiten – 39,95€

Einsteiger		*Fortgeschrittene*		*Profi*
		+	+	+

- C++, Lernen und professionell anwenden – Peter Prinz, Ulla Kirch-Prinz – mitp – 853 Seiten – 44,95€

Einsteiger		*Fortgeschrittene*		*Profi*
+	+	+	+	

Lösungsaufgaben

Praktikum 1

Lösungen zu 4. Datentypen und Variablen

Aufgabe 1 – Text ausgeben:

```
// textOut.cpp - 16.8.2005 - Roman Keller
// Ausgabe eines beliebigen Textes

#include <iostream>
using namespace std;

int main()
{
    cout << "Willkommen beim C++ Kurs" << endl << "an der Fachhochschule
Esslingen" <<endl;

     return 0;
}
```

Aufgabe 2 - Name:

```
// readName.cpp - 16.8.2005 - Roman Keller
// Einlesen eines Textes (NAME) und anschließende Ausgabe

#include <iostream>
#include <string>

using namespace std;

int main ()
{
    string myName = "";
    cout << "Bitte gebe deinen Namen ein:" << endl;
    cin >> myName;
    cout << "Hallo, dein Name lautet: " << myName << endl;

    return 0;
}
```

Aufagbe 3 - Rechnen:

```
// rechnen.cpp - 16.8.2005 - Roman Keller
// Liesst zahlen ein und gibt sie berechnet aus

#include <iostream>
using namespace std;

int main ()
{
     int zahl1,zahl2 =0;
     cout << "Bitte gib Zahl 1 ein: " ;
     cin >> zahl1;
```

```
        cout << "Bitte gib Zahl 2 ein: ";
        cin >> zahl2;
        cout << "Beide Zahlen miteinander addiert: ";
        cout << zahl1 + zahl2 << endl;
        cout << "Beide Zahlen von einander subtrahiert: ";
        cout << zahl1 - zahl2 << endl;
        cout << "Beide Zahlen miteinander multipliziert: ";
        cout << zahl1 * zahl2 << endl;
        cout << "Zahl1 dividiert durch Zahl2: ";
        cout << zahl1 / zahl2 << endl;
        cout << "Zahl1 modulo Zahl2 (REST): ";
        cout << zahl1 % zahl2 << endl;


        return 0;
    }
```

Aufgabe 4 – Sekunden seid Geburt:

```
    // geburtstag.cpp - 16.8.2005 - Roman Keller
    // Eingabe des Geburtdatums und Ausgabe der Sekunden seit der Geburt

    #include <iostream>
    #include <string>

    using namespace std;

    int main ()
    {
        int tag, monat, jahr;
        int nowTag= 6, nowMonat=9, nowJahr=2005;
        int unterTag, unterMonat, unterJahr;
        int erg ;
        cout << "Gibt den Tag ein, an dem du geboren bist: ";
        cin >> tag;
        cout << "Gibt den Monat ein, an dem du geboren bist: ";
        cin >> monat;
        cout << "Gibt das Jahr an, an dem du geboren bist: ";
        cin >> jahr;
        cout << "Bitte gib den heutigen Tag an: ";
        cin >> nowTag;
        cout << "Bitte gib den aktuellen Monat an: ";
        cin  >> nowMonat;
        cout << "Bitte gib das aktuelle Jahr an: ";
        cin >> nowJahr;

        unterTag = nowTag - tag;
        unterMonat = nowMonat - monat;
        unterJahr = nowJahr - jahr;

        unterTag += unterMonat * 30;
        unterTag += unterJahr * 365;
        erg = unterTag * 24 * 60 * 60;

        cout << "Vom " << tag << "." << monat << "." << jahr;
```

```
        cout << " bis zum ";
        cout << nowTag << "." << nowMonat << "." << nowJahr;
        cout << " sind ";
        cout << unterTag << " Tage vergangen." << endl;
        cout << "Du bist "<< erg << " Sekunden alt!"<< endl;

        return 0;
    }
```

Praktikum 2

Lösungen zu 5.1.2. Einfache Alternative – if unf else

Aufgabe 1 – Schaltjahrberechnung:

```
    // schaltjahr.cpp – 15.08.2005 – Anton Trojosky
    // Berechnung von Schaltjahren

    #include <iostream>

    using namespace std;

    int main ()
    {
        int eingabe;

        cout << "Programm zur Schaltjahrberechnung" << endl;
        cout << endl;
        cout << "Geben sie bitte ein Jahr ein: " << endl;
        cin >> eingabe;

        cout << eingabe;
        if  ( (eingabe > 1582) && ((eingabe % 4) == 0) )
        {
            if ( (eingabe % 100) == 0 )
            {
                if ( (eingabe % 400) == 0 )
                {
                    cout << " ist ein Schaltjahr.";
                }
                cout << " ist kein Schaltjahr.";
            }
            cout << " ist ein Schaltjahr.";
        }
    else
        {
            cout << " ist kein Schaltjahr.";
        }

        cout << endl;
        cout << endl << "Programm wird beendet." << endl;

        return 0;
    }
```

Lösung zu 5.1.4 Switch-Verzweigung

Aufgabe 2 – Wochentagsberechnung:

```
//wochentag.cpp - 30.8.2005 - Roman Keller
//Einlesen eines Datums und Ausgabe des dazugehörigen Wochentags unter
verwendung Zellers Kongruenz (Quelle: wikipedia)

#include <iostream>

using namespace std;

int main()
{
     int tag, monat, jahr, jhdt;            //Datum das eingelesen wird
     int wochenTag;                         //Ausgerechneter Wochentag

     cout << "Bitte gib den aktuellen Tag an: ";
     cin >> tag;
     cout << "Bitte gib den aktuellen Monat an: ";
     cin >> monat;
     cout << "Bitte gib das aktuelle Jahr an: ";
     cin >> jahr;

     jhdt = jahr / 100;

     jahr %= 100;


     // für die datumsausgabe muss bei Werten kleiner 10 eine 0 vorangestellt
werden
     if (jahr <= 9 )
     {
        cout << "Du willst das Datum des " << tag << "." << monat << "." <<
jhdt << "0" << jahr <<endl;
     }
     else
     {
          cout << "Du willst das Datum des " << tag << "." << monat << "." <<
jhdt << jahr <<endl;
     }

     // da Januar und Februar als 13. bzw 14. Monat des vorigen Jahres
betrachtet werden muss ueberprueft werden.
     if (monat <= 2)
     {
          monat = monat + 12;
          if (!(jahr == 0))
          {
               jahr--;
          }
          else
          {
               jahr = 99;
               jhdt--;
```

```
        }
    }

    //Formel zur Berechnung
    wochenTag = ( tag + ( ( (monat + 1) * 26) / 10) + jahr + (jahr / 4) +
(jhdt / 4) - 2*jhdt ) % 7;

    cout << "Der heutige Tag ist ein: ";
    switch (wochenTag)
    {
    case 0:
        cout << "Samstag";
        break;
    case 1:
        cout << "Sonntag";
        break;
    case 2:
        cout << "Montag";
        break;
    case 3:
        cout << "Dienstag";
        break;
    case 4:
        cout << "Mittwoch";
        break;
    case 5:
        cout << "Donnerstag";
        break;
    case 6:
        cout << "Freitag";
        break;
    default:
        cout << "Fehler";
        break;
    }
    cout << endl;

    return 0;
}
```

Lösungen zu 5.2.1. For-Schleife

Aufgabe 3 - Quadratzahlen

```
// Quadratzahlen.cpp – 12.08.2005 – Anton Trojosky
// Berechnung und Ausgabe von 10 Quadratzahlen

#include <iostream>

using namespace std;

int main()
{
    for (int loop = 1; loop <= 10; loop++)
    {
```

```
        cout << "Das Quadrat von " << loop << " ist: " << loop*loop
             << "." << endl;
    }

    return 0;
}
```

Aufagbe 4 – Berechnung von Pi

```
    //BerechnungPi.cpp - 19.08.2005 – Anton Trojosky
    //Berechnen von Pi nach der Reihenentwicklung für artan x = x -1/3 x³ +
    1/5 x^5 - 1/7 x^7 + ...

    #include <iostream>

    using namespace std;

    int main ()
    {
        int schritte;
        double pi=1.0;
        bool minusvorzeichen=false;

        cout << "Naeherungsweise Berechnung von Pi" << endl << endl;
        cout << "Wie viele Schritte sollen durchgefuehrt werden? ";
        cin >> schritte;

        for (int a=3; a <= (schritte*2+3); a += 2)
        {
            if (minusvorzeichen)
            {
                pi = pi + 1.0/a;
                minusvorzeichen = false;
            }
            else
            {
                pi = pi - 1.0/a;
                minusvorzeichen = true;
            }
        }
        pi *= 4.0;
        cout.precision(10);                    // cout gibt bis zu 10
    Stellen nach dem komma aus
        cout << "Pi ist nach " << schritte
             << " Schritten naeherungsweise " << pi << "." << endl;

        return 0;
    }
```

Praktikum 3

Lösungen zu 6. Arrays

Aufgabe 1 – Array - Durchschnitt

```cpp
//ArrayAufgabe1.cpp - 15.08.2005 - Dominik Bruhn
//Liest 10 Zahlen vom Benutzer ein, speichert diese in einem Array, gibt
das Array wieder aus und berechnet den Mittelwert

#include <iostream>

using namespace std;

int main()
{
     int zahlen[10];

     //Zahlen einlesen
     for (int index=0;index<10;index++)
     {
          cout << "Bitte gib die Zahl mit dem Index " << index << " ein:
";
          cin>>zahlen[index];
     }

     //Zahlen ausgeben
     for (int index=0;index<10;index++)
     {
          cout << "Die Zahl mit dem Index " << index << " hatte den Wert
" << zahlen[index] << endl;
     }

     //Durchschnitte berechnen
     int summe=0;
     float ergebniss;
     for (int index=0;index<10;index++)
     {
          summe = summe + zahlen[index];
     }

     ergebniss = summe/10.0f;

     cout << "Der Mittelwert der Werte war " << ergebniss << endl;


     return 0;
}
```

Aufgabe 2 – Array - Index

```cpp
//ArrayAufgabe2.cpp - 15.08.2005 - Dominik Bruhn
//Liest 10 Zahlen vom Benutzer ein, speichert diese in einem Array und
lässt diese den Benutzer später abrufen

#include <iostream>

using namespace std;

int main()  {
     int zahlen[10];
```

```
        //Zahlen einlesen
        for (int index=0;index<10;index++) {
             cout << "Gib Zahl mit dem Index " << index << " ein: ";
             cin >> zahlen[index];
        }

        //Index abfragen und ausgeben
        int selindex;
        do {
             cout << "Bitte Index eingeben: ";
             cin >> selindex;
             cout << "Der Wert von Zahl Nummer " << selindex << " ist " <<
zahlen[selindex] << endl;
        } while ((selindex>=0) && (selindex<=9));
}
```

Lösungen zu 6.7. Sortieren von Arrays

Aufgabe 3 – Sortieren von Arrays

```
//ArraySortAufgabe1.cpp - 19.08.2005 - Dominik Bruhn
//Sortiert ein vorgegebenes Array und gibt die Schritte aus

#include <iostream>

using namespace std;


int main()
{
     int z[5];               //Array für die Elemente

     z[0]=5;
     z[1]=2;
     z[2]=6;
     z[3]=4;
     z[4]=1;

     //Ausgabe des Start-Arrays
     cout << "Am Anfang: " << endl;
     for (int index=0; index<5;index++)
     {
          cout << z[index] << "|";
     }
     cout << endl;


     //Schleife zum Wiederholen der Durchläufe
     for (int i=0;i<4;i++)
     {
          //Schleife für die Durchläufe
        for (int j=0;j<4-i;j++)
          {
               //Wenn kleiner, dann tauschen
```

```
                if (z[j]>z[j+1])
                {
                     cout << "Tausche "<<j<<" und "<<(j+1)<<endl;
                     int temp=z[j];
                     z[j] = z[j+1];
                     z[j+1] = temp;
                }

                //Aktuelles Array ausgeben
                for (int index=0; index< 5 ;index++)
                {
                     cout << z[index] << "|";
                }
                cout << endl;
           }
           cout << "====================" << endl;
      }
}
```

Aufgabe 4 – Sortieren von Arrays

```
//ArraySortAufgabe2.cpp - 19.08.2005 - Dominik Bruhn
//Sortiert ein Zufallsarray mit 100 Elementen und gibt es vor und nach
einer Sortierung aus

#include <iostream>
#include <ctime>

using namespace std;


int main()
{
     int count = 100;              //Speicher für die Anzahl der Elemente
     int z[100];                        //Array für die Elemente

     //Zufallsgenerator iniatisieren
     srand((unsigned)time(NULL));

     //Array mit Zufallszahlen
     for (int index=0;index<count;index++)
     {
          z[index] = rand() % 100; // Zahlen zwischen 0 und 99
     }

     //Ausgabe des Start-Arrays
     cout << "Am Anfang: " << endl;
     for (int index=0; index<count ;index++)
     {
          cout << z[index] << "|";
     }
     cout << endl;


     //Schleife zum Wiederholen der Durchläufe
```

```
        for (int i=0;i<count-1;i++)
        {
             //Schleife für die Durchläufe
           for (int j=0;j<count-i-1;j++)
             {
                  //Wenn kleiner, dann tauschen
                  if (z[j]<z[j+1])
                  {
                       int temp=z[j];
                       z[j] = z[j+1];
                       z[j+1] = temp;
                  }
             }
        }

        cout << "Am Ende: " << endl;
        for (int index=0; index< count ;index++)
        {
             cout << z[index] << "|";
        }
        cout << endl;
   }
```

Praktikum 4

Aufgabe 5 – Funktionen - Durchschnitt

```
   //FunktionAufgabe1.cpp - 17.08.2005 - Dominik Bruhn
   //Liest zwei Zahlen ein, übergibt diese an eine Funktion die den
   Durchschnitt berechnet und gibt diesen wieder aus.

   #include <iostream>

   using namespace std;

   //Funktion durchschnitt definieren
   float durchschnitt (int zahl1,int zahl2)
   {
        //Variable mit dem Durchschnitt füllen
        float schnitt = (zahl1+zahl2)/2.0f;

        //Durchschnitt zurückgeben
        return schnitt;
   }


   int main()
   {
        //Variablen
        int zahl1;
        int zahl2;
        float erg;

        //Zahlen von der Tastatur einlesen
        cout << "Bitte gib die erste Zahl ein: ";
```

```
        cin >> zahl1;

        cout << "Bitte gib die zweite Zahl ein: ";
        cin >> zahl2;

        //Bestätigung ausgeben
        cout << "Du hast folgende Zahlen eingegeben: " << zahl1 << " und "
    << zahl2 << endl;

        //Funktion aufrufen und deren rückgabe-Wert in erg speichern
        erg = durchschnitt(zahl1,zahl2);

        //Durchschnitt (erg) ausgeben
        cout << "Der Durchschnitt der Zahlen ist " << erg << endl;

        return 0;
    }
```

Aufgabe 6 – Funktionen - Teiler

```
    //FunktionAufgabe2.cpp - 17.08.2005 - Dominik Bruhn
    //Liest zwei Zahlen ein, übergibt diese an eine Funktion die prüft ob
    die zweite Zahl ein Teiler der ersten ist, gibt eine entsprechende
    Meldung aus

    #include <iostream>

    using namespace std;

    bool isTeiler(int zahl,int teiler)
    {
        // MOEGLICHKEIT 1:

        if (zahl % teiler == 0)
        {
            return true;
        }
        else
        {
            return false;
        }

        //MOEGLICHKEIT 2:

        int rest = zahl % teiler;
        bool isTeiler = (rest==0);
        return isTeiler;

        //MOEGLICHKEIT 3:

       return !(zahl % teiler);
    }


    int main()
```

```
{
	//Variablen
	int zahl;
	int teiler;
	bool erg;

	//Zahlen von der Tastatur einlesen
	cout << "Bitte gib die Zahl ein: ";
	cin >> zahl;

	cout << "Bitte gib den Teiler ein: ";
	cin >> teiler;

	//Bestätigung ausgeben
	cout << "Du hast " << zahl << " und den Teiler " << teiler << endl;

	//Funktion aufrufen und deren rückgabe-Wert in erg speichern
	erg = isTeiler(zahl,teiler);

	//Wenn erg == true:
	if (erg==true) {
		cout << teiler << " ist ein Teiler von " << zahl << endl;
	}
	else {
		cout << teiler << " ist KEIN ein Teiler von " << zahl << endl;
	}

	return 0;
}
```

Aufgabe 7 – Funktionen - Quersumme

```
//FunktionAufgabe3.cpp - 17.08.2005 - Dominik Bruhn
//Liest eine Zahl ein, übergibt diese an eine Funktion, die die
Quersumme berechnet, und gibt diese wieder aus

#include <iostream>

using namespace std;

int quersumme(int zahl)
{
	int summe = 0;
	int stelle= 0;
	int rest = zahl;

	// MOEGLICHKEIT 1:
	while (rest>0) {
		stelle = rest % 10;
		rest = rest / 10;
		summe += stelle;
	}

	return summe;
```

```
        // MOEGLICHKEIT 2:

        while (rest>0) {
            summe += rest % 10;
            rest /= 10;
        }

        return summe;
    }

    int main()
    {
        int eingabe;

        cout << "Bitte gib die Zahl ein: ";
        cin >> eingabe;


        //Quersummen ausgeben
        int qs = quersumme(eingabe);
        cout << "Die Quersumme von " << eingabe << " ist " << qs << endl;


        return 0;
    }
```

Praktikum 5

Aufgabe 1:

```
    //main.cpp - 26.08.2005 - Anton Trojosky
    //BruchKlasse

    #include <iostream>

    using namespace std;

    class Bruch
    {
    private:
        int zaehler;
        int nenner;

    public:
        Bruch(void)
        {
            zaehler = 0;
            nenner = 1;
        }
        Bruch(int newZaehler)
        {
            zaehler = newZaehler;
            nenner = 1;
        }
        Bruch(int newZaehler, int newNenner)
```

```
     {
          zaehler = newZaehler;
          nenner = newNenner;
     }
     void setZaehler(int newZaehler)
     {
          zaehler = newZaehler;
     }
     bool setNenner(int newNenner)
     {
          if (newNenner != 0)
          {
               nenner = newNenner;
               return true;
          }
          else
          {
               return false;
          }
     }
     void print(void)
     {
          cout << "Bruch = " << zaehler << "/" << nenner << endl;
     }
     bool isGanzzahl(void)
     {
        if (zaehler % nenner == 0)
          {
               zaehler /= nenner;
               nenner = 1;
               return true;
          }
          else
          {
               return false;
          }
     }
};

int main()
{
     Bruch meinBruch;
     int zahl;

     cout << "Bitte geben sie einen Zaehler ein: ";
     cin >> zahl;
     meinBruch.setZaehler(zahl);
     cout << "Bitte geben sie einen Nenner ein: ";
     cin >> zahl;
     meinBruch.setNenner(zahl);

     meinBruch.print();
     if (meinBruch.isGanzzahl())
     {
          cout << "Der Bruch ist eine Ganzzahl." << endl;
```

```
            meinBruch.print();
        }

        return 0;
    }
```

Praktikum 6

Aufgabe 1:

```
    #ifndef kreis_h
    #define kreis_h

    #include "graphclass.h"

    class Kreis
    {
    private:
        int x;
        int y;
        int radius;

    public:
        Kreis(int newX,int newY,int newRadius);
        void draw(graphclass *graph);
    };
    #endif

    #include "Kreis.h"
    #include "graphclass.h"

    Kreis::Kreis(int newX,int newY,int newRadius)
    {
        x = newX;
        y = newY;
        radius = newRadius;
    }

    void Kreis::draw(graphclass *graph) {
        graph->circle(BLUE,x,y,radius);
    }
    #ifndef quadrat_h
    #define quadrat_h

    #include "graphclass.h"

    class Quadrat
    {
    private:
        int x;
        int y;
        int width;

    public:
        Quadrat(int newX,int newY,int newWidth);
```

```
     void draw(graphclass *graph);
};
#endif

#include "Quadrat.h"
#include "graphclass.h"

Quadrat::Quadrat(int newX,int newY,int newWidth)
{
     x = newX;
     y = newY;
     width = newWidth;
}

void Quadrat::draw(graphclass *graph) {
     graph->rectangle(GREEN,x,y,x+width,y+width);
}
#ifndef rechteck_h
#define reckteck_h

#include "graphclass.h"

class Rechteck
{
private:
     int x;
     int y;
     int width;
     int height;

public:
     Rechteck(int newX,int newY,int newWidth,int newHeigth);
     void draw(graphclass *graph);
};
#endif

#include "Rechteck.h"
#include "graphclass.h"

Rechteck::Rechteck(int newX,int newY,int newWidth,int newHeight)
{
     x = newX;
     y = newY;
     width = newWidth;
     height = newHeight;
}

void Rechteck::draw(graphclass *graph) {
     graph->rectangle(RED,x,y,x+width,y+height);
}

#include "graphclass.h"

#include "quadrat.h"
#include "kreis.h"
```

```
#include "rechteck.h"


int main() {
          graphclass graph(500,500);

          //(X,Y,Kantenlänge)
          Quadrat q1(30,40,50);
          Quadrat q2(300,100,250);

          //(X,Y,Radius)
          Kreis k1(100,230,80);
          Kreis k2(230,123,150);

          //(X,Y,Breite,Höhe)
          Rechteck r1(100,200,380,230);
          Rechteck r2(435,43,50,450);

          q1.draw(&graph);
          q2.draw(&graph),

          k1.draw(&graph);
          k2.draw(&graph);

          r1.draw(&graph);
          r2.draw(&graph);

          graph.getch();
}
```

Praktikum 7

```
// GraphObj.h - 26.8.2005 - Roman Keller
// Headerdatei der Basisklasse GraphObj
#ifndef GraphObj_h
#define GraphObj_h

#include "graphclass.h"

class GraphObj
{
protected:
     int x;
     int y;
     int farbe;

     GraphObj(int newFarbe, int newX, int newY);

public:

     void setFarbe(int newFarbe);

     void move(int newX, int newY);
};
#endif //GraphObj_h
```

```
// GraphObj.cpp - 26.8.2005 - Roman Keller
// Quellcodedatei für Basisklasse GraphObj

#include "GraphObj.h"

GraphObj::GraphObj(int newFarbe, int newX, int newY)
{
     farbe = newFarbe;
     x = newX;
     y = newY;
}

void GraphObj::move(int newX, int newY)
{
     x = newX;
     y = newY;
}
// Kreis.h - 26.8.2005 - Roman Keller
// Headerdatei der Klasse Kreis
#ifndef Kreis_h
#define Kreis_h

#include "GraphObj.h"

class Kreis : public GraphObj
{
protected:
     int radius;

public:

     Kreis(int farbe, int newX, int newY, int newRadius);

     void draw(graphclass * graph);
};
#endif //Kreis_h
// Kreis.h - 26.8.2005 - Roman Keller
// Quellcodedatei der Klasse Kreis

#include "Kreis.h"
Kreis::Kreis(int newFarbe, int newX, int newY, int newRadius)
:GraphObj(newFarbe, newX, newY)
{
     radius = newRadius;
}
void Kreis::draw(graphclass * graph)
{
     graph->circle(farbe,x,y,radius);
}
// Rechteck.h - 26.8.2005 - Roman Keller
// Headerdatei der Klasse Rechteck
#ifndef Rechteck_h
#define Rechteck_h
```

```
#include "GraphObj.h"

class Rechteck : public GraphObj
{
protected:
     int width;
     int height;

public:

     Rechteck(int newFarbe, int newX, int newY, int newWidth, int
newHeight);

     void draw(graphclass *graph);

};
#endif //Rechteck_h
// Rechteck.h - 26.8.2005 - Roman Keller
// Quellcodedatei der Klasse Rechteck

#include "Rechteck.h"

Rechteck::Rechteck(int newFarbe, int newX, int newY, int newWidth, int
newHeight)
:GraphObj(newFarbe,newX,newY)
{
     width = newWidth;
     height = newHeight;
}

void Rechteck::draw(graphclass * graph)
{
     graph->rectangle(farbe,x,y,x+width,y+height);
}
//Quadrat.h - 26.8.2005 - Roman Keller
// Headerdatei für Quadrat


#ifndef Quadrat_h
#define Quadrat_h

#include "Rechteck.h"

class Quadrat : public Rechteck
{
public:
     Quadrat(int newFarbe, int newX, int newY, int newWidth);
};
#endif //Quadrat_h
// Quadrat.h - 26.8.2005 - Roman Keller
// Quellcodedatei der Klasse Quadrat

#include "Quadrat.h"

Quadrat::Quadrat(int newFarbe, int newX, int newY, int newWidth)
```

```
:Rechteck(newFarbe,newX,newY,newWidth,newWidth)
{
}
// main.cpp - 26.8.2005 - Roman Keller
//Hauptprogramm

#include "Kreis.h"
#include "Rechteck.h"
#include "Quadrat.h"
#include "graphclass.h"


int main()
{
     graphclass graph(500,500);

     Kreis kreis1(RED,200,200,100);

     kreis1.draw(& graph);

     Rechteck rechteck1(GREEN,100,100,200,300);

     rechteck1.draw(& graph);

     Quadrat quadrat1(YELLOW,200,200,100);

     quadrat1.draw(& graph);

     graph.getch();
}
```