Informatik

~~C++~~ für Nicht-Informatiker

Jörg Schake

SoSe 2019

Informatik für Nicht-Informatiker
Jörg Schake
Hochschule Darmstadt
Fachbereich Informatik
Birkenweg 7
64295 Darmstadt

cppbuch@offline-student.de

Stand: März 2020

1 Vorwort

Bücher zu C++ gibt es wie Sand am Meer, auf Papier genauso wie Online. Warum dann noch eins schreiben?

Die Antwort ist einfach: in vielen Jahren in der Programmierausbildung am Fachbereich Informatik der Hochschule Darmstadt habe ich immer wieder festgestellt, dass Studierende sich auf die Angaben in den diversen Flyern und Infozetteln verlassen und tatsächlich ein Studium in einem technischen Fach beginnen, ohne jemals zuvor auch nur eine Zeile programmiert zu haben.

Denn so steht es in nahezu allen Infomaterialien der verschiedenen Fachbereiche drin: Programmierkenntnisse (oder auch andere Vorkenntnisse) sind nicht erforderlich! Da darf man den Erstsemestern auch keinen Vorwurf machen, wenn sie zum ersten Mal in einem PC-Labor sitzen sich beim Programmieren schwer tun, auch wenn es sich bei der ersten Übung in der Regel nur um ein einfaches Abtippen (oder sogar nur Kopieren) von ein paar Programmzeilen handelt.

Die Vorlesung Programmieren hastet durch den Stoff, der Dozent drückt aufs Tempo, weil er in der nächsten Übung ja auch etwas Schönes programmieren lassen möchte. Es beginnt ein Wettlauf zwischen Hase und Igel, den man als echter Programmieranfänger nur verlieren kann.

Unter Informatikern ist bekannt: *Programmieren lernt man nur durch Programmieren!* Also hilft auch nur das Programmieren, also Üben, Üben, Üben. Und hier unterscheidet sich das vorliegende Buch von anderen Werken, ich versuche, in möglichst kleinen Schritten vorzugehen, die leicht nachvollziehbar sind und sich sofort praktisch überprüfen lassen. Es findet sich also in den ersten Kapiteln immer die Vorstellung eines neuen Themas und dann ein paar Übungen dazu, die das soeben Beschriebene vertiefen und anwenden.

Zu den Aufgaben gibt es keine Lösungen! Denn jeder Leser kann selbst ganz schnell überprüfen, ob das, was er/sie dort eingetippt hat, auch das tut, was erwartet wurde. Die Addition zweier Zahlen kann man in der Regel noch im Kopf kontrollieren, beim Sinus muss man sich vielleicht schon etwas überlegen. Aber auch das gehört dazu: Wer ein Programm schreibt, muss sicherstellen, dass es genau das tut, was verlangt wurde.

Der ursprüngliche Titel dieses Buchs lautete ja *C++ für Nicht-Informatiker*, allerdings werden wir von dem ++ im Titel nicht sehr viel behandeln. Ich kümmere mich hier im wesentlichen um die Grundlagen der Programmierung, die auch bei C++ später ganz selbstverständlich benutzt werden, aber in den Vorlesungen meist nur kurz vorgestellt werden, um dann *richtige* Programme schreiben zu können.

Wir nutzen einige der Techniken von C++ immer dann, wenn sie gleiche oder bessere Möglichkeiten bieten wie die klassische C-Variante, z.B. bei der Ein- und Ausgabe. Damit ersparen wir uns auch das spätere Umlernen. Wer also reines C lernen möchte (wie die Mechatroniker im 1. Semester) wird an manchen Stellen ein zusätzliches Buch zu Rate ziehen müssen.

Zum SoSe 2019 habe ich ein zusätzliches Kapitel „Fehler: Verweis nicht gefunden“ ergänzt, darin sind viele Punkte gesammelt, die über das Semester verteilt in der Vorlesung auftauchen und das Programmieren erst zur Informatik machen. Und im C++-Teil gibt es jetzt einen kleinen Einblick in die STL, die vor allem den Elektrotechnikern unter den Lesern den Übergang ins zweite Semester und GIT erleichtern soll.

Geeignet ist dieses Werk für Studierende von Fachbereichen, die die Informatik im 1. und/oder 2. Semester als Pflichtfach im Studienprogramm haben und dabei auf die Programmiersprache C oder C++ setzen. Aktuell sind das an der h_da die Fachbereiche EIT (Elektrotechnik und Informationstechnik), MN (Mathematik) und MK (Maschinenbau und Kunststofftechnik). Für Informatiker sei an dieser Stelle auf die

Standardwerke (siehe Literaturverzeichnis) verwiesen, weil hier eine andere Herangehensweise gefordert ist und Stoffumfang und -tiefe deutlich über das hier Gebotene hinaus gehen.

Die Reihenfolge des Stoffs ist nicht unbedingt identisch mit dem Vorgehen eines Dozenten in der Vorlesung. Insbesondere bei den höheren Kapiteln ist es sehr von persönlichen Vorlieben und Geschmack abhängig, in welcher Reihenfolge diese Themen behandelt werden. Wer also dieses Skript parallel zu einer Vorlesung benutzt und dann das ein oder andere Kapitel überspringt oder erst später bearbeitet steht dann manchmal vor dem Problem, dass er in den Übungsaufgaben mit (noch) unbekannten Ideen und Fragestellungen konfrontiert wird. Das ist aber kein Beinbruch, in der Informatik gibt es immer mehrere Wege, die zu einer Lösung führen. Aber man kann das ja auch als Ansporn sehen und das fragliche Kapitel schon vor der Vorlesung durcharbeiten. Dann kann man in der Vorlesung sogar mit Genuss und einem innerlichen Grinsen Zwischenfragen stellen!

Bei der schrittweisen Einführung neuer Themen lässt es sich leider nicht immer vermeiden, dass Begriffe benutzt werden die bei ihrer ersten Verwendung noch nicht bekannt sind. Auf den ersten Seiten mache ich darauf noch aufmerksam, danach nicht mehr. Es kommt der Zeitpunkt, an dem sich auch der letzte unbekannte Begriff klärt. Den ganz Eiligen sei an dieser Stelle ein Blick in das Glossar (auf Seite 193) empfohlen.

Einige Abschnitte sind als „weitergehend…“ markiert, hier stehen dann die Informationen, die Sie zumindest beim ersten Durchblättern überlesen sollten. Holen Sie das dann nach, wenn das entsprechende Kapitel in der Vorlesung behandelt oder im Praktikum benötigt wird.

Die gezeigten Beispielprogramme und Listings stammen aus verschiedenen Entwicklungsumgebungen, daraus ergeben sich unterschiedliche Farben beim Syntaxhighlighting.

Ein solches Skript lebt mit mit und von seinen Lesern. Ich freue mich daher, Kritik, Anregungen, Wünsche, Fehlermeldungen und Verbesserungsvorschläge zu erhalten. Senden Sie diese einfach an cppbuch@offline-student.de.

Informatik oder Programmieren?

Inhalte eines Studienganges unterliegen ständiger Fortschreibung, ‚Änderung und Anpassung, die sich in aktualisierten Modulhandbüchern widerspiegelt. Auch das Fach „Informatik“ hat vor kurzem vom Fb EIT eine Änderung in „Einführung in die Programmierung“ erfahren. Dabei wurde nicht nur der Titel geändert, sondern einige Inhalte entfernt, andere hinzugefügt oder vertieft und damit der Schwerpunkt noch weiter in Richtung Programmieren verlagert. Dies wirkt sich natürlich auf die Inhalte der Vorlesung aus, die sich an der Modulbeschreibung ausrichtet. Ich habe daher den klassischen Teil über „Informatik“ nach hinten verschoben, gleichzeitig aber neu gefasst und überarbeitet. Dieses Kapitel kann also jetzt tatsächlich komplett ignoriert werden, wenn es in der Vorlesung auch nicht behandelt wird. Umgekehrt darf man es aber durchaus lesen, weil dort ein paar Hintergründe aufgehellt werden, die einem das Programmieren einfacher machen.

2 Inhalt

Falls Sie sich zunächst einen Überblick verschaffen wollen, was hier wie in welchem Zusammenhang behandelt wird, sollten Sie diese Seite hier in Angriff nehmen. Sind Sie der Typ für klassische Inhaltsverzeichnisse, blättern Sie um. Zur leichteren Navigation sind die angegebenen Seitenzahlen klickbar (je nach PDF-Viewer mehr oder weniger gut zu erkennen).

2.1 Inhaltsübersicht

Kap. 2

Kapitel 7: Ablaufsteuerung

Daran schließt sich mit Kapitel 7 das Thema **Ablaufsteuerung** an. Es geht um ein-, zwei- und mehrseitige **Auswahlen** (S. 61), kopf- und fußgesteuerte **Schleifen** (S. 69) und verbotene **Sprünge** (S. 73). Ihre Programme können jetzt **Entscheidungen** treffen und mit **Wiederholungen** nerven.

Kapitel 8: Strukturierte Programmierung

Kapitel 8 bringt **Struktur** in die Programme. Sie erfahren (fast) alles über **Funktionen** (S. 76), **Parameter** (S. 78), **Referenzen** (S. 80) und **Prototypen** (S. 79). Außerdem lernen Sie etwas über **Gültigkeitsbereiche** (S. 85) und **lokale**, **globale** und statische Variablen.
Es geht aber noch einen Schritt weiter, für größere Projekte verwenden Sie mehrere Dateien und **verteilen** die Arbeit auf mehrere Personen und/oder auf **mehrere Dateien** (S. 88).

Kapitel 9: Zusammengesetzte Datentypen

Weil das Zerlegen und Zusammenfügen nicht auf Programmteile beschränkt ist, beschäftigt sich Kapitel 9 mit **zusammengesetzten Datentypen**, an erster Stelle das **Array** (S. 97) für eine Reihe von gleichen Datentypen, auch in **mehreren Dimensionen** (S. 99). In diese Kategorie fallen auch die diversen **Zeichenketten** (S. 100), die bisher zwar schon fleißig verwendet, aber noch nie erläutert wurden.
Über die **Strukturen (struct**, S. 103) kommen wir zu der **Ein- und Ausgabe** mittels **Textdateien** (S. 105), um auch große Datenmengen und **unterschiedlichste Datentypen** verarbeiten zu können.
Den Abschluss bildet der Abschnitt über **Zeiger** und ihre Nutzung (S. 108), speziell bei der Verarbeitung von **Listen** (S. 112).

Kapitel 10: Weitere Sprachelemente und ausgewählte Bibliotheksfunktionen

Es gibt noch ein paar Themen, die in keines der anderen Kapitel passten. So finden sich im Kapitel 10 Hinweise zu dem **Template-Funktionen** (S. 119), **Namensräumen** (S. 120), der **Fehlerbehandlung** mit try und catch (S. 121) und eine Sammlung von **Bibliotheksfunktionen** (S. 121), die man immer mal brauchen kann.

Kapitel 11: Fallstudie Zahlenrätsel

Das Kapitel 11 ist etwas Besonderes. Hier wird einmal eine **komplette Übung** von der Aufgabenstellung bis zur fertigen Lösung vorgestellt. Ja, sie lesen richtig, hier gibt es tatsächlich eine Lösung! Diese **Fallstudie** empfiehlt sich als **Selbstkontrolle** etwa nach der Hälfte des Semesters, denn dann sollten Sie das notwendige Wissen haben und diese Aufgabe lösen können. Wenn Sie dabei größere Schwierigkeiten haben, ist entweder mehr Zeitaufwand für das Fach Informatik notwendig oder eine Überprüfung, ob es nicht sinnvoller ist, im nächsten Semester einen neuen Versuch zu starten und sich dafür auf andere Fächer zu konzentrieren (siehe dazu auch „Wenn das Praktikum (und/oder die Vorlesung) nicht so laufen wie geplant“ auf Seite 19)
Außerdem finden sich in diesem Kapitel noch jede Menge **weitere Übungen** (ab Seite 140, natürlich ohne **Lösungen**), an denen Sie Ihr Wissen überprüfen können.

Kapitel 12: Informatik im Schnelldurchgang

Dieses Kapitel ist ein Opfer der Änderungen in der Modulbeschreibung. Hier geht es um die „andere" Seite der Informatik, die nichts oder nur sehr wenig mit dem Programmieren zu tun hat. Sie lernen die verschiedenen **Teilgebiete** der Informatik Theoretische Informatik (S. 150), Technische Informatik (S. 150), Praktische Informatik (S. 151) und Angewandte Informatik (S. 151) kennen. Wir werfen einen Blick auf die Konsequenzen, die die Informatik auf die Gesellschaft hat (S. 152) und widmen uns dann dem Aufbau von Rechenanlagen (S. 153). Danach wird es etwas theoretischer, es beginnt mit **bool'scher Algebra** (S. 155), der **Aussagenlogik** (S. 155) und den Rechenregeln der bool'schen Algebra auf Seite 156.

Achtung, auf Seite 161 wird es dann elektrisch oder elektronisch , es drohen (digitale) **Grundschaltungen**. Die erleichtern aber das Verständnis für den Aufbau eines Rechners.

Das Dualsystem beschäftigt uns weiter, wir erfahren, wie man Zahlen von einem ins andere **Zahlensystem** umwandeln kann (S. 168) und wie Zahlen im Rechner gespeichert werden (S. 177). Den Abschluss bildet ein Abschnitt über die **Architektur** von Rechner (S. 175) und einen kurzen Überblick über die darauf laufende **Software** (S. 180).

Kapitel 13: Über den STeLLerrand geblickt ist in er Version 2019 neu hinzugekommen. Hier wagen wir einen Blick auf die Fähigkeiten von C++, die in der Vorlesung selten dran kommen, aber einem das Leben als Programmierer doch sehr erleichtern. Das sind exemplarisch die Containerklassen **Vektor** (S. 183) und **Liste** (S. 184) und ausgewählte Funktionen der STL (S. 185) sowie **Iteratoren** (S. 184) und der Datentyp **auto** (S. 182).

Kapitel 14 bis 17: Literatur, Programmierhäppchen, Glossar und ASCII-Tabelle

Den Abschluss bilden hilfreiche Seiten zum **Nachschlagen**. Hier findet sich eine Auswahl an **Literatur** (Kapitel 14), wenn Sie noch mehr wissen wollen; eine kleine **Sammlung von kurzen Programmteilen,** die man immer mal wieder brauchen kann (Kapitel 15) sowie ein **Glossar** (Kapitel 16) und die obligatorische **ASCII-Tabelle** (Seite 195).

2.2 Inhalt (tabellarisch)

3 Lernstrategien für Programmieranfänger

Dieses Kapitel suchen Sie in anderen Lehrbüchern vergeblich, es ist auch nicht allgemeingültig sondern speziell auf den Lehr- und Übungsbetrieb der h_da zugeschnitten. Und es steht zu Beginn, weil es von grundlegender Bedeutung für das Studieren im allgemeinen ist und nicht nur beim Programmieren seine Berechtigung hat.

3.1 Lerngruppe suchen oder gründen

Das A und O eines erfolgreichen Studiums liegt in der Wahl bzw. dem Aufbau einer gut funktionierenden Lerngruppe. Die ideale Größe sollte bei fünf bis sechs Personen liegen, perfekt ist eine Mischung aus Personen mit unterschiedlichen Vorbildungen und Wissensgebieten. Der Wissensvorsprung von Einzelnen darf dabei nicht allzu groß werden und es muss sicher gestellt sein, dass jeder mal von seinem Wissen abgibt aber auch jeder vom Wissen der anderen profitiert.

Die Lerngruppe sollte sich an einem neutralen Ort (leerer Hörsaal, Bibliothek, o.ä) treffen, die Wohnung eines Gruppenmitgliedes ist eher hinderlich (wegen Heimvorteil und der möglichen Ablenkungen).

3.2 Aufgabe besprechen

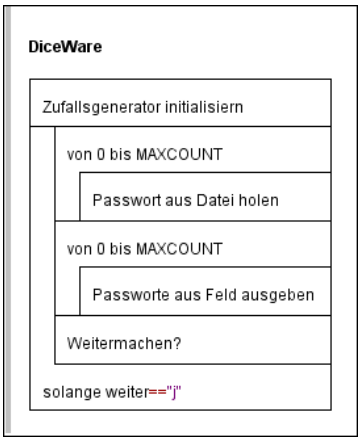
Lesen Sie in der Gruppe die Aufgabenstellung genau, evtl. auch laut oder mehrfach, durch, bis wirklich jedem klar ist, worum es geht. Bei unterschiedlichen Interpretationen diskutieren Sie und halten Sie dann fest, welches Interpretation der Gruppe am wahrscheinlichsten erscheint. Kommen Sie hier zu keinem Ergebnis, scheuen Sie sich nicht, Kontakt zum Dozenten aufzunehmen (am besten per Mail oder über das Diskussionsforum in Moodle). Er weiß am besten, was gemeint ist und hat ein großes Interesse daran, dass die Aufgabe in seinem Sinne verstanden wird oder das Problem wurde bereits von Kommilitonen erfolgreich gelöst

Diese Lerngruppe muss sich nicht auf ein Fach beschränken, am besten funktioniert sie, wenn man diese Lerngruppe in so vielen Fächern wie möglich einsetzt.

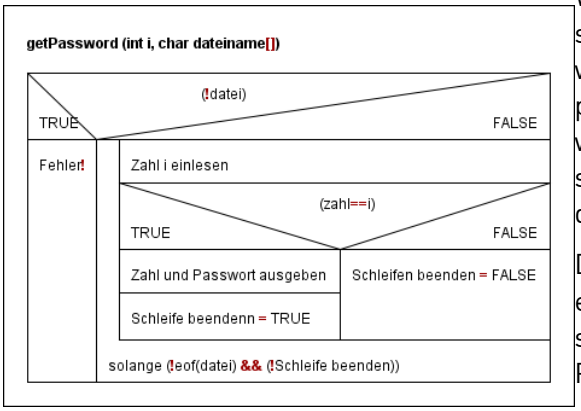
Hier beschränken wir uns allerdings auf den Einsatz im Fach Informatik (oder Programmieren).

3.3 Lösungswege erarbeiten

Für diesen Schritt brauchen Sie keinen Rechner. Im Gegenteil, er ist kontraproduktiv. Besprechen Sie in der gesamten Lerngruppe eine oder mehrere Lösungen, Lösungswege oder Lösungsansätze dieser Aufgabe. Bei diesem Brainstorming hilft eine Tafel (falls vorhanden) oder zumindest ein Stapel Papier auf dem Tisch. Wer als Erster weiß, wie die Lösung aussieht macht sich selbst zum Protokollanten und hält seine Ideen zunächst erst einmal zurück.



Zerlegen Sie die Aufgabe[1] (bzw. die Lösung) in einzelne Schritte, die jeweils einen Teil der Aufgabe lösen. Diese Einzelschritte lassen sich gut als Struktogramm oder Flußdiagramm oder auch textuell als Pseudocode dokumentieren. (Die Fallstudie im Kapitel 11 zeigt das an einem konkreten Beispiel.)



Wenn das Gesamtprogramm grob zerlegt ist, nehmen Sie sich jeden dieser Schritte einzeln vor und zerlegen ihn wiederum in einzelne Schritte. So wird das Gesamtproblem systematisch verfeinert und gelöst. Das Ganze wiederholen Sie solange, bis nur noch einzelne Konstrukte aus dem C++-Baukasten übrig bleiben. Damit hat die Lerngruppe in diesem Falle ihre Aufgabe erfüllt.

Dieses Verfahren nennt sich Top-Down-Entwurf und ist eine der gängigsten Verfahren, um größere Projekte zu stemmen (nicht nur beim Programmieren). Je kleiner ein Problem zerlegt wurde, desto leichter ist es zu lösen.

3.4 Umsetzung in ein Programm

Nach der Zerlegung in die einzelnen Konstruktionselemente setzt sich jeder Student alleine an einen Rechner und erstellt zunächst aus den zerlegten Teilproblemen ein Programm aus Kommentaren, die den besprochenen Ablauf wiedergeben. Danach wird aus jedem Kommentar ein Stück Programm (die Kommentare bleiben drin!). Natürlich klappt das nicht auf Anhieb, aber für den eigenen Lernfortschritt ist es unumgänglich, selbst das Programmieren zu üben und im Umgang mit dem System vertraut zu werden.

Teile, die partout nicht funktionieren wollen, bleiben als Kommentar eingefügt.

Danach trifft man sich mit seinem Praktikumspartner und stellt sich gegenseitig die umgesetzte Lösung vor (wer keinen Praktikumspartner hat, macht das mit einem anderen Mitglied der Lerngruppe).

Wer keinen Rechner zur Verfügung hat, verfeinert seinen Entwurf auf Papier und bereitet sich so optimal darauf vor, das Programm während des Praktikums einzugeben.

3.5 Finale Lösung erarbeiten

Aus diesen beiden Lösungen sollte sich eine gemeinsame Lösung entwickeln lassen. Diese Lösung darf noch Lücken und Fehler enthalten, die können dann gezielt im Praktikum vor Ort mit den Betreuern und Tutoren angegangen werden. Dabei hilft es dann, wenn man die Unterlagen, die zu dieser Lösung geführt haben (also die Papierentwürfe) vorliegen. Die zeigen den Betreuern nämlich zum einen, dass Sie sich gut vorbereitet haben und zum anderen, an welchen Stellen es Probleme gibt. Erwarten Sie aber nicht, dass der Stoff der Vorlesung extra für Sie noch einmal wiederholt wird, der sollte also sitzen und bekannt sein!

```
/*
 * File:  main.cpp
 * Author: Joerg
 */
#include <iostream>
using namespace std;

int main() {
  std::cout << "Hallo Welt!" << endl;
  return 0;
}
```

1 Bitte beachten Sie: Die Illustrationen in diesem Abschnitt haben symbolischen Charakter und stehen untereinander in keinem Zusammenhang

3.6 Externe Hilfe zuziehen

Es ist keine Schande, sich bei Problemen externe Hilfe z.B. durch Informatik-Studierende zu besorgen. Aber: nutzen Sie solche Berater nur als allwissendes Lexikon, nicht als Ghostwriter. Fragen Sie ihren Berater Löcher in den Bauch, aber achten Sie darauf, dass seine Antworten zu ihrem Kenntnisstand passen. Es ist niemanden damit geholfen, wenn Sie Lösungen präsentiert bekommen (und dann später vorführen), die mit dem Vorlesungsstoff so überhaupt nicht zu lösen sind. Es geht ja nicht darum, die Aufgabenstellung irgendwie zu lösen, sondern so, wie man sie mit dem Wissen aus der Vorlesung her lösen kann (und muss). Das ist etwas, was solchen Beratern nicht leicht fällt, weil sie es gewohnt sind, mit allen zur Verfügung stehenden Mitteln eine Aufgabe zu lösen. So würde kein Informatiker ernsthaft daran gehen, einen Haufen Daten mittels eines selbst geschriebenen Programms zu sortieren, dafür gibt es fertige Bibliotheken. In solchen Fällen muss er sich eben mal daran zurück erinnern, wie das in seiner ersten Vorlesung war, da durfte er diese Bibliothek nämlich auch noch nicht benutzen. Und übrigens: Ihre Praktikumsbetreuer erkennen solche Lösungen auf einen Blick, die sind ja selber Informatiker. Und mit zwei oder drei gezielten Fragen stecken Sie in Erklärungsnöten. Lässt sich aber mit dem hier beschriebenen Verfahren leicht vermeiden.

3.7 Mit den Vorbereitungen ins Labor

Hier haben jetzt alle, die zu Hause keinen Rechner zur Verfügung haben einen kleinen Vorteil, sie kommen mit Papier und müssen ihre Lösung noch schnell zum Laufen bringen, das Eintippen ist dank aktueller Entwicklungsumgebungen heute nicht mehr das Problem (Codevervollständigung und Co. helfen, Fehler zu vermeiden).

Wer bereits eine (angefangene) Lösung hat, muss diese ja trotzdem irgendwie auf den Labor-PC bekommen. Wegen unterschiedlicher Versionen führt das regelmäßig zu Problemen, selbst innerhalb der selben Produktfamilie (Visual Studio 2013 kann die Dateien von Visual Studio 2017 nicht lesen). Da hilft ein ganz schlichtes Vorgehen: Kopieren der Quelltexte.

Erstellen Sie auf dem Laborrechner ein neues (leeres!) Projekt und fügen Sie eine leere Datei hinzu. Dann öffnen Sie die mitgebrachte Datei in einem normalen Editor (z.B. Notepad++), markieren dort mit Strg-A den gesamten Quellcode und kopieren ihn dann mit Strg-C in die Zwischenablage. Jetzt wechseln Sie in Ihr neues Projekt und fügen in der leeren Datei mittels Strg-V den ganzen Text ein. Voila!. Besteht das Projekt aus mehreren Dateien, wiederholen Sie dieses Verfahren für jede Datei. Besteht das Projekt aus mehreren Dateien, wiederholen Sie diese Prozedur für jede einzelne Datei. Eventuell müssen Sie noch die Dateinamen bei den #include-Anweisungen anpassen, wenn die Dateien nicht exakt so benannt sind wie im Ausgangsprojekt. Danach einfach das Projekt neu übersetzen und es sollte genauso laufen wie vorher zu Hause. Das klappt mit kompletten Projekten, aber auch mit einzelnen Dateien oder kurzen Programmstücken.

Der umgekehrte Weg ist einfacher, kopieren Sie dann einfach nur die einzelnen Quelltext-Dateien aus dem Projektverzeichnis auf ihren Stick oder senden Sie die per Mail an sich selbst. Wenn Sie zu Hause noch an dem Projekt weiter arbeiten möchten, verfahren Sie analog, neues Projekt aufmachen, leere Datei(-en) anlegen und den Quelltext per Copy & Paste einfügen.

Auf diese Weise können Sie auch Dateien aus früheren Projekten in ein neues Projekt überführen, wenn Übungen auf bereits vorhandenen Lösungen aufbauen und diese erweitern. Legen Sie dazu auch ein neues, leeres Projekt an, achten Sie dabei insbesondere darauf, dass für die Dateinamen geeignete Namen gewählt werden und nicht einfach nur die vorgegebenen Standardnamen verwendet werden. So sollten die Dateien im Projekt „Aufgabe5“ nicht `aufg4.cpp` heißen. Das macht die spätere Fehlersuche sehr viel einfacher.

Probleme treten auch auf, wenn Text aus einem PDF in eine Quelltextdatei kopiert wird. Je nach verwendetem Browser, PDF-Viewer und anderen Umständen gehen dabei die Zeilenumbrüche verloren und müssen mühsam wiederhergestellt werden. Hier lohnt es sich, verschiedene Browser zu probieren, um ein optimales Ergebnis zu erzielen.

Ähnliches gilt für typografische Zeichen in PDF-Dokumenten. Manche Textverarbeitungen ändern eigenmächtig einzelne Zeichen in eingefügten Programmlistings ab. Bei Anführungszeichen „…“ ist das noch gut zu erkennen, beim Minuszeichen meckert dann der Compiler.

Vermeiden Sie es, die bestehenden Quelltext-Datei(-en) zum neuen Projekt **hinzuzufügen**, das hat bisher immer zu Komplikationen geführt. Mit dem hier beschriebenen Vorgehen klappt sogar der Wechsel der Entwicklungsumgebung (z.B. von VisualStudio zu NetBeans) ohne Fehler.

3.8 Die (kaputte) Zweiergruppe

Rechnerplätze im Labor sind begrenzt, deswegen gehen die Planungen und Einteilungen der Übungsgruppen immer von zwei Studierenden pro Rechnerplatz aus. Das spart nicht nur Ressourcen, sondern hat für die Studierenden den Vorteil, dass man auch nur die Hälfte einer Aufgabe erledigen muss, die andere Hälfte macht ja der Partner.

Träumen Sie weiter.

Dieser Idealzustand kann nur funktionieren, wenn sich beide Praktikumspartner regelmäßig treffen und sich gemeinsam auf die Übung vorbereiten. Leider klappt das in der Praxis nicht wie man sich das so vorstellt.

Daher ist jeder Studierende selbst dafür verantwortlich, sich auf die Aufgaben vorzubereiten und sie im Ernstfall auch alleine durchführen zu können. Dieser Ernstfall tritt schneller auf, als man glaubt, denn da reicht schon eine ausgefallene S-Bahn, und man sitzt ohne Partner im Labor. Da ist es dann ziemlich blöd, wenn der gerade nicht verfügbare Partner die gesamte Vorbereitung bei sich trägt und man nicht dran kommt. Ihr Betreuer ist da gnadenlos, schlechte oder fehlende Vorbereitung kann einen ganz schnell den Schein für das Labor kosten.

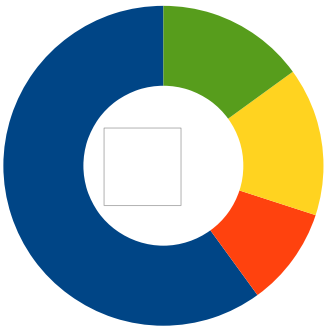
Beugen Sie solchen Eventualitäten auf jeden Fall vor! Wenn Sie sich gemeinsam vorbereiten, kopieren Sie die Vorbereitungen so, dass jeder Partner einen kompletten Satz hat. Treffen Sie sich nur im Praktikum, achten Sie darauf, dass jeder Partner am Ende des Praktikums alle erarbeiteten Lösungen und Unterlagen hat. Insbesondere dann, wenn Aufgaben aufeinander aufbauen, ist das lebenswichtig.

In den Laboren des Fachbereichs Informatik empfiehlt sich folgendes Vorgehen: während des Praktikums wird mit den Zugangsdaten (Account) eines beliebigen Praktikumsteilnehmers gearbeitet. Nach dem Praktikum werden die wichtigen Dateien (also die *.cpp und die *.h-Dateien) aus dem Projekt auf die lokale Platte D: kopiert (man könnte auch von Anfang an das Projekt dort anlegen, das macht es noch einfacher). Jetzt meldet sich der aktuelle Benutzer ab und der verbliebene Praktikumspartner meldet sich mit seinen Zugangsdaten an. Dann die auf der Platte D: liegenden Dateien in das Homeverzeichnis (Eigene Dateien…) kopieren und fertig ist die Sicherheitskopie für den nächsten Übungstermin.

Wer die Dateien auch zu Hause haben möchte, kann sie natürlich nach diesem Schema auch auf einen USB-Stick kopieren, sich selbst per Mail schicken oder in der Cloud ablegen. Es muss nur sicher gestellt sein, dass jeder Partner einen kompletten Satz hat und nach 2 Wochen noch weiß, wie man ohne Partner dran kommt.

3.9 Was muss man (selbst zusätzlich) lernen?

Es gibt einen gravierenden Unterschied zwischen Lernen in der Schule und Lernen im Studium: der Anteil an eigenverantwortlichen Wissenserwerb steigt stark an. Mit anderen Worten: es wird nicht mehr jedes Detail und jede Variante eines Themas in der Vorlesung behandelt, sondern oft nur ein allgemeiner Aspekt mehr oder weniger intensiv vorgestellt. Alles weitere muss man sich selbst erarbeiten, durch Lesen von Literatur, Anschauen von Tutorials und natürlich durch Programmieren. Denn nur beim Anwenden des Stoffes auf ein konkretes Problem oder eine bestimmte Aufgabenstellung stößt man auf die Punkte, die in der Vorlesung nicht zur Sprache kamen oder nur eine geringe Rolle spielten.



Es lohnt sich also durchaus, zu Beginn des Semesters einen Besuch in der Bibliothek zu machen und sich das ein oder andere Buch näher anzusehen und gegebenenfalls auch mit nach Hause zu nehmen. Dort sollte man aber auch regelmäßig hinein schauen, sonst war die Aktion vergeblich.

Wer besser mit Tutorials klar kommt muss aber den dort dargebotenen Stoff auch noch einmal unterfüttern, sonst bleibt es beim oberflächlichen Verständnis für den Stoff. Richtig sitzt der Stoff grundsätzlich erst nachdem man ihn sich praktisch, also in Form von selbst geschriebenen Programmen, erarbeitet hat.

3.10 Die CP (ECTS) und der Zeitaufwand

Für alle Studiengänge an Hochschulen gibt so etwas wie Lehrpläne an den Schulen. Das nennt sich dann Modulhandbuch bzw. Modulbeschreibung. Darin wird unter anderem festgelegt, wie viel „Gewicht" einer Lehrveranstaltung im Studium beigemessen wird. Diese Gewichtung findet in CP nach dem ECTS[2] statt und ist eine Zahl. Viele Lehrveranstaltungen sind mit 5 CP bewertet und beinhalten 2 bis 3 SWS Vorlesung und ebensoviele Praktikumsanteile. Konkret steht im Modul B04 (Bachelor E-Technik[3]) die Veranstaltung „Informatik" mit 5 CP bei 2 SWS Vorlesung und 2 SWS Praktikum. Und im Laufe des Textes steht auch noch, dass diese 5 CP sich auf 150 Stunden (Zeitstunden, keine SWS!) verteilen. Der Rest ist einfache Mathematik: in den 16 Wochen des Semesters (da sind die Prüfungswochen mit drin!) kommt man so auf 4 SWS * 16 Wochen * 0,75 (1 SWS=45 Minuten) auf 48 Stunden, die durch die Teilnahme an der Lehrveranstaltung erfüllt sind. Fehlen nur noch 102 Stunden, um den Plan zu erfüllen. Wenn man die auf das gesamte Semester gleichmäßig verteilt, bleiben da pro Woche noch 102/16 = 6,375 Stunden/Wochen zusätzlicher Aufwand offen. Man könnte auch einfach so rechnen: für jede Stunde, die Sie in Vorlesung oder Praktikum an der Hochschule verbringen (sollten), müssen Sie zwei Stunden zusätzlich zu Hause oder in der Lerngruppe aufbringen. Und es wird noch schlimmer: ein Semester umfasst 30 CP, die also zusammen rechnerisch eine gesamte Arbeitszeit von 900 Stunden im Semester ergeben. Und das sind dann 16 Wochen mit je 50 Stunden oder (ohne Prüfungswochen gerechnet) 900/14=65 Stunden pro Woche. Fürs Geld verdienen und Freizeit bleibt da nicht mehr viel Zeit, um so wichtiger ist es, sich das rechtzeitig klar zu machen und entsprechend vom ersten Tag an am Ball zu bleiben.
Studieren ist ein Vollzeit-Job!

2 https://de.wikipedia.org/wiki/European_Credit_Transfer_System

3 Das Modulhandbuch im Internet war zum Zeitpunkt der Überarbeitung des Skriptes noch auf dem Stand von 2017!. http://www.fbeit.h-da.de/fileadmin/EIT/Dokumente/Elektrotechnik_BA/Modulhandbuch_EIT_PO2013_Anlage5.pdf

3.11 Wenn das Praktikum (und/oder die Vorlesung) nicht so laufen wie geplant

Zunächst einmal ist es keine Katastrophe, wenn es in einem Fach nicht auf Anhieb so läuft, wie man sich das vor Beginn des Studiums gedacht hat. Entscheidend ist, dies rechtzeitig zu erkennen. Daher richtet sich dieser Abschnitt in erster Linie an die Studierenden, die ihre Praktikumsaufgaben nicht komplett selbst lösen (können) und dabei auf immer größere Unterstützung durch Dritte zurückgreifen müssen. Das gipfelt dann darin, dass ab der vierten oder fünften Übung (von sechs) keinerlei Eigenleistung mehr erbracht wird, sondern eine fertige Lösung ins Labor mitgebracht wird und als eigene Arbeit vorgeführt wird.

Die Betreuer im Labor erkennen dies aber auf einen Blick, und man ist mit einer oder zwei geschickten Fragen ertappt. Natürlich kann man das abstreiten und vehement darauf hinweisen, dass man zwar Hilfe hatte, aber das trotzdem alles alleine gemacht hat. Das ist aber Selbstbetrug! Dem Betreuer ist das vollkommen gleichgültig, er muss die Klausur nicht bestehen! Und die Klausur bestehen Sie nicht, wenn Sie die Praktikumsaufgaben nur vorgeführt, aber nicht selbst gelöst haben.

Überlegen Sie sich also ganz genau, wie sie sich verhalten, wenn es in einem Fach klemmt und nur noch externe Unterstützung die Rettung ist. In der Regel hat man den Faden ja schon einige Zeit vorher verloren und macht sich selbst nur noch etwas vor. Seien Sie mutig und beenden Sie dieses grausame Spiel selbst. Ziehen Sie die Reißleine und versuchen Sie das Fach ein oder zwei Semester später erneut. Nur so haben Sie den Kopf frei für die anderen Fächer und können sich diesen verstärkt widmen. Nutzen Sie die so gewonnene Zeit für die Vor- und Nachbereitung der übrigen Lehrveranstaltungen und erzielen dort dann auch bessere Ergebnisse.

Versuchen Sie aber auf keinen Fall, eine Lehrveranstaltung zu belegen, die auf einer abgebrochenen Veranstaltung aufbaut (also etwa Mathe 2 beginnen, obwohl man in Mathe 1 aufgegeben hat oder GIT ohne Einführung in das Programmieren/Informatik angehen). Das kann nicht funktionieren, der Stoff aus der Grundlagenveranstaltung muss sitzen!.Planen Sie also ihr nächstes Semester entsprechend und kümmern Sie sich rechtzeitig um die Wiederholung der abgebrochenen Praktika oder Vorlesungen. Das ist vor allem dann wichtig, wenn Plätze automatisch vergeben werden oder Sie mit Überschneidungen im Stundenplan rechnen müssen.

Bei der Wiederholung kann es von Vorteil sein, einen neuen Dozenten zu wählen, um einen anderen Vorlesungsstil zu erfahren. Aber selbst, wenn man es beim gleichen Dozenten noch einmal versucht, ist es ratsam, auch die bereits bestandenen Teile noch einmal zu besuchen, auch wenn man das (angeblich) ja alles schon einmal gehört oder gemacht hat (warum hatten Sie die Veranstaltung abgebrochen?). Selbst wenn Sie z.B. das Informatik-Labor bestanden haben und Ihnen nur noch die Klausur fehlt, kann ich nur empfehlen, das Praktikum noch einmal (als Gast) zu besuchen und sich praktisch mit der Thematik zu befassen: Programmieren lernt man nur durch Programmieren!!!

4 Entwicklungsumgebungen

Man kann Programme auf einem Bierdeckel entwerfen und mit Hilfe von Schaltern in einen Rechner eingeben, das hat man in der Computersteinzeit[4] auch so gemacht. Heute nutzt man moderne Entwicklungsumgebungen, um Programme schnell, effizient und sicher zu schreiben.

Eine solche *Entwicklungsumgebung* (auch IDE für Integrated Development Enviroment genannt) bietet unter einer Oberfläche alles, was man als Programmierer so braucht:

- Einen *Editor* zum eigentlichen Erstellen der Programme. Dabei handelt es sich um eine spezielle Version einer Textverarbeitung, die darauf ausgelegt ist, lange Texte ohne Schnickschnack zu erfassen und zu bearbeiten. Man findet darin Funktionen, die zusammengehörige Klammern suchen, die komplette Bereiche auf- oder zuklappen und Bereiche einrücken. Das Suchen und Ersetzen von Text spielt eine große Rolle und kann vielfältig angewendet werden.
- Eine Projektübersicht, in der alle an dem Programm beteiligen Dateien und ihre Abhängigkeiten dargestellt werden, um die Übersicht nicht zu verlieren. Unsere Projekte werden fast immer nur aus einer oder einigen wenigen Dateien bestehen, die Umgebung muss aber auch mit großen Projekten und entsprechenden Anzahlen von Dateien klar kommen.
- Einen *Navigator*, der die Übersicht auf Klassen- und Funktionsebene darstellt (was das ist, werden wir noch sehen).
- Ein *Ausgabefenster*, in dem die Ausgaben unseres Programms dargestellt werden oder die Fehler, die wir beim Programmieren gemacht haben.
- Steuerelemente, die die vielen externen Helferlein starten, stoppen und steuern, die sich hinter dem englischen Wort Integrated verbergen. Dazu gehören der eigentliche *Compiler*, *Linker* und der *Debugger*. Auch dazu später mehr.

Am Fachbereich Informatik der h_da werden in den Praktika vorrangig zwei unterschiedliche Entwicklungsumgebungen eingesetzt. Das hängt von den Vorlieben des jeweiligen Dozenten ab. Für unsere Aufgaben ist es völlig belanglos, für welche dieser Umgebungen man sich entscheidet.

Es gibt aber viele weitere Entwicklungsumgebungen unter Windows, ebenso wie für andere Betriebssysteme auch, auf die ich hier nicht weiter eingehen möchte. Sie alle erfüllen den gewünschten Zweck und haben ihre Vor- und Nachteile. Im Laufe der Zeit lernt man aber mit diesen Vor- und Nachteilen zu leben und möchte dann die einmal erlernte Umgebung nur sehr ungern wechseln.

E-Techniker können außerhalb des Labors auch mit Eclipse arbeiten, das die Basis für die Veranstaltung GIT im 2. Semester bildet. Um im Labor dann mit einer der dort installierten Entwicklungsumgebungen arbeiten zu können, kann man wie im Kapitel „Mit den Vorbereitungen ins Labor“ vorgehen.

Im späteren Berufsleben werden Sie ständig die Entwicklungsumgebungen wechseln, es kann sich also nur lohnen, das rechtzeitig zu üben.

Im Folgenden beschreibe ich exemplarisch eine der gängigen Entwicklungsumgebungen und versuche, Ihnen zu vermitteln, wie man damit umgeht und zu (schnellen und brauchbaren) Ergebnissen kommt. Dabei gibt es mal wieder ein Henne-Ei-Problem: ohne C++-Kenntnisse kann man die Umgebungen schlecht erläutern, ohne die Kenntnis der Entwicklungsumgebungen ist das Programmieren sehr viel

4 Einen kleinen Einblick in die Frühgeschichte der Informatik und die dabei verwendeten Geräte gibt es im 3. Stock des Informatik-Gebäudes D14

schwieriger. Die abgebildeten Programme sind daher extrem kurz und übersichtlich, um nicht vom eigentlichen Thema abzulenken. Dafür gibt es viele Bilder, die das beschriebene anschaulicher machen sollen. Dazu kommt im praktischen Einsatz die Sprachverwirrung, manche Umgebungen kann man deutsch oder englisch installieren, andere sind immer englisch und die dritte Variante übernimmt die Sprache des laufenden Betriebssystem. Ich habe daher versucht, beim ersten Verwenden eines Begriffs immer die jeweils andere Sprache in Klammern hinzuzufügen, um Einsteigern den Start zu erleichtern.

In den früheren Versionen dieses Skriptes folgte hier ein Kapitel über die IDE NetBeans. Diese unterstützt seit der Version aus dem Herbst 2019 die Programmiersprache C++ aber nicht mehr offiziell, daher beschränke ich mich auf Visual Studio

4.1 Visual Studio (Version 2019)

Visual Studio Professional wird von Microsoft vertrieben und läuft daher nur unter den aktuellen Windows-Betriebssystemen. Dafür wird hier in einem Paket alles Notwendige installiert, um Programme in einer Fülle von Programmiersprachen zu entwickeln. Aktuell (Frühjahr 2020) ist die Version 2019, die mit kleinen Verbesserungen gegenüber der Version 2017 aufwartet. Die Grafiken und Screenshots in diesem Skript wurden mit der Version 2017 erstellt, leichte Abweichungen sind daher möglich.

Allerdings hat der Fachbereich Informatik beschlossen, in den Laboren nur noch die englischen Versionen einer Software einzusetzen, das gilt sowohl für das Betriebssystem (Windows 10) als auch für die eingesetzten IDE. Daher unterscheiden sich die Bilder hier und die im Labor angetroffene Wirklichkeit mindestens in der Sprache.

Als Student der h_da hat man die Möglichkeit, die aktuelle Enterprise-Version[5] über das Microsoft-Azure-Programm kostenlos zu beziehen, diese ist nahezu identisch mit der Community-Edition, die jedermann kostenlos bei Microsoft herunterladen kann (diese Version erfordert zur dauerhaften Benutzung die Verwendung eines Microsoft-Kontos, die Enterprise-Version benötigt eine Seriennummer, die man beim Download aus dem Azure-Programm automatisch mitgeteilt bekommt))

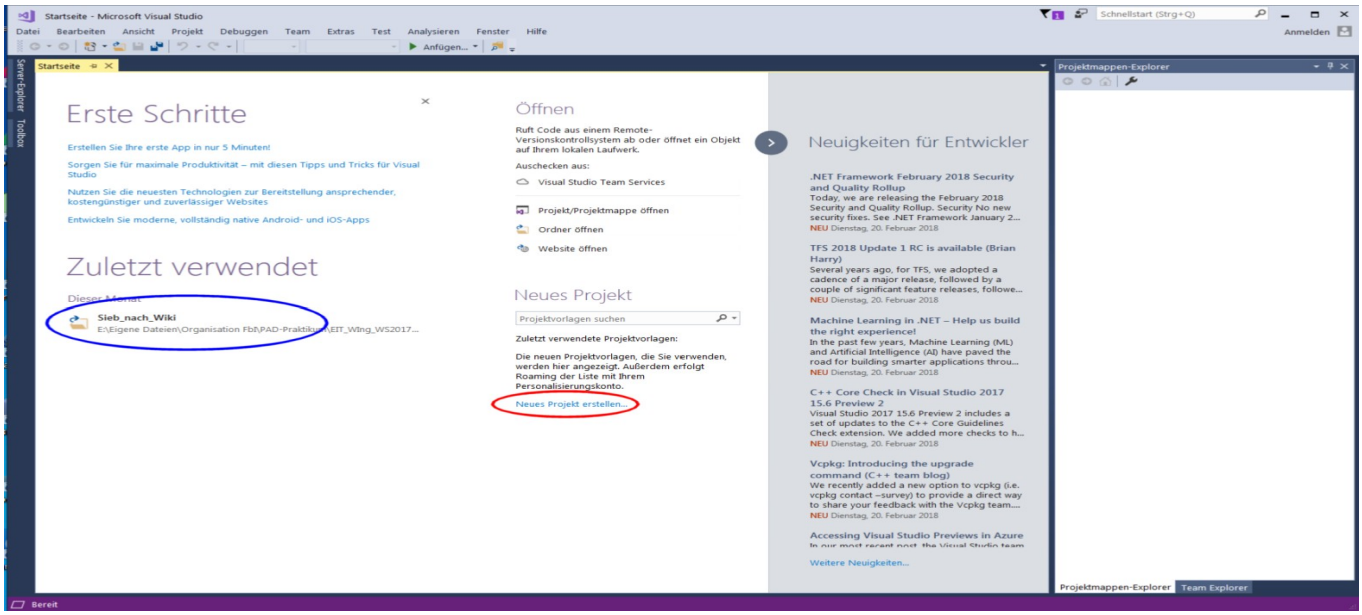
Microsoft ändert sporadisch die Bezeichnungen der Produkte innerhalb des Azure-Programms, so wurde im Jahre 2018 statt der Enterprise-Version eine Education-Version angeboten. Diese Versionen unterscheiden sich aber nur unwesentlich an Punkten, die im normalen Einsatz sowieso keine Rolle spielen. Nehmen Sie einfach, was gerade angeboten wird. In der Regel bekommen Sie so nur das Gerüst einer IDE, beim ersten Start müssen Sie dann noch die passenden Compiler nachladen.

Ältere Versionen: Wenn Sie eine ältere Version (etwa die von 2013) bereits auf ihrem Rechner vorfinden, ist das Inhaltlich kein Problem, aber technisch. Da die alte Version noch nicht wissen konnte, wie die aktuelle Version (oder jede andere, neuere Version) ihre Daten ablegt und welche Formate dabei zum Einsatz kommen, ist es zwar möglich, Projekte der alten Version unverändert mit der aktuellen Version zu öffnen und zu bearbeiten, aber leider nicht umgekehrt. Aber auch hier hilft das in 3.7 „Mit den Vorbereitungen ins Labor“ beschriebene Verfahren aus der Klemme.

5 Am 2. April 2019 erschien die Version 2019, Änderungen ergeben sich bei der Auswahl der Projekte und mit zusätzlichen Möglichkeiten beim Debugging. An der Sprache C++ wurden keine Änderungen vorgenommen

4.1.1 Der erste Start

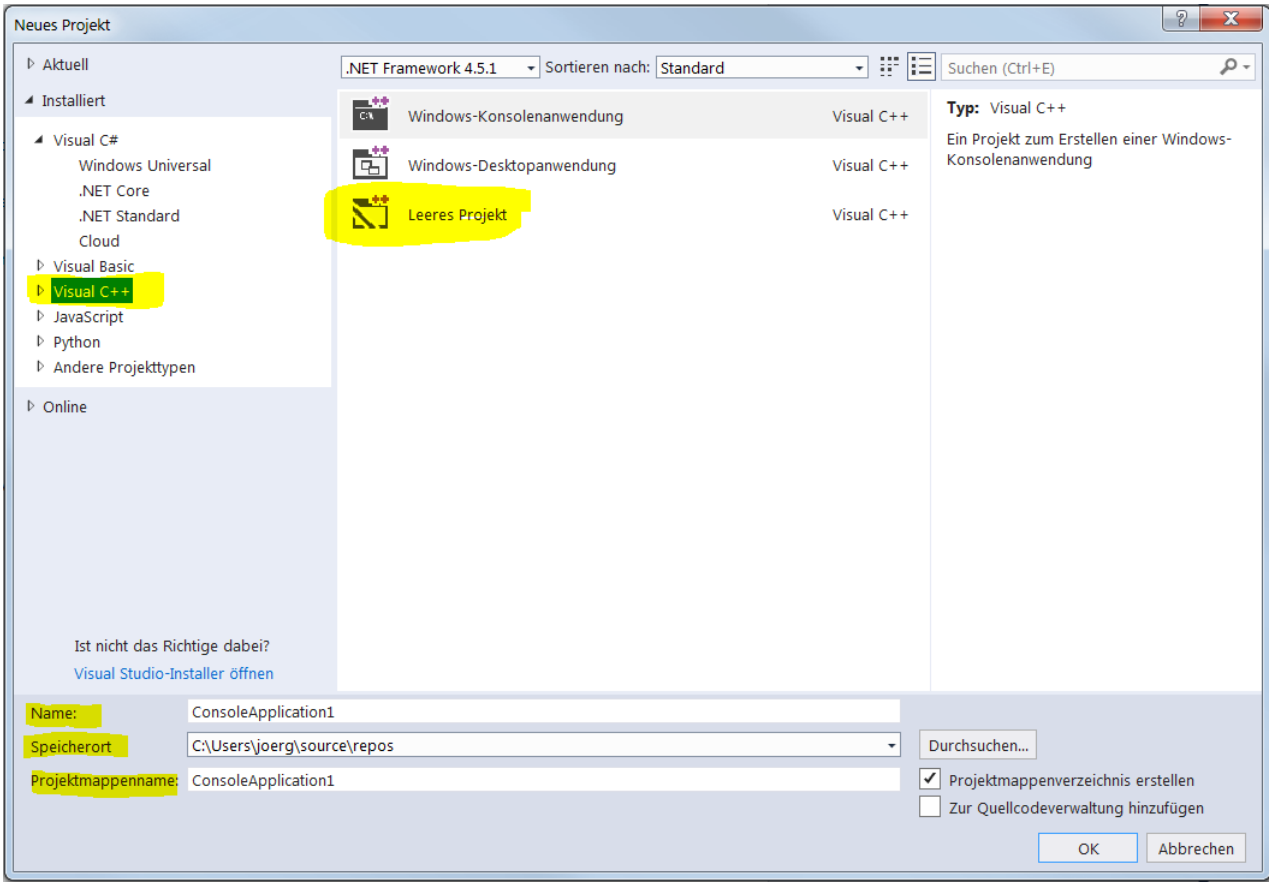
Ein frisch installiertes Visual Studio begrüßt seinen zukünftigen Benutzer so:



Man sieht, dieses Visual Studio ist nicht mehr ganz so frisch wie versprochen, denn es gibt schon einen Eintrag unter „Zuletzt verwendet“. Beim wirklich allerersten Start hat man nur die Chance, ein „Neues Projekt erstellen“ zu wählen.

Der Begriff „Projekt“ ist bei Microsoft so zu verstehen, dass jedes noch so kleine, aber auch jedes sehr große Programm jeweils ein eigenes Projekt darstellt. Wir müssen also selbst für unsere relativ geringen Anforderungen immer den Mehraufwand eines Projektes treiben. Dank sinnvoller Voreinstellungen ist das aber sehr schnell erledigt und geht einem ab dem zweiten oder dritten Projekt in Fleisch und Blut über.

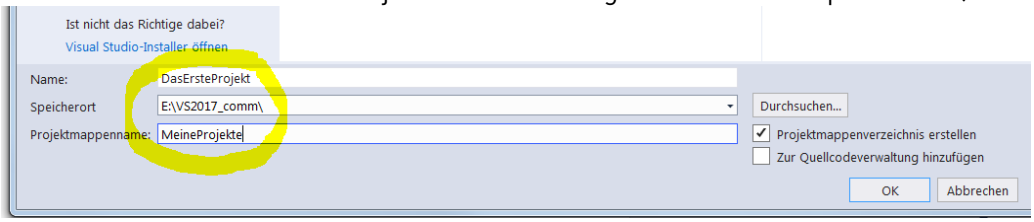
Wählt man „Neues Projekt erstellen“, erhält man folgenden Dialog (der je nach Installation insbesondere im linken Teil auch ein wenig anders aussehen kann). Wichtig sind die Einstellungen und Eingaben, die im Bild farbig hervorgehoben sind:



Ganz links unter „Aktuell“ muss zunächst die richtige Programmiersprache eingestellt werden, wir wollen ja C++ programmieren und nichts anderes. Fehlt C++ in dieser Auswahl, kann man die Sprache jederzeit über den „Visual Studio Installer“ hinzufügen (dazu muss man über eine funktionierende Internet-Verbindung verfügen).

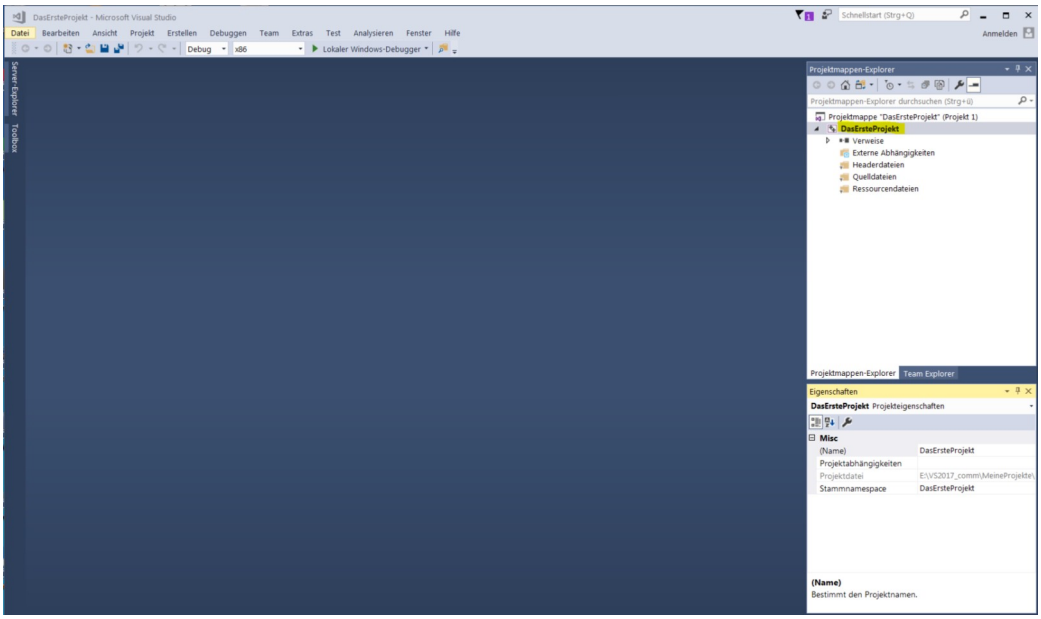
Im rechten Fenster werden die zur ausgewählten Programmiersprache passenden Projekttypen aufgelistet. Für die Windows-Desktopanwendung muss man schon etwas Erfahrung mitbringen, das heben wir uns für das dritte oder vierte Semester auf. Für Anfänger eignen sich sowohl die Windows Konsolenanwendung (Console Application) und das leere Projekt (empty Project). Ich empfehle an dieser Stelle immer, mit einem leeren Projekt zu beginnen, weil man dann sämtliche Einstellungen unter Kontrolle hat.

Im unteren Bereich muss man seinem Projekt einen Namen geben und einen Speicherort (Location)



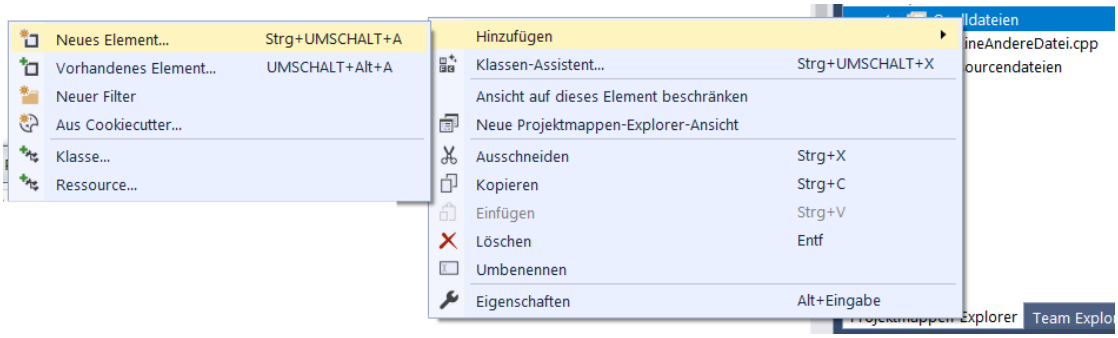
festlegen. Alle Dateien, die mit unseren Projekten zu tun haben, werden unterhalb dieses Speicherortes abgelegt. Zum Schluss kann man auch noch einen Projektmappennamen (Solution name) vergeben, das ist der Name für die Gesamtheit aller unserer Projekte, üblicherweise wird hier einfach der Projektname übernommen. Eine mögliche Einstellung für ein leeres Projekt sieht so aus:

Alle wichtigen Einstellungen sind jetzt vorgenommen, nach ein wenig Bedenkzeit (die für das Anlegen der Ordnerstrukturen benötigt wird) erscheint die arbeitsbereite Entwicklungsumgebung:



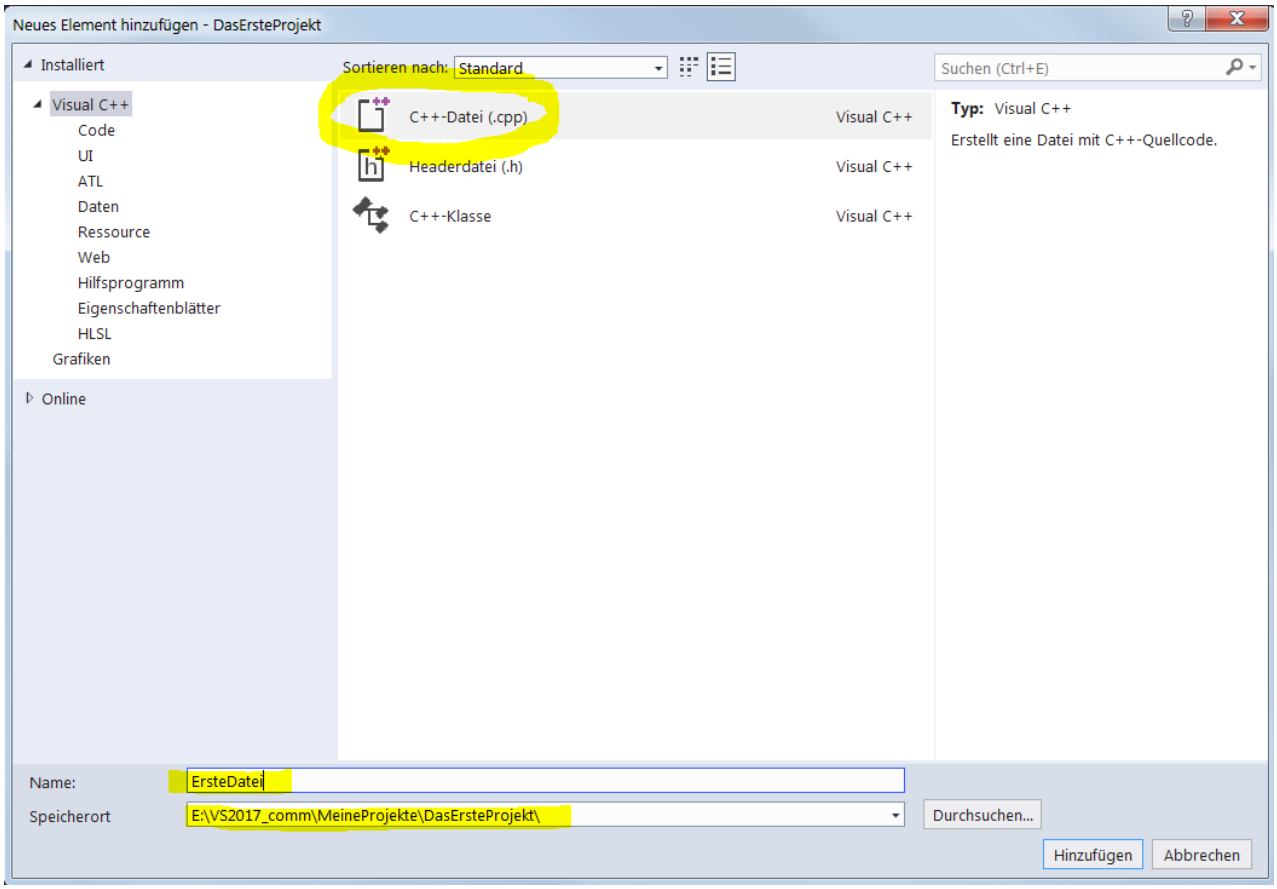
Recht oben (oder auch links oben, je nach Einstellung!) sehen wir den Projektmappenexplorer (Solution Explorer), den Namen unserer Projektmappe (Solution), den wir im vorherigen Dialog mühsam eingegeben haben, wurde übrigens nicht übernommen. Wer sich daran stört, kann das jetzt hier durch einen Klick mit der rechten Maustaste und „Umbenennen" korrigieren. Daher steht der Name unseres Projektes jetzt an zwei Stellen, weil ich das unterlassen habe.

Der größte Bereich der IDE ist noch leer, das ändert sich, wenn man dem leeren Projekt durch einen Rechtsklick auf „Quelldateien" (Source Files) eine neue, leere Datei hinzufügt:



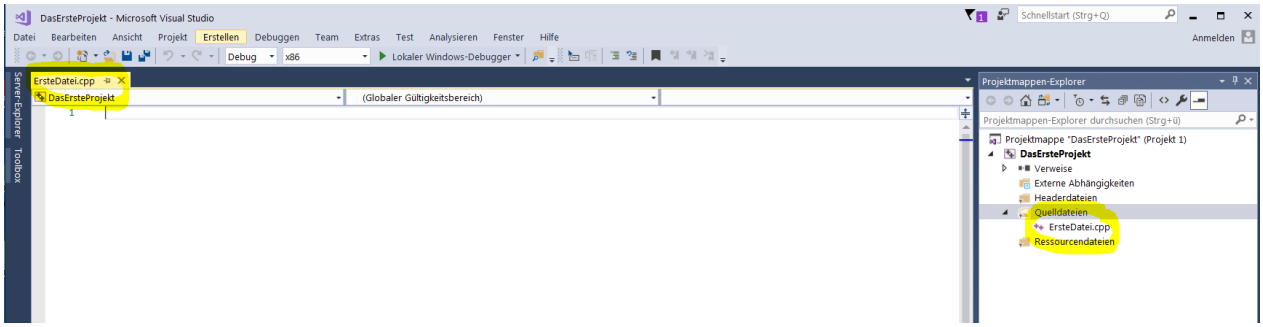
An dieser Stelle ist die Windows Konsolenanwendung im Vorteil, da wurde bereits eine (fast) leere Datei dem Projekt hinzugefügt. Probleme mit diesem Projekt treten erst später auf, wenn man sie am wenigsten brauchen kann. Es soll aber auch schon Fälle gegeben haben, die keinerlei Probleme mit der Console Application hatten

Durch den Start mit einem leeren Projekt können wir selbst bestimmen, wie unsere Datei heißen soll und machen davon auch Gebrauch:



Wichtig ist der **Typ** der Datei, für die ersten Programmierversuche kommt nur die C++-Datei in Frage (die Headerdatei muss bis zur dritten oder vierten Übung warten und die C++-Klasse sogar bis zum zweiten Semester).

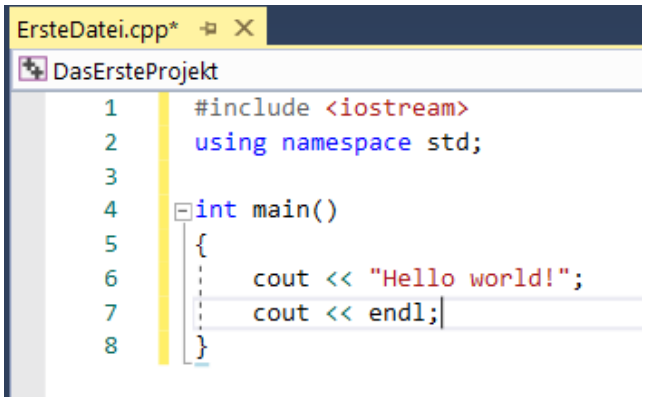
Der Name der Datei sollte mit Bedacht und Nachdenken gewählt werden. Er sollte etwas über den Inhalt der Datei aussagen und sich so auch später leicht zuordnen lassen. Auf keinen Fall darf man den vorgeschlagenen Namen übernehmen! (Spätestens beim zweiten Projekt rächt sich das, weil plötzlich



zwei Dateien mit unterschiedlichen Inhalten den gleichen Namen haben und damit nicht mehr zu unterscheiden sind. Das führt zu interessanten Fehlern und Problemen im Praktikum). Auf jeden Fall kommt durch die hinzugefügte Datei mehr Licht in den dunklen Bereich der IDE:

Jetzt können Sie loslegen und endlich das aufgestaute C++-Wissen in den Editor tippen. In erster Näherung ist so ein Editor nichts anderes wie eine ganz gewöhnliche Textverarbeitung, nur dass

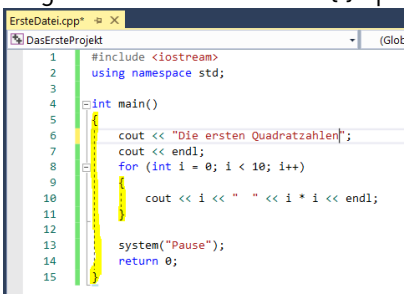
ausgerechnet am Layout keinerlei Änderungen möglich (oder sinnvoll) sind. Unser ersten Programm steht schon mal drin:



Auffällig ist, dass vollautomatisch Begriffe in bestimmten Farben dargestellt werden. Das erleichtert in größeren Programmen den Überblick unheimlich /das nennt man Syntax Highligthing).

Auf Anhieb nicht einsichtig ist das automatische Einrücken, aber wagen Sie es bitte nicht, diese Automatik von Hand außer Betrieb zu nehmen, weil Sie Texte gerne linksbündig formatieren. Diese Einrückung spiegelt den logischen Programmablauf wieder und hilft Ihnen (und vor allem ihren Betreuern im Praktikum), den Ablauf eines Programms auf einen Blick zu erkennen.

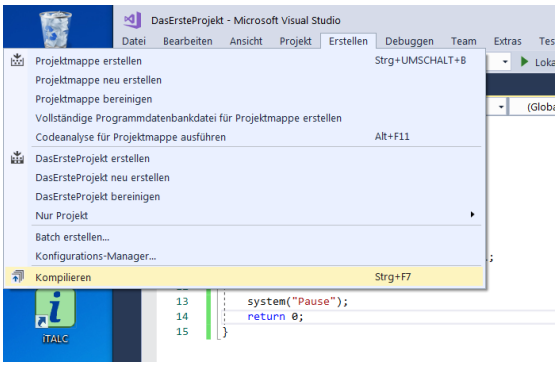
Die geschweifte Klammern { } spielen in C++ eine besonders große Rolle und treten (wie bei Klammern allgemein üblich) immer paarweise auf. Und weil ein besonders häufiger Fehler darin besteht, eine Klammer aus Versehen oder mit Absicht zu löschen oder an eine falsche Position zu schreiben, hilft die gestrichelte Linie dabei, die Klammern paarweise zuzuordnen.



In diesem Beispiel wurden jetzt sogar geschweifte Klammern geschachtelt (das kommt ständig vor, auch mit drei, vier oder noch mehr Ebenen), aber dank Hilfslinien und automatischer Einrückung bleibt die Übersicht erhalten. Fehlt also eine solche Hilfslinie oder endet aprupt haben Sie einen logischen Fehler im Programm, der bis zur Vorführung dann noch gesucht, gefunden und beseitigt werden muss.

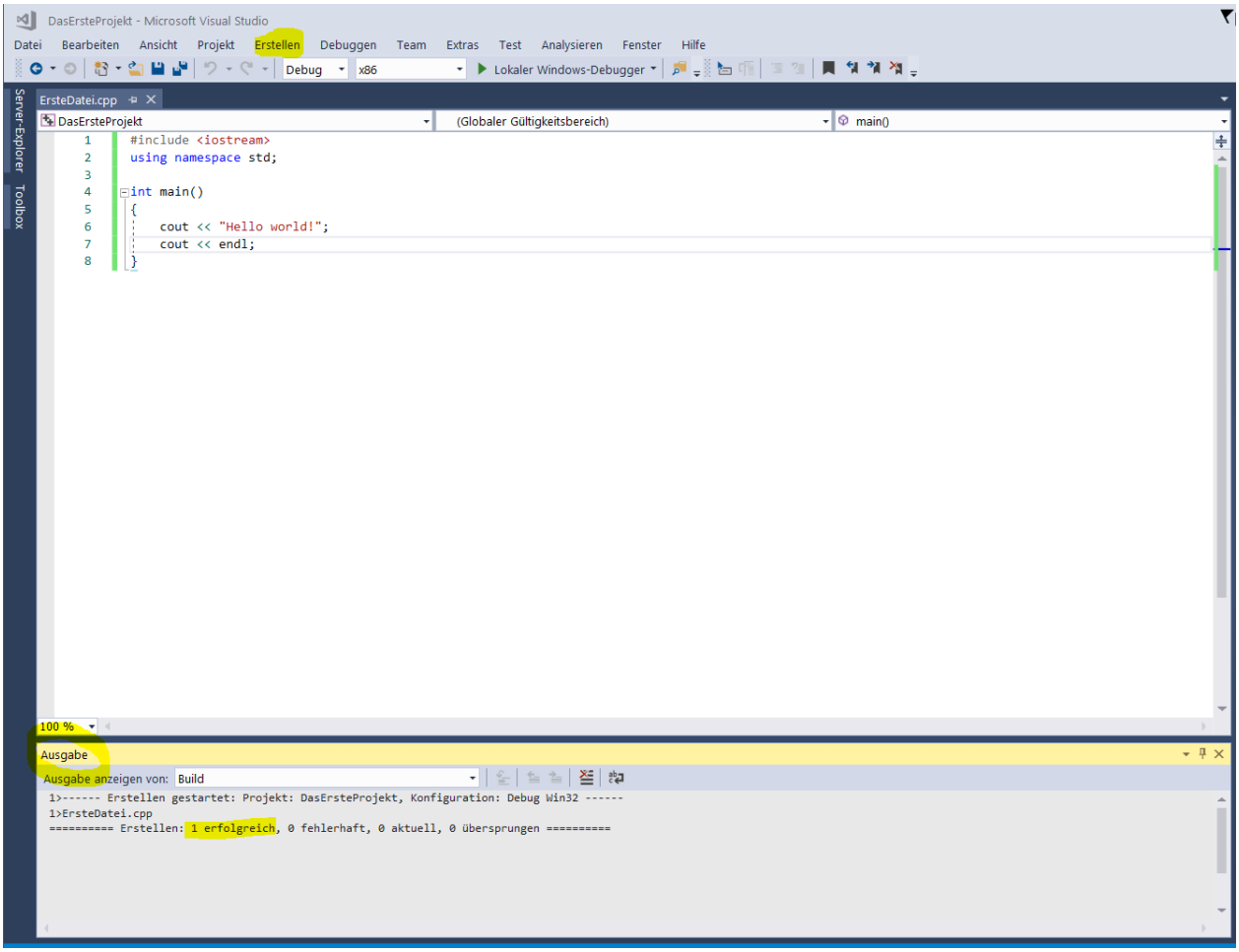
4.1.2 Übersetzen (kompilieren)

Irgendwann ist auch das längste Programm einmal in den Rechner eingegeben und dann beginnt die eigentliche Arbeit. Das Programm soll ja nicht nur im Rechner drin sein, sondern es soll auch laufen. Dazu sind zwei weitere Schritte erforderlich, die durch entsprechende Menüpunkte in der IDE aufgerufen werden.



Schritt 1 ist das Übersetzen unseres Quelltextes in die Maschinensprache des Computers. Dieser Vorgang heißt Kompilieren (engl. to compile). In Visual Studio versteckt sich dieser Schritt im Menü unter „Erstellen → Kompilieren“ oder der Tastenkombination Strg+F7:

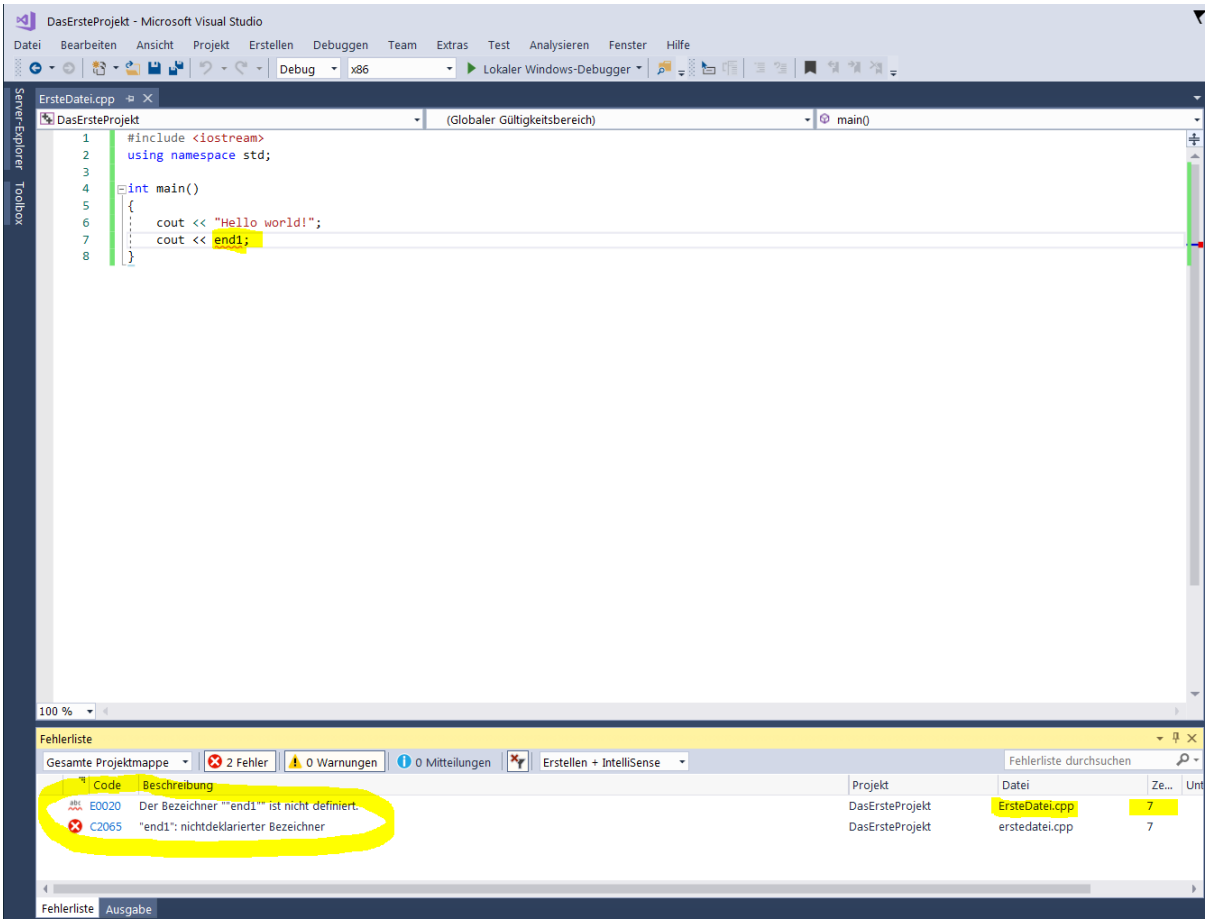
Im Idealfall war unsere Eintipperei erfolgreich und wir werden mit diesem unscheinbaren, aber sehnlichst erwarteten Bild belohnt.



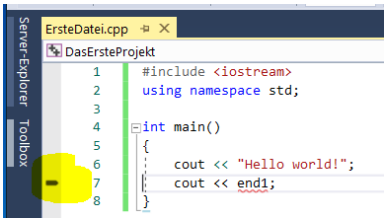
Zwei Dinge fallen ins Auge. Zum einen steht unten im Bereich „Ausgabe“ der magische Satz Satz „1 erfolgreich, 0 fehlerhaft“. Damit müssen wir zumindest keine Fehler mehr im C++-Code suchen, die sind alle gefunden und beseitigt. Damit das auch der Betreuer im Praktikum auf den ersten Blick sieht, hat der vertikal, farbige Balken seine Farbe von gelb nach grün gewechselt.

Besteht unser Projekt aus mehreren Teilen (= Dateien), müssen wir diesen Schritt für jede Datei wiederholen. Und erst, wenn jede Datei erfolgreich übersetzt werden konnte, geht es weiter mit Schritt 2.

Leider sieht der Normalfall anders aus und ein frisch eingetipptes Programm ist alles andere als fehlerfrei. Das sieht dann so aus:



4.1.3 Behebung von Übersetzungsfehlern

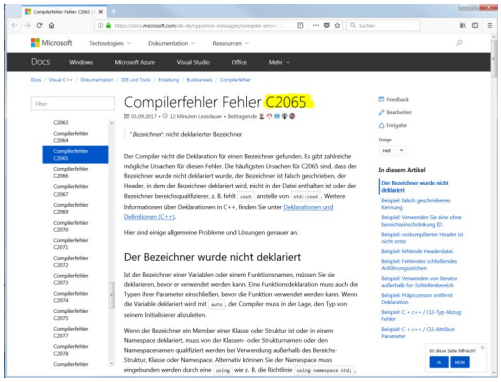


Um diesen Fehler zu provozieren, wurde in Zeile 7 die Anweisung `endl` falsch geschrieben (der Buchstabe l wurde durch die Ziffer 1 ersetzt). Visual Studio ist allerdings ein intelligentes und ausgereiftes Produkt und merkt solche offensichtlichen Fehler sofort. Das sieht man ganz schnell an der roten Wellenlinie unter dem falsch geschriebenen Wort. Das kennen Sie vielleicht aus einer gewöhnlichen Textverarbeitung wie Word oder LibreOffice. Eine rote Wellenlinie bedeutet in Visual Studio immer Handlungsbedarf, denn anders wie bei Prosa in Word oder LibreOffice gibt es bei der Schreibweise von C++-Anweisungen keinen Interpretationsspielraum, weil es nur genau eine richtige Schreibweise gibt.

Wir ignorieren die Wellenlinie aber einfach mal und kompilieren unsere Datei trotzdem mal. Prompt haut uns die IDE unten eine Fehlerliste um die Ohren. Gleich zwei Ideen liefert sie zu unserem `end1`, entweder ist der „Bezeichner" nicht „definiert" oder nicht „deklariert". Und falls wir die Stelle nicht finden, werden auch der Name der Datei und die Zeile mitgeliefert.

Man hat jetzt mehrere Möglichkeiten. Ist man mit der Fehlermeldung vertraut und weiß, was dagegen zu tun ist, kann man mit einem Doppelklick auf den Text der Fehlermeldung direkt zur richtigen Datei springen und bekommt die fragliche Zeile markiert.

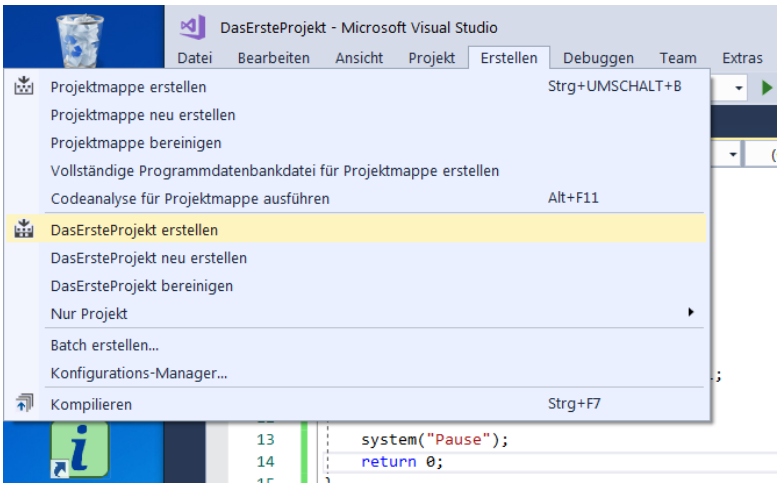
Hat man so gar keine Idee, was der Fehler bedeuten könnte, hilft ein Klick auf die Nummer der Fehlermeldung. Bei bestehender Online-Verbindung landet man direkt auf der entsprechenden Hilfeseite bei Microsoft. Dort kann man sich dann über mögliche, unmögliche oder wahrscheinliche Fehlerursachen informieren.



Versuchen Sie unbedingt, zunächst selbst den Fehler zu beheben, bevor Sie ihren Betreuer rufen. Bevor der Ihnen hilft, fragt er Sie nämlich nach genau den Dingen, die auf der Hilfeseite stehen.

Gibt es mehrere Fehler, ist es immer eine gute Idee, mit dem ersten Fehler zu beginnen. Sehr oft sind die weiteren Fehler nämlich nur Folgefehler, die dann von ganz alleine verschwinden. Und wenn dann alle Fehler beseitigt sind, geht es endlich weiter mit Schritt 2.

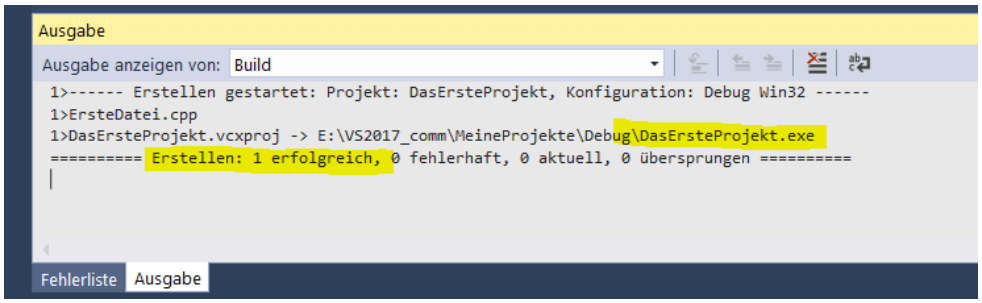
4.1.4 Erstellen (Link, Build)



Schritt 2 verbindet jetzt unsere frisch übersetzten Dateien mit zusätzlich erforderlichen Elementen, die vom Hersteller oder anderen Zulieferern stammen. Dazu gehören in erster Linie die zu Beginn der Datei aufgeführten Bibliotheken, die bereits übersetzt vorliegen, aber noch ins Projekt eingebaut werden müssen. Das holen wir jetzt nach:

Über den Menüpunkt „Erstellen → Projekt erstellen“ werden jetzt alle benötigten Dateien zusammengebaut (engl. to build, andere verwenden auch den Begriff „Linker“)

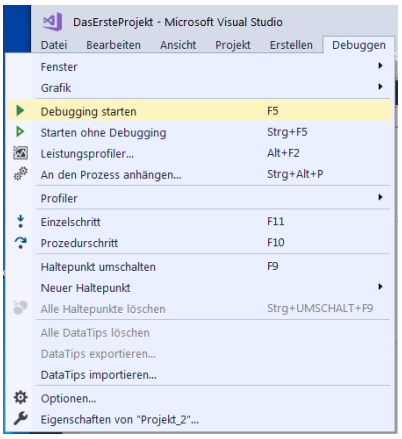
Das sollte bei unserer ersten Datei, die ja jetzt fehlerfrei ist, problemlos funktionieren, da solche mitgelieferten Bibliotheken beliebig oft getestet und verwendet werden und daher hoffentlich ohne Fehler sind. Außer einer Meldung, dass alles erfolgreich war, passiert allerdings nichts:



4.1.5 Ausführen (Run, Debug)

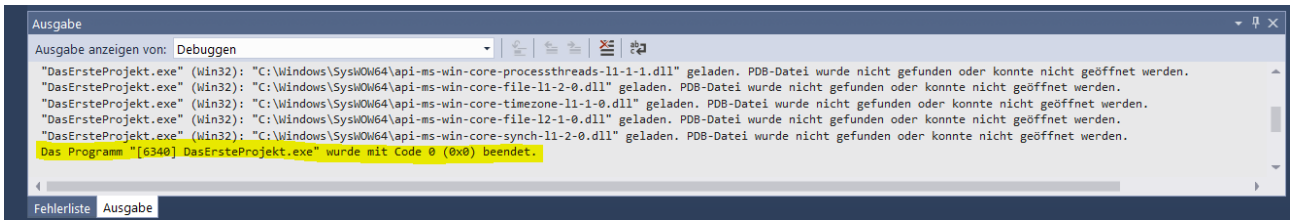
Erst Schritt 3 führt uns zum krönenden Abschluss, einem laufenden Programm. Das erzeugte, lauffähige Programm (zu erkennen an der Endung .EXE für „executable“) wird in den Hauptspeicher des Rechners geladen und Windows übergibt die Programmmausführung an dieses Programm. Das geht am schnellsten über die Funktionstaste F5 oder über das Menü „Debuggen → Debuggen starten“.

Und weil dieser Schritt doch recht häufig vorkommt, hat er mit dem kleinen grünen Dreieck sogar eine eigene Schaltfläche in der Symbolleiste.



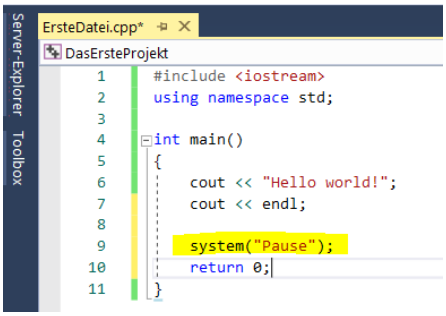
(Debugging ist der englische Begriff für das Suchen und Finden von Fehlern. Bei der Entwicklung von Programmen wird grundsätzlich die IDE in einem Modus betrieben, der das Fehler suchen sehr erleichtert. Das sehen wir uns aber später noch genauer an. Im Moment ist es aber unerheblich, ob wir im Debug-Modus sind oder nicht, wir bemerken zunächst keinen Unterschied).

Und was macht unser Programm jetzt?. Nun ja, genau das, was es tun soll. Es schreibt den Text „Hello world!“ auf den Bildschirm. Nur blöd, dass man davon so überhaupt nicht mitbekommt, weil das alles ziemlich schnell geht und das menschliche Auge ziemlich träge ist. Lediglich die Ausgabe macht da ein wenig Hoffnung:



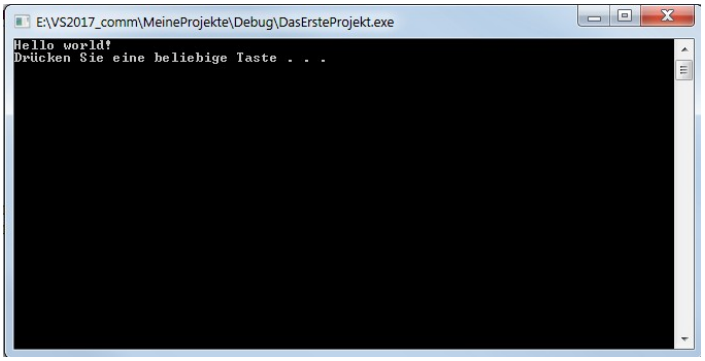
Unser Programm wurde also erfolgreich (mit Code 0x0) beendet, toll.

Natürlich gibt es eine Lösung für dieses Problem, diese muss aber nur bei Visual Studio eingesetzt werden. Machen Sie das auf keinen Fall in einer anderen IDE!



Die Anweisung in Zeile 9 sorgt dafür, dass unser Programm, bevor es beendet wird, noch auf einen Tastendruck wartet[6]. Und damit sehen wir auch etwas auf dem Bildschirm:

6 Diese Anweisung ist ab der Version 2019 nicht mehr erforderlich und kann einfach weggelassen werden.

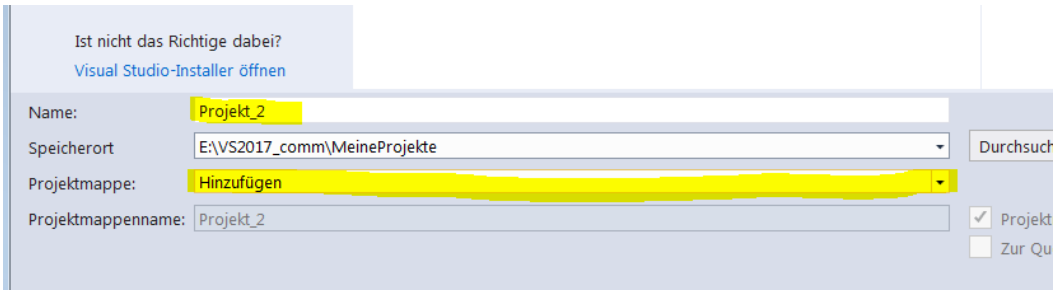


Dieses schwarze Loch nennt man Konsole und sie wird uns über das komplette Semester begleiten. Für Programme mit einer grafischen Oberfläche, wie man sie heute allgemein hat, fehlt uns einfach die Zeit, den Stoff zu vermitteln. Leider.

4.1.6 Weitere praktische Funktionen der IDE

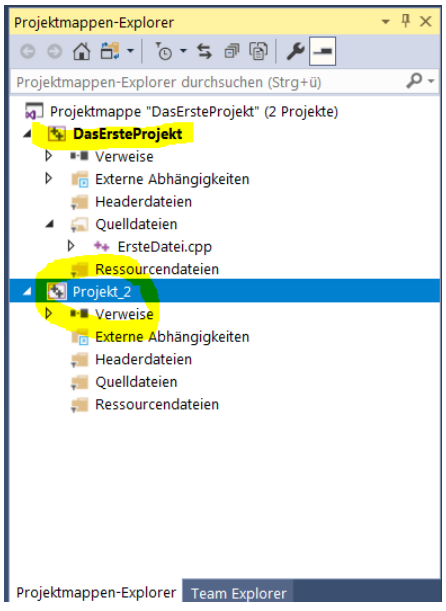
Mehrere Projekte unter einem Dach

Visual Studio ist ein Werkzeug, mit dem auch große und größte Projekte erstellt werden, nicht zuletzt Visual Studio selbst oder Schwergewichte wie Word und Excel. Diese Fähigkeiten nutzen uns auch im Praktikum. Mit der Projektsammelmappe (Solution) kann man mühelos mehrere Projekte gleichzeitig bearbeiten und vor allem Dateien aus einem Projekt schnell und unkompliziert in ein neues Projekt übernehmen. Dazu muss man lediglich beim Anlegen eines neuen Projektes eine zusätzliche Einstellung vornehmen:

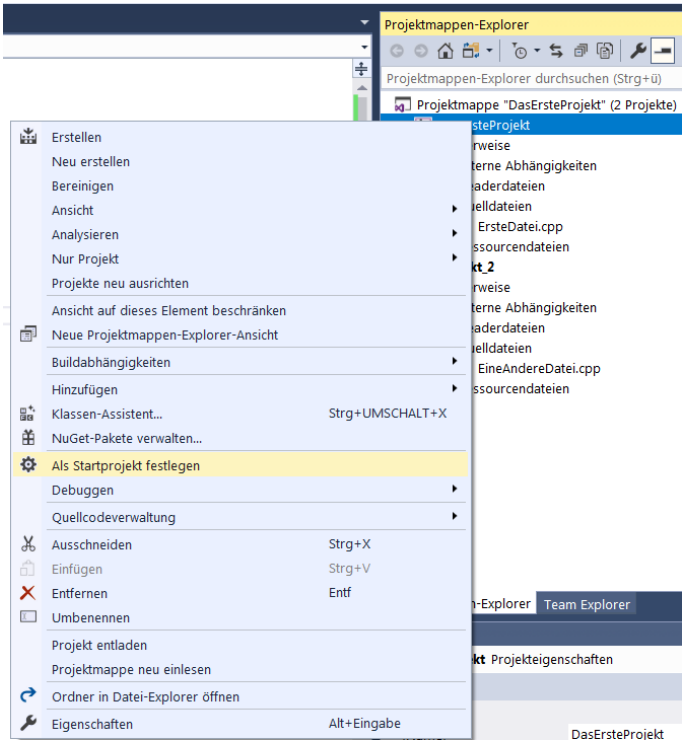


Beim Projektnamen bleibt alles beim Alten, auch beim Speicherort ist alles wie gehabt. Nur bei der „Projektmappe“ stellt man die Auswahl auf „Hinzufügen“. Und schon hat man zwei (oder noch mehr) Projekte in der IDE.

Achtung, wenn mehrere Projekte in der IDE angemeldet sind, muss man festlegen, welches Projekt gerade das ist, an dem gearbeitet wird und auf das sich unsere Übersetzungs- und Erstellenbefehle beziehen.



Dieses Projekt ist das aktive Projekt und ist am Fettdruck zu erkennen, im Bild also „DasErsteProjekt“. Um das neu hinzugekommene „Projekt_2“ zum aktiven Projekt zu machen, muss man es mit der rechten Maustaste anklicken und den Menüpunkt „Als Startprojekt festlegen“ auswählen. Ab sofort ist das neue Projekt das aktive Projekt.

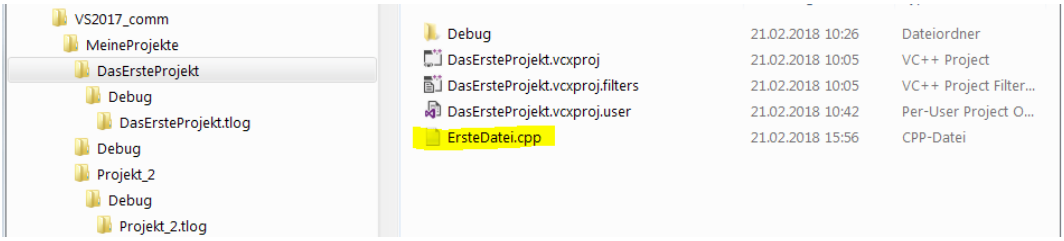


Spätestens an dieser Stelle wird klar, warum man die Namen der Dateien in den Projekten nicht bei den Standardeinstellungen belassen sollte: bei drei Projekten hat man dreimal die Datei quelle.cpp auf dem Bildschirm. Und ändert garantiert in der falschen Datei....

Dateien von einem anderen Projekt übernehmen

In manchen Praktikumsaufgaben soll man eine alte Aufgabe erweitern oder modifizieren. Dazu ist es natürlich ganz praktisch, wenn man die bereits erstellten Dateien einfach weiter benutzen kann. Diese einfache Wiederverwendung ist aber in Visual Studio nicht vorgesehen, das einfache Kopieren der Dateien innerhalb der IDE, wie man es aus dem Windows-Explorer kennt, funktioniert nämlich nicht.

Im folgenden Bild sieht man die Ordnerstruktur, die den oben gezeigten Bildern zu Grunde liegt. Unter dem Ordner „VS2017_Comm“ liegt der Ordner mit dem Namen der Projektsammelmappe (hier „MeineProjekte“) , darunter gibt es für jedes Projekt einen weiteren Ordner.



Hier ist der Ordner „DasErsteProjekt“ geöffnet, darunter befindet sich dann der Ordner „Debug“ (für alle Dateien, die bei der Ausführung benötigt werden), eine Handvoll Projektdateien, die Visual Studio zum Funktionieren benötigt und selbst organisiert und dann ziemlich unauffällig unsere selbst geschriebene Datei „ErsteDatei.cpp“, also das Kernstück unseres Programms.

Analog verhält es sich mit den Dateien im zweiten Projekt, die finden sich komplett unterhalb von „Projekt_2“. Kopiert man jetzt innerhalb der Entwicklungsumgebung eine Datei aus dem ersten Projekt ins zweite, wird diese Kopieraktion auf Windows-Dateiebene **nicht** mitgemacht. Und damit stimmt die Ordnerstruktur auf der Festplatte nicht mehr mit den Informationen in der Projektmappen-Explorer (Solution-Explorer) überein.

Man muss also von Hand die Dateien auf Windows-Dateiebene in das richtige Verzeichnis kopieren, danach über „Hinzufügen → vorhandenes Element“ dem neuen Projekt hinzufügen. Oder man legt im neuen Projekt die benötigten Dateien als neue (leere) Dateien an und kopiert anschließend per Editor Copy&Paste die Inhalte der alten Dateien hinein. Dabei kann man auch gleich die Namen der Dateien an

die neuen Gegebenheiten anpassen. Falls Ihnen das Verfahren irgendwie bekannt vorkommt, haben Sie die Seite 16 gelesen und verstanden.

Besonders wichtig ist diese Einschränkung insbesondere bei Include-Dateien, die unbedingt im richtigen Verzeichnis stehen müssen, weil sie sonst nicht gefunden werden und es zu beliebig vielen Fehlern der Art „nicht aufgelöstes externes Symbol" kommt, die erst beim Erstellen auftreten und nicht so leicht zu finden sind (das in 4.1.3 „Behebung von Übersetzungsfehlern" beschriebene Verfahren läuft nämlich bei diesen Fehlern ins Leere und ist nicht zu verwenden.

4.1.7 Fazit

Sie haben gesehen, wie man ein erstes Projekt anlegt, ein Projekt übersetzt und ausführt, Fehler findet und beseitigt. Visual Studio bietet insbesondere im Editor noch viele Funktionen, die das Arbeiten schneller, sicherer und komfortabler machen. Diese hier alle zu beschreiben würde den Rahmen sprengen, aber es lohnt sich trotzdem, auf Entdeckungsreise zu gehen und es auszuprobieren.

4.1.8 Debuggen

Dieses Kapitel wäre unnötig, wenn jeder nur fehlerfreie, gut strukturierte und leicht nachvollziehbare Programme schreiben würde. Willkommen in der Wirklichkeit.

Wir haben ja bereits in Abschnitt 4.1.5 davon gelesen, dass die Ausführung unseres Programms unter Kontrolle des Debuggers läuft, das aber für uns keinen Unterschied zum „normalen" Ablauf macht. Das stimmt genau solange, wie das Programm genau das tut, was es soll. Interessant und spannend wird es, wenn ein Programm Ergebnisse liefert, die nichts mit den Testwerten (ja, die leidige Vorbereitung muss doch einen Sinn haben) zu tun haben oder andere Unregelmäßigkeiten auftreten. Dann schlägt die Stunde des Debuggers.

Streng genommen ist der Debugger nichts weiter wie eine geschützte Umgebung in der unser Programm unter Aufsicht abläuft. Also so ähnlich wie in der Fahrschule: fährt der Fahrschüler „richtig", muss der Fahrlehrer nicht eingreifen. Im Ernstfall tritt er aber mal heftig auf die Bremse, um Schäden zu vermeiden.

Ist der Debugger einmal aus seiner Deckung hervorgekommen, kann er jede Programmzeile einzeln ausführen,. Die Inhalte (Werte) von Variablen anzeigen, diese sogar ändern und natürlich auch Amok-laufende Programme beenden.

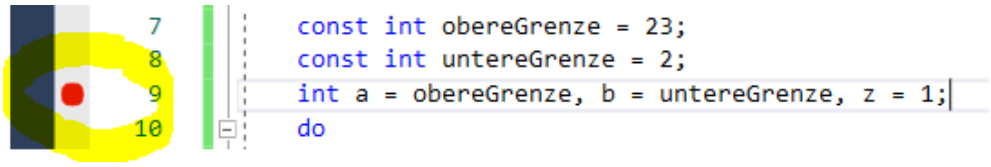
Damit das passiert, muss man (mindestens) einen sogenannten Haltepunkt (Breakpoint) in das Programm einfügen, ab dort wird dann der Debugger aktiv und übernimmt die Kontrolle. Für unsere Fehlersuche verwenden wir eine Aufgabe aus einer Klausur, bei der man den Vorteil des Debuggers gegenüber einer manuellen Fehlersuche ganz schnell sieht. Das ist das Programm:

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

int main()

{
	const int obereGrenze = 23;
	const int untereGrenze = 2;
	int a = obereGrenze, b = untereGrenze, z = 1;
	do
	{
```

```
            cout << z << ". Zeile: ";
            for (b = untereGrenze; b <= obereGrenze; b = b + 5)
            {
                  cout << setw(5) << (a + b);
            }
            cout << endl;
            a = a - 5;
            z++;
      } while (untereGrenze < a);
}
```

Die Aufgabe in der Klausur ist es, die Ausgabe dieses Programms zu ermitteln und aufzuschreiben. Dazu muss man konsequent Zeile für Zeile durchgehen und dabei alle Änderungen in den Variablen nachvollziehen. Machbar, aber mühsam. Deswegen nutzen wir dafür den Debugger, der kann das perfekt und vergisst auch nichts.

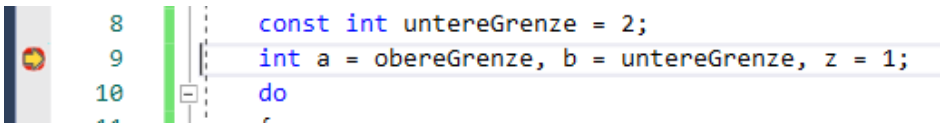
Um das gesamte Programm unter der Kontrolle des Debuggers laufen zu lassen, muss der Breakpoint gleich zu Beginn der ausführbaren Anweisungen stehen. Ich habe mich für die Zeile 9 entschieden:



Ein einfacher Klick links in den (schwer erkennbaren) grauen Rand setzt den Haltepunkt und aktiviert ihn (durch einen erneuten Klick wird man den Haltepunkt auch genauso einfach wieder los).

Startet man die Programmausführung jetzt mittels 

beginnt die Programmausführung auch ganz normal, aber sie stoppt sofort beim Erreichen des Haltepunktes. Der kleine gelbe Pfeil gibt an, wo die Programmausführung aktuell steht, VOR der Zeile 9.



Gleichzeitig erscheint im unteren Bereich der IDE ein neues Fenster mit dem Titel „Auto“:

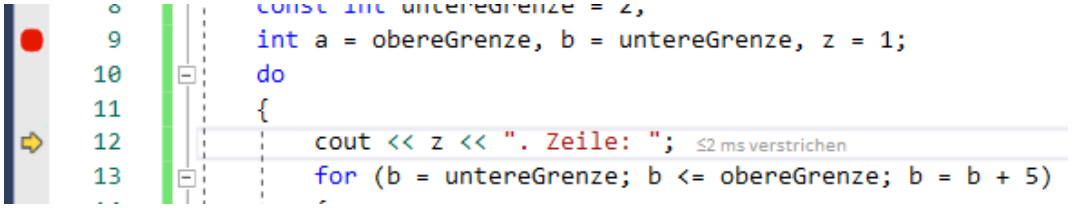
Hier werden jetzt alle Variablen mit Name, Inhalt und Typ aufgelistet, die Visual Studio an dieser Stelle für relevant hält (daher der Titel „Auto“). Man erkennt die beiden Variablen `obereGrenze` und `untereGrenze`, die „sinnvolle“ Werte enthalten, weil diese in Zeile 8 gesetzt wurden. Bei den Inhalten von a, b und z sieht das anders aus, der Wert `-858993460` kommt in unserem Programm nämlich nirgends vor[7]. Das ist auch kein Wunder, denn diese Variablen wurden bisher weder definiert, noch deklariert und erst recht nicht initialisiert. Der Debugger weiß aber, an welcher Speicherstelle die

7 Diese Zahl kann bei Ihnen andere, aber sehr ähnliche Werte haben. Ganz so zufällig scheint sie auch nicht, in hexadezimaler Schreibweise ist das nämlich `FFFF FFFF CCCC CCCC`, zu schön für einen zufälligen Wert.

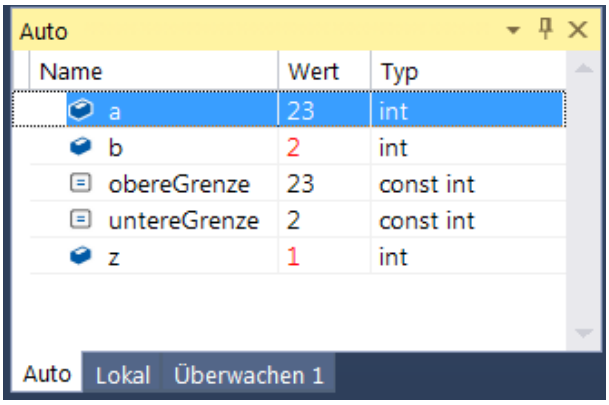
Variablen angelegt sind und gibt diesen Speicherinhalt schon einmal aus. Der Inhalt des Speichers ist aber eine mehr oder weniger zufällige Folge von Bits (digitales Rauschen), daher kommt dieser zufällige Wert zustande.

Falls Ihr Programm einmal derartige Zahlenwerte ausgibt, ist das ein deutliches Zeichen dafür, dass hier eine Variable verwendet wird, die nicht mit einem Wert initialisiert wurde oder es wurde auf einen Speicherbereich zugegriffen, der nicht zum Datenbereich des aktuellen Programms gehört. Von einem solchen Programm kann man keine korrekten Ergebnisse erwarten, also ist Fehlersuche angesagt

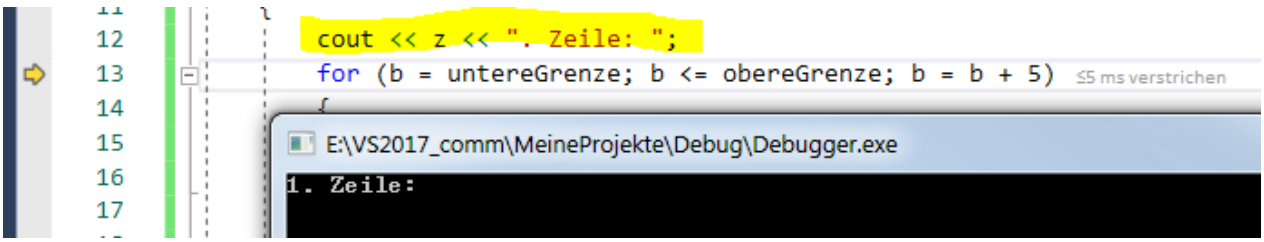
Aktuell steht unser Programm ja, es bedarf also eines Anstoßes zum weiterlaufen. Die nötigen Schaltflächen sind zufällig oben in der Symbolleiste ganz von alleine aufgetaucht. Mit den drei blauen Pfeilen kann man das Programm Schritt für Schritt weiter abarbeiten, für uns reicht der mittlere  aus, der sich Zeile für Zeile vorwärts bewegt. Statt auf die Schaltfläche zu klicken, kann man auch mit F10 arbeiten. Nach der ersten Betätigung sieht unsere IDE so aus:



Der gelbe Pfeil ist auf die nächste ausführbare Anweisung gesprungen, die Anweisungen in Zeile 9 wurden also ausgeführt. Das sieht man auch an den geänderten Werten unter „Auto":

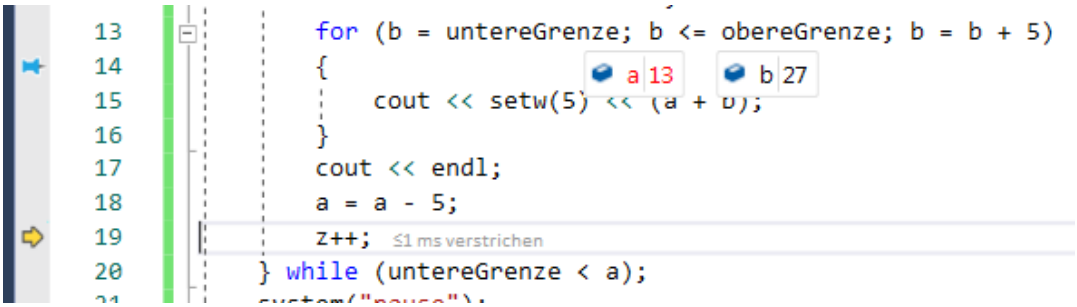


Die Werte von a, b und z haben sich genau so geändert, wie es die gerade abgearbeiteten Anweisungen in Zeile 9 vorgegeben haben. Damit das auch bei vielen Variablen leichter gefunden wird, werden geänderte Werte in rot angezeigt. Der nächste Schritt führt die Zeile 12 mit der Ausgabeanweisung aus:

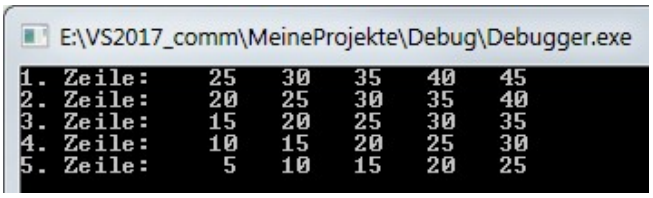


Der gelbe Pfeil wandert weiter zu Zeile 13.

Auf diese Weise kann man jetzt das gesamte Programm Schritt für Schritt durchlaufen und dabei die Variablen und ihre Werte beobachten. Fehler sollten dabei relativ schnell auffallen, da dies ja eine Abweichung der Testwerte von den tatsächlichen Werten bedeutet.



Hier sieht man noch eine zweite Methode, die Werte von Variablen zu verfolgen. Fährt man mit der Maus über eine Variable, wird deren Wert in einem kleinen PopUp angezeigt. Dieses PopUp lässt sich „pinnen" und bleibt dann ständig sichtbar. Damit behält man auch in unübersichtlichen Variablenlisten den Überblick. Das ist übrigens die Ausgabe des Programms:



Der Debugger erscheint vielen Programmieranfängern als umständlich und kompliziert. Er bietet aber viele Möglichkeiten, Ablauffehler einzugrenzen und zu finden. Alternativ geht das nur mit jeder Menge zusätzlicher Ausgaben, die aber nichts anderes machen, wie der Debugger, aber bedeutend aufwändiger sind. Es lohnt sich also durchaus, etwas Zeit für das Kennenlernen des Debuggers zu investieren.

4.2 Weitere IDEs

Sehr beliebt und ähnlich leistungsfähig ist Eclipse, zu bekommen unter http://www.eclipse.org/. Ursprünglich für Java entwickelt, werden auch hier keine Compiler mitgeliefert, sondern Eclipse greift auf die (getrennt zu installierenden oder bereits vorhandenen) Compiler zu.

Stark im Kommen ist auch CodeBlocks von http://www.codeblocks.org, diese IDE wurde speziell für C und C++ entwickelt, benötigt aber auch zusätzliche Compiler. Diese IDEs werden in einer späteren Auflage Eingang in diese Skript finden.

Bei Informatikern sehr beliebt ist Qt, das es unter https://www.qt.io/download für Windows und Linux gibt.

4.3 Die Programmiersprache

Ein Computer (genauer gesagt der Prozessor) ist eigentlich ein ziemlich dummes Bauteil, das lediglich Einsen und Nullen miteinander verknüpfen kann, das aber in einer wahnsinnig hohen Geschwindigkeit. Uns Menschen fehlt aber die Möglichkeit, diese schnelle Abfolge von Ziffern zu verstehen. Der Mensch hat also Methoden und Regeln gefunden, eine Sprache zu definieren, die sowohl der Mensch als auch der Computer benutzen können. Zunächst waren das Befehle in einer sehr maschinennahen Sprache, dem Assembler. Der war noch sehr kryptisch, aber immerhin lesbar.

Darauf bauen die modernen Programmiersprachen auf. Sie übersetzen aus einer Hochsprache, die der Mensch sehr gut versteht, in eine Sprache, die die Maschine versteht. Je nach Programmiersprache

passiert das in dem Moment, in dem der Programmierbefehl ausgeführt werden soll (Interpreter) oder das gesamte Programm wird übersetzt und in maschinenlesbarer Form auf dem Computer ausgeführt (Compiler). Bei C++ findet das zweite Verfahren Anwendung.

Zu jeder Programmiersprache gehören nicht nur Compiler oder Interpreter, sondern immer auch eine gewisse Anzahl Bibliotheken, die häufig benötigte Funktions- und Arbeitsabläufe als fertige Module bereitstellen. Dazu gehören z.B. die Befehle für die Ein- und Ausgabe oder umfangreiche mathematische Funktionen. Diese Funktionen werden durch den Linker nach dem Übersetzen zu den selbst geschriebenen Programmen dazu gebunden und machen daraus erst ein einsatzfähiges Programm. (Das ist sehr vereinfacht dargestellt, es sind weitere Schritte und Helfer notwendig, um ein lauffähiges Programm zu erzeugen.)

Programmiersprachen gibt es wie Sand am Meer, sie werden sehr oft nach speziellen Fähigkeiten verwendet, die eine kann prima mit der Hardware eines Rechners umgehen, eine andere ist ideal, um Zahlen ohne Verluste zu bearbeiten (wie in der Buchhaltung), wieder andere können sehr gut mit riesigen Datenmengen umgehen usw. C++ kann das alles relativ gut und ist damit ziemlich vielseitig einsetzbar. Und daher schreiben wir jetzt unser ersten Programm in C++

4.4 Warum C++?

Weltweit führen C++, Java und Python die Liste der am häufigsten eingesetzten Programmiersprachen an. Als Informatiker kommt man also nicht umhin, sich mit diesen intensiv zu beschäftigen. Es reicht aber, **eine** Programmiersprache so richtig von Grund auf zu erlernen, dann muss man sich bei jeder neuen Sprache nur noch deren Spezialitäten und Besonderheiten ansehen und verinnerlichen, dann kann man auch diese Sprache einsetzen.

Für Nicht-Informatiker ist es wichtig, die Konzepte einer Programmiersprache zu erlerne, zu verstehen und später anwenden zu können. Auch wenn dieses Anwenden bei vielen später nur aus dem Was-wäre-wenn von Excel besteht, die Makros von Excel sind auch nur eine spezielle Programmiersprache.

C++ hat den Vorteil, dass es sowohl die klassische imperative (oder funktionale) Programmierung und die aktuelle objektorientierte Programmierung ohne Brüche abdeckt, man kann also zunächst das Programmieren im funktionalen Stil erlernen und dann zur Objektorientierung weiter gehen. Dazu kommt, dass es sich bei C++ um eine **typisierte** Sprache handelt, ein unschätzbarer Vorteil bei Kategorisierung von Daten.

Java versteckt den prozeduralen Unterbau vor dem Anwender und setzt fast komplett auf die Objektorientierung.Das macht den Einstieg für Anfänger deutlich schwerer, Informatiker fühlen sich dadurch heraus gefordert.

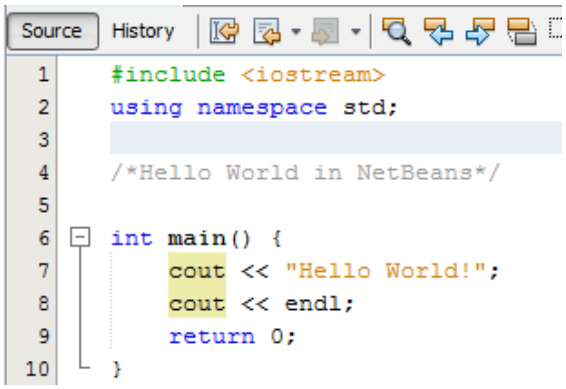
Python kann auch prozedural und objektorientiert, verzichtet aber auf die Typisierung bei den Daten. Das ist verlockend, weil man sich keine Gedanken über die Datentypen mehr machen muss, führt dann aber später zu Problemen.

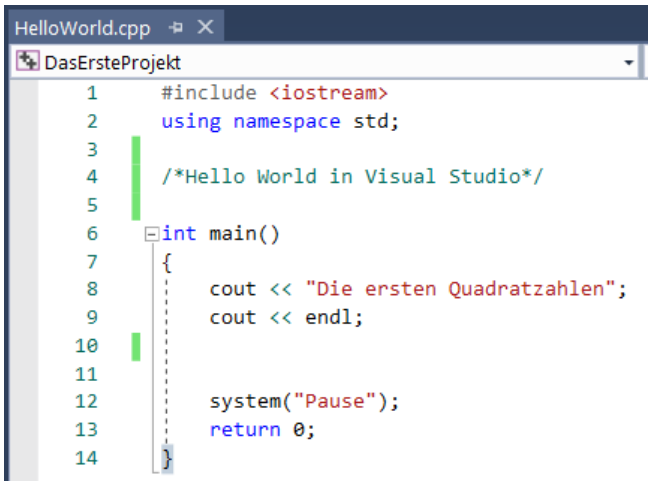
Daher haben wir uns an der h_da für den Einstieg mit C++ entschieden, im ersten Semester beginnen wir mit der funktionalen Programmierung und erweitern dies im zweiten Semester auf die Objektorientierung (die Informatiker machen das in einem Semester, haben aber auch einen höheren Stundenumfang dafür zur Verfügung)

4.5 Der Klassiker: Hello World

Dieses Programm muss immer dann herhalten, wenn es um die Einführung oder Vorstellung einer (neuen) Programmiersprache geht. Auch ich möchte auf solche, Jahrzehnte alte Gepflogenheiten nicht

verzichten und habe dieses Programm in den vorherigen Abschnitten einfach mal genutzt. Zur Erinnerung hier noch einmal dargestellt, links in NetBeans, rechts in Visual Studio:





Der Quelltext sieht nahezu identisch aus, jede IDE benutzt andere Farben zur Hervorhebung der Syntax und in Visual Studio ist die Zeile 12 zusätzlich enthalten (siehe dazu Abschnitt 4.1.5).Abgesehen von diesen Kleinigkeiten, sind beide Programme identisch. Es wäre auch nicht wirklich schön, wenn jede IDE einen eigenen „Dialekt" von C++ spricht, dann wäre es mit der universellen Verbreitung ganz schnell vorbei.

Und weil das so ist, kann man die Quelltexte aus den Entwicklungsumgebungen auch direkt unter den IDEs austauschen, ohne dass sich etwas an der Lauffähigkeit des Programms ändert.[8] Klarer Vorteil: man kann mit einer beliebigen IDE das Programmieren zu Hause üben und dann im Praktikum ganz einfach auf eine andere IDE wechseln, ohne schädliche Nebenwirkungen zu befürchten.

Im Rahmen dieses Skriptes kann ich mich so darauf beschränken, die Programme unabhängig von einer IDE darzustellen und mich im Folgenden rein auf die Programmiersprache konzentrieren.

8 Leider hört diese Kompatibilität dann doch irgendwann auf, spätestens, wenn es um die Gestaltung grafischer Oberflächen geht, aber die kommen bei uns ja sowieso nicht vor

5 Was steht drin im ersten Programm?

Egal mit welcher Entwicklungsumgebung Sie dieses erste Programm erstellt haben, die Inhalte sind sich doch sehr ähnlich. Das sollte auch so sein, denn ein Programm darf sich ja nicht anders verhalten oder anders aussehen, nur weil eine andere Entwicklungsumgebung verwendet wird.

5.1 Die Bestandteile eines Programms

5.1.1 Kommentare

Das Paar aus `/*` und `*/` (in Zeile 4) nennt man Kommentar und der ist für das eigentliche Programm ohne jede Bedeutung. Mit anderen Worten: es interessiert den Compiler überhaupt nicht, ob ein Kommentar existiert oder nicht. Etwas anderes ist es aber für einen menschlichen Beobachter- der kann mit den Kommentaren in einem Programm etwas anfangen, wenn sie denn sinnvoll angebracht wurden. Ein Kommentar ist dazu gedacht, die wesentlichen Schritte, die nicht offensichtlich zu erkennen sind, so zu beschreiben, dass man sie nachvollziehen kann. Der Kommentar `/*Berechnung der Dreiecksfläche nach dem Heron-Verfahren*/` an der mathematischen Formel `a=sqrt(s(s-a)(s-b)(s-c)` macht also Sinn während der Kommentar `/*Eingabe des Namens*/` an einer Programmzeile `Bitte Name eingeben:` keine zusätzliche Information bringt.

Ein anderer, häufiger Anwendungsfall für einen Kommentar ist es, aktuell nicht benötigte Programmteile durch eine Auskommentierung für begrenzte Zeit von der Programmausführung auszunehmen. Damit kann man dann andere Teile besser überprüfen und testen, oder fehlerhafte Teile überspringen. Der Phantasie sind dabei keine Grenzen gesetzt. Der Editor unterstützt dieses Vorhaben mit einer eigenen Funktion, die markierten Text in Sekundenbruchteilen in einen Kommentar verwandelt und auch genauso schnell wieder aktiviert.

Es gibt als weiteren Kommentar noch die Ausführung `//` mit der der Text ab dieser Stelle bis zum Zeilenende zum Kommentar wird. Das könnte in einem Programm etwa so aussehen:

```
a=sqrt(s*(s-a)*(s-b)*(s-c)); //Heron-Formel
```

Ein Kommentar ist immer dann angebracht, wenn ein Programmstück von einem Außenstehenden nicht auf einen Blick erfasst und nachvollzogen werden kann oder wenn zusätzliches Wissen eingeflossen ist, das nicht jeder Leser parat hat. Und dann stehen ja aus unserer Entwurfsphase (Seite 14) noch Kommentare drin, die bei unserem Top-Down-Entwurf entstanden sind. Die bleiben natürlich drin!

5.1.2 Leerzeilen

Zeile 3 und Zeile 5 sind einfach nur leer und dienen zur Erhöhung der Übersicht. Eine Leerzeile darf überall da stehen, wo mindestens ein Leerzeichen verlangt wird. Sie empfiehlt sich zur Trennung der verschiedenen Abschnitte, sollte aber nur wohldosiert eingesetzt werden. Auf keinen Fall ist es empfehlenswert, nach jeder Zeile eine Leerzeile einzufügen, damit erscheint das Programm zwar länger, aber auch sehr viel unübersichtlicher, weil nur noch die Hälfte davon auf den Bildschirm passt. Ziel sollte es sein, dass immer ein kompletter Funktionsbereich auf den Bildschirm passt.

5.1.3 Bibliotheken

In Zeile 1 passiert etwas Wichtiges. Das Zeichen # sagt dem Compiler, dass die nächste Anweisung kein Bestandteil der Programmiersprache an sich ist, sondern dem Compiler selbst gilt. Und zwar soll er

mittels `#include` an dieser Stelle etwas einfügen, was dem Compiler unter dem Namen `iostream` bekannt ist. Dieses `iostream` ist eine sogenannte Bibliothek, die vorgefertigte Programmteile enthält, die jeder Programmierer mehr oder weniger häufig auf jeden Fall brauchen wird. Daher werden solche Standard-Bibliotheken auch professionell erstellt bei nahezu allen Compilern mitgeliefert. Der „normale" Programmierer kann diese dann direkt und ohne lange Überlegung für seine Zwecke einsetzen

Zu jedem C++-Paket gehört ein sehr umfangreicher Satz an Bibliotheken, darunter die klassischen mathematischen Funktionen, die man auch von jedem besseren Taschenrechner kennt. Mit die wichtigsten Bibliotheken sind aber die, die sich um die Ein- und Ausgabe kümmern. Denn zum einen gehört die Ein-/Ausgabe nicht unmittelbar zur Programmiersprache, ist zum anderen aber notwendig, um überhaupt mit einem Programm etwas anfangen zu können. Und genau das enthält unsere Bibliothek `iostream`. Darin findet man alles, was man zum Ein- und Ausgeben braucht. Die spitzen Klammern `<...>` um den Namen zeigen dem Compiler, an welchem Ort (auf der Festplatte) er diese Bibliothek findet. Wird der Name einer Bibliothek in diesen spitzen Klammern eingeschlossen, findet sie sich in einem eigenen Verzeichnis innerhalb des Compilers. Steht der Name in doppelten Anführungszeichen `"..."` liegt die Bibliothek im aktuellen Projektverzeichnis (und ist dann meistens selbst geschrieben).

5.1.4 Hauptfunktion

In Zeile 6 geht es endlich wirklich richtig los! Es beginnt mit dem Schlüsselwort `int`, gefolgt von dem Bezeichner `main` und einem runden Klammerpaar (). Beginnen wir mit dem mittleren Teil, dem Bezeichner `main`, genau genommen mit dem Begriff Bezeichner.

Ein Bezeichner ist ein vom Programmierer nach gewissen Regeln selbst wählbarer Begriff, der vielen Dingen in einem Programm einen passenden Namen gibt. Diese Dinge sind u.a. Funktionen, Klassen (Mit Klassen werden wir uns hier nur am Rande beschäftigen), Variablen und Konstanten. So ein Bezeichner sollte so ausgewählt werden, dass sein Namen auch tatsächlich etwas aussagt. Beispiele: `istPrimzahl`, `tabelleAusdrucken`, `eingabe_lesen`, `x_Koordinate` usw. Zum Bilden von Bezeichnern gelten folgende Regeln:

- Ein Bezeichner muss mit einem Buchstaben a - z, A- Z oder dem Unterstrich _ beginnen.
- Die weiteren Zeichen dürfen auch Zahlen sein
- Groß- und Kleinschreibung wird unterschieden, Umlaute und andere Sonderzeichen sind verboten

Visual Studio ist an dieser Stelle (leider) etwas anders veranlagt wie alle anderen Entwicklungsumgebungen. Hier sind diverse Sonderzeichen, insbesondere die deutschen Umlaute, in Bezeichnern erlaubt. Man sollte sich aber daran gewöhnen, diese Möglichkeit zu meiden, ebenso bei den Namen der Dateien im Projekt. An irgendeiner Stelle funktionieren die Sonderzeichen dann nämlich plötzlich nicht mehr (bei den Dateinamen spätestens bei Verwendung des Debuggers)

Es hat sich durchgesetzt, **nicht** mit einem Unterstrich _ zu beginnen (solche Bezeichner erzeugt der Compiler selbst). Um längere Bezeichner (aus zusammengesetzten Worten) besser lesen zu können, werden die Anfangsbuchstaben der einzelnen Worte groß geschrieben (CamelCase) oder ein Unterstrich eingefügt. Außerdem ist es üblich, Variablen mit Kleinbuchstaben und Klassen mit Großbuchstaben beginnen zu lassen. Konstanten werden komplett in Großbuchstaben geschrieben.

Den Bezeichner `main` wählt man als Programmierer nicht zufällig, denn er spielt in jedem C- und C++-Programm eine besondere Rolle. An dieser Stelle (genauer: mit dieser Funktion) beginnt grundsätzlich und immer die Programmausführung. Es muss in jedem Projekt genau eine Funktion `main` geben, aber auf keinen Fall mehrere davon[9].

Vor dem `main` steht noch ein unauffälliges `int`, eine Angabe darüber, welche Art von Daten diese Funktion als Ergebnis zurück gibt. Die `main` -Funktion liefert üblicherweise einen Zahlenwert an seinen Aufrufer zurück. Ist der 0 (Null) war alles in Ordnung, bei jedem anderen Wert hat irgendetwas nicht so richtig geklappt. Der Aufrufer der Funktion `main` ist nicht der Compiler, sondern das Betriebssystem, also etwa Windows, Linux, MacOS oder wo auch immer wir unser Programm dann laufen lassen.

Die beiden runden Klammern ( ) dahinter umrahmen die Parameterliste, das sind die Argumente, die der Funktion zur Verarbeitung übergeben werden. In diesem konkreten Fall gibt es keine Argumente, die Klammern müssen aber trotzdem stehen!

5.1.5 Block

Zum Abschluss der `main`-Zeile findet sich noch eine geschweifte Klammer {, die zugehörige } findet sich ganz am Ende unseres Programms. Dieses Paar bildet einen Block, das grundlegende Element der Programmierung. So ein Block spielt immer dann eine wichtige Rolle, wenn nach den Regeln der Programmiersprache (der Syntax) nur genau eine Anweisung erlaubt ist. Wie wir später (auf Seite 61) noch sehen werden, ist das eher der Normalfall als die Ausnahme, daher werden uns die geschweiften Klammern auch sehr häufig begegnen. Die Entwicklungsumgebung unterstützt hier besonders gut: wenn man eine öffnende Klammer eingibt, wird die korrespondierende schließende Klammer automatisch eingefügt. Man muss nur aufpassen, diese Klammer nicht aus Versehen wieder zu löschen.

5.1.6 Anweisungen

`cout << "Hallo Welt!";` ist endlich die Zeile, in der auch etwas passiert. `cout` ist eine Anweisung, die etwas ausgibt. Was sie ausgeben soll, steht hinter dem <<, hier ist der Text *Hello World!*. Die Anführungszeichen begrenzen den Text, werden aber selbst nicht ausgegeben. ({Wie man das Anführungszeichen ausgibt erfahren wir auf Seite 45 im Kapitel 6.1.3)

Danach kommt noch eine sehr ähnliche Zeile, das `endl` steht aber nicht in Anführungszeichen, weil es sich nicht um einen einfachen Text handelt, sondern um eine Steueranweisung. Sie erzeugt am Ende der Zeile einen Wechsel in die nächste Ausgabezeile, also einen Zeilenvorschub.

Der Ausgabeoperator << zeigt an, was in die Ausgabefunktion hinein soll, die auszugebenden Daten fließen wie ein Strom zur Ausgabe. Und dieser Strom kann durch weitere << beliebig verlängert werden. (Strom heißt auf englisch *stream*, damit ist auch klar, warum die Bibliothek für die Ein- und Ausgabe `iostream` heißt.)

Streng genommen müsste dort statt `cout` eigentlich `std::cout` stehen, aber Informatiker sind faule Menschen (deswegen lassen sie ja so gerne Maschinen für sich arbeiten) und haben dafür die Zeile 2 im Programm untergebracht. So wird sicher gestellt, dass alle fehlenden `std::` bei Bedarf einfach automatisch eingefügt werden. Mehr dazu steht in Abschnitt 10.2 auf Seite 120.

Die letzte Anweisung sorgt dafür, dass unser Programm das Versprechen aus der `main`-Zeile mit dem numerischen Rückgabewert auch einhält. Sie liefert die 0 ab, damit das Betriebssystem weiß, dass alles in Ordnung war und das Programm ohne Fehler beendet wurde.

9 Das ist ein häufiger Fehler, wenn man eine bestehende Lösung in eine neue Datei kopiert, die alte aber im Projekt belässt

5.2 Style-Guide für die Programmerstellung

In Firmen und Arbeitsgruppen ist es üblich (und unumgänglich), dass gewisse Regeln bei der Gestaltung von Programmen eingehalten werden.Diese Regeln werden unter dem Begriff Style-Guide zusammengefasst. Auch im Praktikum kann es sinnvoll sein, solche Regeln zu beachten, weil es den Umgang mit der Programmiersprache vereinfacht.

Namenskonventionen

Alle Namen und Bezeichner, die in einem Programm verwendet werden, sind so zu benennen, dass aus diesem Namen jederzeit Rückschlüsse auf deren Bedeutung und Inhalt möglich sind. Namen beginnen grundsätzlich mit einem Kleinbuchstaben, bei zusammen gesetzten Worten sind die einzelnen Wortteile mittels Unterstrich oder durch einen Großbuchstaben (CamelCase) zu kennzeichnen.

Namen von Funktionen und Prozeduren sollen die Abläufe in der Funktion/Prozedur mit einem Begriff beschreiben. Daher kommen als Funktionsnamen regelmäßig Verben zum Einsatz. Bezeichner für logische Werte (Bool) bekommen Namen, denen man diese Charakteristik sofort ansieht. Oft kann man dies durch das Voranstellen von `ist`... (deutsch) bzw. `is`...(englisch) erreichen. Der Name sollte zusätzlich auch mit einem vorangestellten „nicht“ das Gegenteil bedeuten.

Handelt es sich um bekannte Größen aus der Mathematik, Physik oder Chemie, so sind die dort üblichen Namen zu verwenden.

Beispiele:

Gut gewählte Namen	Nicht geeignete Namen
anfangsDatum, ende_datum	datum1, datum_2
ist_primzahl, ist_neu, gefunden	primzahl_test
formular_ausgeben(..)	ausdruck(...)

Einrückungen

Einrückungen dienen der Übersichtlichkeit und werden daher von allen IDE unterstützt und nahezu automatisch richtig gemacht. Manuelle Änderungen sind daher zu unterlassen bzw. bei größeren Änderungen im Quelltext ist dieser anschließend komplett neu zu formatieren.

Geschweifte Klammerpaare sind so anzuordnen, dass öffnende und schließende Klammer eines Paares immer in der gleichen Spalte (also untereinander) stehen. Alternativ kann die öffnende Klammer am Ende einer Zeile stehen, wenn die schließende Klammer bündig unter dem entsprechenden Schlüsselwort steht:

```
for (b = unten; b <= oben; b++ )
     {
          cout << (a + b);
     }
```

```
for (b = unten; b <= oben; b++ ) {
               cout << (a + b);
}
```

Es darf in einem Projekt nur eine der beiden Varianten genutzt werden.

Formatierungen

Grundsätzlich steht jede Anweisung in einer eigenen Programmzeile. Längere Ausgabeanweisungen dürfen sich über mehrere Zeilen erstrecken.

Leerzeilen sind nur zwischen den einzelnen Funktionen eines Programms sowie nach den Headern erlaubt. Zwischen den Anweisungen einer Funktion oder eines Blockes sind sie zu vermeiden.

Funktionen/Prozeduren sollen eine Bildschirmseite nicht überschreiten und ohne Blättern komplett zu erfassen sein.

Globale und lokale Variablen
Auf globale Variablen ist zu verzichten. Alle Variablen sind lokal anzulegen.

Testwerte
Zum Testen eines Programms sind solche Testwerte vorab zu ermitteln, die sicherstellen, dass jede Verzweigung des Programms mindestens einmal durchlaufen wird. Die zu einem Testwert erwarteten Reaktionen (Ausgaben, Meldungen, Ergebnisse) sind vorab manuell zu bestimmen.

Vorgaben durch den Dozenten
Einige Dozenten machen zusätzliche Auflagen für ihre Aufgaben. Diese gehen den hier gemachten Regelungen vor.

5.3 Übungen

Welche der folgenden Bezeichner sind gültig und welche sind nicht erlaubt?

`alpha_dog`	`Betakarotin`	`Gamma-Cyclin`	`Delta Force`
`1_2_3`	`One2Three`	`One&Only`	`one4All`

Erweitern Sie das erste Programm so, dass anstelle von Text auch Zahlen ausgegeben werden. Was erscheint in der Ausgabe, wenn Sie `"2 + 5"` mit und ohne Anführungszeichen in den Quelltext schreiben? Probieren Sie das auch mit anderen Rechenzeichen (Operatoren) aus. Macht der Rechner einen Unterschied zwischen `2000.000` und `2000,000`, jeweils mit oder ohne Anführungszeichen?

Schreiben Sie eine große Zahl von Ausgaben mittels << hintereinander. Bei welcher Anzahl von Argumenten liegt die Grenze?

Stellen Sie die Ausgabe so ein, dass dieses Ergebnis heraus kommt:

```
Die Differenz von 8 und 2 ist 6.
```

Dabei ist das Ergebnis 6 in der Ausgabe zu berechnen!

5.4 Zusammenfassung

Wir haben erfolgreich unsere Entwicklungsumgebung gestartet und ein erstes Programm eingegeben. Wir können elementaren Text, Zahlen und Berechnungen ausgeben, auch gemischt in einer Zeile.

Im nächsten Kapitel werden wir uns weiter mit der Ausgabe beschäftigen und lernen, wie man die Ausgaben so gestaltet, dass ansprechende Ergebnisse erzielt werden. Damit wir das aber können, müssen wir uns auch mit der Eingabe von Werten befassen und damit, wie man sich diese Eingaben für die spätere Verwendung merkt.

6 Eingabe, Ausgabe und Speicherung von Werten

Im letzten Kapitel haben wir gesehen, wie ein C++-Programm grundsätzlich aufgebaut ist. Dieses Gerüst werden wir weiter ausbauen und dabei auch die Eingabe von Werten und deren (mehr oder weniger) dauerhafte Speicherung betrachten.

Eine der wichtigsten Regeln der Informatik in diesem Zusammenhang ist: **Es dürfen nur Werte verwendet werden, die vorher einen sinnvollen Inhalt bekommen haben**. Mit anderen Worten heißt das nichts weiter, als dass es nicht so sinnvoll ist, etwas mittels einer Formel zu berechnen, wenn man die einzelnen Komponenten der Formel noch nicht hat. Ähnliches gilt, wenn man einen Wert erhöht, ohne den Wert vorher einmal festgelegt zu haben.

Wer diese Regeln nicht beachtet, wird sich wahrscheinlich über seine Ergebnisse wundern, der Compiler sieht über solche Fehler aber großzügig hinweg[10], weil er nicht wissen kann, was der Programmierer denn so denkt.

Im Folgenden verwenden wir dieses Gerüst, um darin unsere Experimente durchzuführen. (Die Zeilennummern am Anfang dienen nur zur Übersicht und gehören nicht ins Programm. Man kann aber alle Entwicklungsumgebungen so einstellen, dass sie Zeilennummern anzeigen.):

```
1 #include <iostream>
2 #include <iomanip>
3
4 using namespace std;
5
6  int main () {
7
8  /* Hier ist Platz fuer Experimente */
9
10 }
```

Neu sind die Zeilen 2 und 4, in Zeile 2 wird neben der schon bekannten Bibliothek `iostream` zusätzlich die Bibliothek `iomanip` eingebunden. Der Name steht für Ein- und Ausgabe Manipulationen und liefert alles, was man für eine schöne Ausgabe braucht.

Mit `using namespace std;` ersparen wir uns eine Menge Schreibarbeit und erhöhen so die Übersicht. Durch diese Direktive an den Compiler kann man z.B. das `std::` vor dem `cout` weglassen und es funktioniert trotzdem.

6.1 Ausgabe

6.1.1 Ausgabe von Text

Um Text auszugeben muss dieser in doppelte Hochkomma (oder auch Anführungszeichen " genannt) eingeschlossen sein. Die Anführungszeichen selbst werden nicht mit ausgegeben:

10 Die meisten Compiler sind so eingestellt, dass sie hier eine Warnung (warning) ausgeben, trotzdem aber übersetzen und das Programm ausführen

```
cout << "Dieser Text wird ausgegeben! " << " und dieser Text auch " << " und
dann noch eine neue Zeile: " << endl;
```

Durch diese Anweisung wird unser Text genau so ausgegeben, wie er im Programm steht. Jedes Leerzeichen und jedes Satzzeichen innerhalb der Anführungszeichen landet wie eingegeben auf dem Bildschirm. Und weil die optische Gestaltung des Programmcodes ja keine Rolle spielt liefert

```
cout << "Dieser Text wird ausgegeben! "
<< " und dieser Text auch ";
<< " und dann noch eine neue Zeile: "
<< endl;
```

exakt die gleiche Ausgabe der drei Textstücke in einer Zeile wie das Programmstück weiter oben.

Man nennt solche unveränderlichen Texte *Literale*, dazu gehören auch Zahlenfolgen wie 12345, die unveränderlich im Programm auftauchen.

Aber Vorsicht: es gibt auch Zeichen und Buchstaben, die nicht so auf dem Bildschirm erscheinen, wie man sie eingegeben hat. Dazu gehören in erster Linie die Sonderzeichen der deutschen Sprache wie Umlaute und das ß.

6.1.2 Übung

Probieren Sie aus, was folgende Programmzeile auf dem Bildschirm ausgibt (das Ergebnis hängt von der verwendeten Entwicklungsumgebung und vom Betriebssystem ab, kann also auf unterschiedlichen Rechnern auch unterschiedliche Ergebnisse liefern).

```
cout << "Jürgen hört ohne Unterlaß hämische Gerüchte. " << endl;
```

6.1.3 Ausgabe von Sonderzeichen

Und dann gibt es ja noch die Zeichen, die man nicht ausgeben kann, weil sie als Begrenzungszeichen dienen, etwa die Anführungszeichen " selbst. Selbstverständlich kann man die ausgeben, man muss dem Compiler nur deutlich machen, dass es sich jetzt eben nicht um ein Begrenzungszeichen, sondern um ein normales Zeichen handelt.

Das erledigt man durch einen vorangestellten Backslash \. Um ein Anführungszeichen auszugeben muss man also programmieren:

```
cout << "Das ist ein Anfuehrungszeichen \"!" << endl;
```

und erhält die Ausgabe: `Das ist ein Anfuehrungszeichen "!`

Auf diese Weise sind auch Zeichen auf dem Bildschirm darstellbar, die auf Papier nicht zu erkennen sind. Die folgende Tabelle liefert eine Übersicht:

Zeichen	bedeutet	Zeichen	bedeutet
\a	Alert (Beep)	\b	BackSpace
\t	Tabulator	\n	Neue Zeile (newline)
\v	Vertikaler Tabulator	\f	Seitenvorschub (FormFeed)
\r	Zeilenumbruch	\“	“
\'	'	\?	?
\\	\	\0	Stringende
\ooo	Oktaler Wert	\xhh	Hexadezimaler Wert

Alle Zeichen werden im Rechner nach dem *American Standard Code for Information Interchange*, kurz *ASCII* dargestellt. Das bedeutet für einen Programmierer nur, dass jedes darstellbare Zeichen intern durch eine Zahl zwischen 0 und 127 dargestellt wird. So hat der Großbuchstabe A den Wert 65, der Kleinbuchstabe a den Wert 97. Deutsche (und auch andere nationale) Sonderzeichen kommen in diesem Code nicht vor und lassen sich daher auch nicht so einfach ausgeben.

Inzwischen finden aber Zeichencodierungen wie UTF-8 oder UTF-16 weite Verbreitung, weil immer mehr Programme und Betriebssysteme damit umgehen können. Hier werden die einzelnen Zeichen der Sprache nicht mehr mit nur einem Byte dargestellt, sondern mit zwei oder mehr Byte. Damit sind dann neben den europäischen auch die asiatischen Sprachen abgedeckt. Allen diesen Zeichensätzen ist aber gemeinsam, dass die ersten 127 Byte mit dem ASCII identisch sind.

Der Rechner selbst speichert aber immer nur den numerischen Wert ab, erst die Ausgabe ändert die Darstellung in das gewünschte Zeichen. Und nur die Darstellung! Dadurch lassen sich Zeichen sortieren, weil man ihren ASCII-Wert numerisch vergleichen kann. Und dieser numerische Wert lässt sich auch direkt in der Ausgabe verwenden, indem man den ASCII-Wert *oktal* (mit `\ooo`) oder *hexadezimal* (mit `\xhh`) eingibt. Eine ASCII-Tabelle befindet sich im Anhang auf Seite 195.

6.1.4 Weiterführend: Deutsche Umlaute in der Ausgabe

Umlaute und das ß kann man auf zwei verschiedene Arten in seine Ausgabe integrieren, zum einen durch die Angabe des hexadezimalen Wertes innerhalb des String-Literals (z.B. "Drau\xE1en br\x81llen B\ x84ren".

Zeichen	Hexadezimal
ä	\x84
Ä	\x8e
ü	\x81
Ü	\x9a
ö	\x94
Ö	\x99
ß	\xe1

Einfacher ist es aber mittels der Anweisung

```
setlocale(LC_ALL, "");
```

Dann kann man direkt die deutsche Tastatur benutzen.

Und wenn unerwünschte Effekte auftreten, kann man mittels

```
setlocale(LC_ALL, "C");
```

alles wieder zurück stellen.

Die Variante mittels hexadezimaler Schreibweise ist zwar umständlicher, hat aber keine Auswirkungen auf alle sonstigen Funktionen bei der Behandlung und Verarbeitung von Zeichenketten.

Setzt man auf setlocal(...) wird im Hintergrund der Zeichensatz umgeschaltet, damit ändert sich auch die Zuordnung von Zeichen und zugeordneter Position im ASCII (oberhalb des Zeichens 127). Die Werte in der Ersetzungstabelle oben gelten also dann genau nicht mehr.

6.2 Formatierung von Zahlenwerten

Hauptsache schön- so soll die Ausgabe sein. Und das zu Recht. Das haben auch die Entwickler von C++ eingesehen und die Bibliothek `iomanip` ersonnen. Darin finden sich Steueranweisungen, die man in den Ausgabestrom einbaut und sich auf die nachfolgenden Ausgaben entsprechend auswirken.

Und diese Manipulatoren zur Zahlendarstellung kennt `iomanip` :

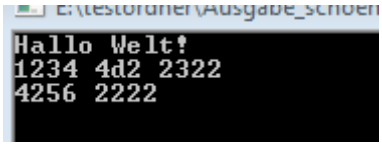
Manipulator	Wirkung
`oct`	Oktale Darstellung
`hex`	Hexadezimale Darstellung
`dec`	Dezimale Darstellung
`showpos`	Positive Zahlen mit Vorzeichen ausgeben
`noshowpos`	Positive Zahlen werden ohne Vorzeichen ausgegeben (Standard)
`uppercase`	Bei der Ausgabe von Hexadezimalzahlen werden Großbuchstaben A-F verwendet
`nouppercase`	Bei der Ausgabe von Hexadezimalzahlen werden Kleinbuchstaben a-f verwendet (Standard)

Diese Manipulatoren werden einfach wie jede andere Ausgabe in den Ausgabestrom hinein geschoben und wirken sich solange aus, bis sie mit einem gegenteiligen Befehl wieder aufgehoben oder geändert werden. Man spricht auch von *persistenten* Anweisungen, weil sie ihre Wirkung nicht am Zeilenende verlieren.

Das folgende Programmstück (mit dem Rahmenprogramm von Seite 44) demonstriert die Wirkung:

```
cout << "Hallo Welt!" << endl;

cout << dec << 1234 << " " << hex << 1234 << " " << oct << 1234 << " " <<
endl;
```

```
cout << 2222 << " " << dec << 2222 << endl;
```

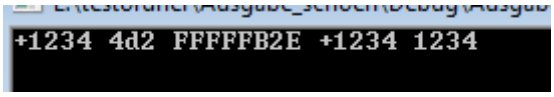


In der zweiten Zeile (nach "Hallo Welt!") wird dreimal der dezimale Wert 1234 ausgegeben, zunächst dezimal, dann hexadezimal und am Zeilenende noch oktal. Diese Einstellung auf oktal wirkt auch in der letzten Zeile noch nach, erst die zweite Ausgabe erscheint wieder als Dezimalzahl, weil erst unmittelbar davor mit `dec` wieder auf dezimale Darstellung umgeschaltet wird.

Und hier noch die Demonstration der anderen Manipulatoren:

```
 cout << dec << showpos << 1234 << " " << hex << 1234 << " " << uppercase << -1234 << " " << dec << 1234 << " ";
  cout << noshowpos << dec << 1234 << " " << endl;
```

liefert:



Auch hier sieht man, dass die Manipulatoren persistent arbeiten und nach Gebrauch wieder deaktiviert werden sollten. Außerdem sieht man, dass Hexadezimalzahlen kein Vorzeichen kennen und negative Zahlen als *Zweierkomplement* dargestellt werden.

Es gibt eine weitere Gruppe von Manipulatoren, die die Ausgabe von Fließkommazahlen steuern:

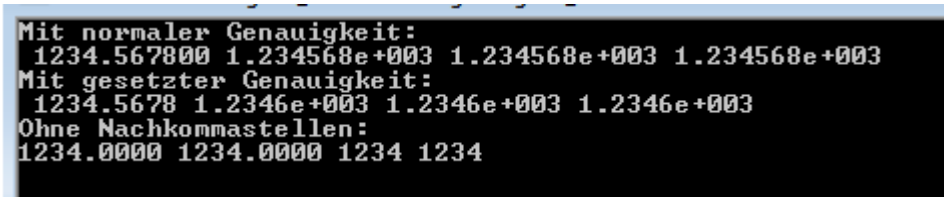
Manipulator	Wirkung
`showpoint`	Der Dezimalpunkt wird immer angezeigt. Es werden so viele Ziffern angezeigt, wie die eingestellte Genauigkeit ermöglicht
`noshowpoint`	Nachfolgende Nullen werden nicht angezeigt, gibt es keinen Nachkommateil, wird auch der Dezimalpunkt unterdrückt
`fixed`	Darstellung erfolgt als Festpunktzahl
`scientific`	Darstellung erfolgt in Exponentialschreibweise
`setprecision(int n)`	Setzt die Genauigkeit auf n Stellen

Folgendes Programmstück verdeutlicht die Tabelle:

```
  cout << "Mit normaler Genauigkeit: \n " << fixed << 1234.5678 << " " << scientific << 1234.5678 << " ";
  cout << showpoint << 1234.5678 << " " << noshowpoint << 1234.5678 << " ";
  cout << setprecision(4) << "\nMit gesetzter Genauigkeit:\n ";
  cout << fixed << 1234.5678 << " " << scientific << 1234.5678 << " ";
  cout << showpoint << 1234.5678 << " " << noshowpoint << 1234.5678 << " ";
```

```
  cout << "\nOhne Nachkommastellen: \n";
  cout << showpoint << fixed << 1234.000 << " " << noshowpoint << 1234.000
<< " ";
  cout << showpoint << fixed << 1234 << " " << noshowpoint << 1234 << " ";
```

Das Ergebnis ist nicht immer so, wie es nach den Aussagen in der Tabelle erwartet wird. Hier hilft nur ausprobieren:



Hier kann es dann durchaus zu unterschiedlichen Aussehen bei unterschiedlichen Compilern kommen (Compiler ist nicht die Entwicklungsumgebung, sondern wird von dieser aufgerufen und übersetzt den Quellcode tatsächlich in eine für die Maschine lesbare Sprache).

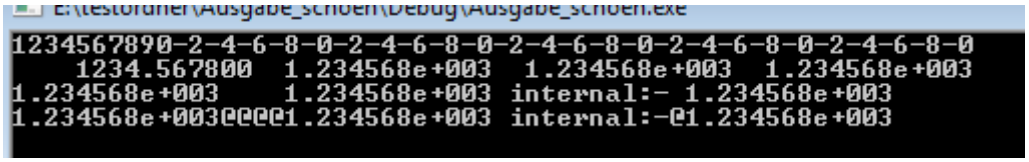
Fehlt noch die dritte Gruppe mit den Manipulatoren für eine tabellarisch formatierte Ausgabe:

Manipulator	Wirkung
`setw (int n)`	Setzt die Feldbreite auf n Zeichen
`setfill (char ch)`	Setzt das Füllzeichen auf ch
`left`	Linksbündige Ausgabe innerhalb des Feldes
`right`	Rechtsbündige Ausgabe innerhalb des Feldes
`internal`	Vorzeichen linksbündig, Wert rechtsbündig innerhalb des Feldes

Auch hier ein Stück Programm zur Verdeutlichung:

```
  cout << "1234567890-2-4-6-8-0-2-4-6-8-0-2-4-6-8-0-2-4-6-8-0-2-4-6-8-0" <<
endl;
  cout << setw(15) << fixed << 1234.5678 << setw(15) << scientific <<
1234.5678;
  cout << setw(15) << 1234.5678 << setw(15) << noshowpoint << 1234.5678 <<
endl;
  cout << setw(15) << left << 1234.5678 << setw(15) << right << 1234.5678
<<" internal:" << setw(15) << internal << -1234.5678 << endl;
  cout << setfill('@');
  cout << setw(15) << left << 1234.5678 << setw(15) << right << 1234.5678
<<" internal:" << setw(15) << internal << -1234.5678 << endl;
```

Auch hier muss man sich genau ansehen, welchen Effekt welcher Manipulator hat oder eben auch nicht. Eine Besonderheit gibt es bei `setw`(n). Dieser Manipulator ist (genau wie endl) nicht persistent, d.h. er wirkt nur auf die nächste Ausgabe und nicht weiter, noch nicht einmal bis zum Zeilenende. Für den Aufbau von Tabellen muss man dies also bedenken.



6.3 Datentypen und Eingabe von Werten

Nahezu alle Programme arbeiten nach dem sogenannten *EVA-Prinzip*. EVA steht dabei für die Begriffe Eingabe, Verarbeitung und Ausgabe. Und dieses Prinzip gilt ganz konkret und jederzeit, denn es macht keinen Sinn, erst etwas zu verarbeiten und danach erst die benötigten Eingaben zu machen. Ebenso wenig erhält man eine richtige Ausgabe, wenn entweder die Verarbeitung oder die Eingabe fehlen. Ein Auto bringt einen auch nur ans Ziel, wenn man erst einsteigt, dann den Motor startet und dann losfährt.

Es ist ganz wichtig, sich dieses Prinzip immer wieder klar zu machen, wenn ein Programm nicht das tut, was man erwartet oder merkwürdige Ergebnisse liefert. Sehr oft liegt es dann an der Missachtung des EVA-Prinzips.

Um einem Programm die Möglichkeit zu geben, eine Eingabe zu verarbeiten, muss man eine Aufbewahrung für die gemachte Eingabe schaffen. Das ist ähnlich wie beim Einkaufen: 10 Eier sind sehr viel leichter unterzubringen, wenn sie in einem passenden Karton gelagert werden. Bevor wir also auf die Eingabe direkt zu sprechen kommen, müssen wir uns ein wenig mit den Hintergründen beschäftigen.

In der EDV nennt man Lagerplätze für Werte aller Art *Speicher*. Das sind in Wirklichkeit Milliarden von elektronischen Schaltungen, die sich entweder im Zustand an oder aus befinden. Und weil es so viele sind, fasst man immer einige davon (üblicherweise 8) zusammen und gibt ihnen eine Hausnummer, die in der Fachsprache als *Adresse* bekannt ist. Und je nach Umfang und Größe eines Wertes werden ein oder mehrere solcher Adressen benötigt, um einen Wert abzuspeichern.

Nun ist es ziemlich viel von einem Programmierer verlangt, sich zu merken, welchen Wert er unter welcher Adresse abgelegt hat. Vor allem bei vielen Werten kann das ganz schnell unübersichtlich werden. Alle modernen Programmiersprachen können daher Speicherbereiche mit einem *Namen* versehen, damit der Programmierer sich nur noch diesen *Bezeichner* merken muss, ohne zu wissen, welche absolute Speicherstelle damit gemeint ist.

Wie so ein Bezeichner aussieht, haben wir bereits auf Seite 40 erfahren. Zur Erinnerung nur so viel: Ein Bezeichner sollte einen sprechenden Namen haben, damit auch jeder, der ein Programm liest, erkennt, um was es sich handelt. Jetzt geht es um die unterschiedlichen Inhalte, die man im Speicher ablegen kann, diese unterschiedlichen Inhalte heißen *Datentypen*.

6.4 Elementare Datentypen

Wie viel Platz man für einen Wert (ein Datum) benötigt, hängt davon ab, welchen *Wertebereich* dieses Datum abdecken soll. Für ein einfaches Ja oder Nein reicht ein einziges *Bit* aus, das ist nichts anderes als eine elektronische Schaltung, bei der entweder Strom fließt oder eben nicht. Davon gibt es in einem Rechner ein paar Milliarden, daher hat man sich angewöhnt, immer acht davon zusammenzufassen und diese 8 Bit als Gruppe zu verwenden und den Titel *Byte* zu geben. Ein Byte ist die kleinste im Rechner adressierbare Einheit.

6.5 Ganze Zahlen

Da ein Byte aus acht einzelnen Bit (Schaltern) besteht, gibt es 2^8 verschiedene Zustände, die ein Byte annehmen kann. Damit lassen sich alle Buchstaben des westlichen Alphabets sowie Ziffern und Satzzeichen abbilden. Für Zahlen taugt dieses eine Byte aber kaum, man kommt halt gerade mal von 0 bis 255 (also 256 verschiedenen Zuständen). Jeder Grundschüler kennt einen größeren Zahlenraum.

Für Zahlen, die einen größeren Wertebereich umfassen, muss man also mehrere Byte zusammenfassen. Grundsätzlich könnte man für jede Ziffer ein Byte verwenden und dann mit zehn Byte die Zahlen von 0 bis 999999999 abbilden. Das wiederum ist aber eine ziemliche Verschwendung, weil rund 96% der Möglichkeiten ungenutzt bleiben. Man greift daher zu einem Trick und rechnet alle Zahlen zunächst ins *Dualsystem*, also eine Folge von Nullen und Einsen, um und packt jeweils acht Stück davon in ein Byte. Mit zwei Byte lassen sich so Zahlen darstellen, die von 0 bis 65535 (2^{16}) gehen Braucht man mehr, nimmt man sogar vier Byte zusammen und kommt auf einen Wertebereich von 0 bis 4294967295 (2^{32}). Es gibt auch einen Datentyp mit acht Byte, der geht dann sogar bis 18446744073709551615.

Stellt sich die Frage, was man mit negativen Zahlen macht. Dazu wird ein Bit geopfert und zum *Vorzeichenbit* umdefiniert. Bei vier Byte bleiben dann zwar nur noch 31 Bit für die Zahl, dafür aber gibt es jetzt ein Vorzeichen. Der Wertebereich beginnt also jetzt bei -2147483648 und endet bei +2147483647. Bei den zwei und acht Byte langen Werten ändert sich der Wertebereich entsprechend.

Jeder dieser Wertebereich bekommt in C++ einen Namen, den *Datentyp*. Je nach Länge spricht man von `short`, `int`, `long` oder `long long`. Benötigt man nur positive Werte, kann man das Vorzeichenbit wieder dem Wertebereich zuführen, das erkennt man an einem vorangestellten `unsigned`. Die Tabelle gibt einen Überblick über die elementaren Datentypen für ganze Zahlen:

Datentyp	Speicherbedarf	Wertebereich
`char`	1 Byte	-128 bis 127 oder 0 bis 255
`unsigned char`	1 Byte	0 bis 255
`int`	meistens 4 Byte selten 2 Byte	-2147483648 bis +2147483647 -32768 bis 32767
`short`	2 Byte	-32768 bis 32767
`unsigned short`	2 Byte	0 bis 65535
`long`	4 Byte	-2147483648 bis +2147483647
`unsigned long`	4 Byte	0 bis 4294967295
`long long`	8 Byte	-9223372036854775808 bis 9223372036854775807
`unsigned long long`	8 Byte	0 bis 18446744073709551615

Wenn man sich die Tabelle genau anschaut, erkennt man sehr schnell, dass mal wieder auf nichts Verlass ist. Die Datentypen `int` und `long` haben unter Umständen den gleichen Wertebereich. Denn es gibt in C++ keine genauen Vorschriften darüber, welcher Datentypen welchen Wertebereich umfasst, sondern nur Empfehlungen. Die einzige Vorschrift, die hier gilt ist: `char` <= `short` <= `int` <= `long` <= `long long`. Außerdem ist der Datentyp `short` mindestens 2 Byte und der Typ `long` mindestens 4 Byte lang. Die konkreten Wertebereich hängen vom verwendeten Rechner und dem Betriebssystem und dem Compilerhersteller ab. Hier hilft in Zweifelsfällen ein Blick ins Handbuch oder in die Headerdatei `climits`.

Der Datentyp `char` umfasst in jedem Fall 1 Byte, kann aber je nach Hersteller von -127 bis 128 oder von 0 bis 255 reichen. Auch das ist in C++ nicht festgelegt.

Vorteil der ganzzahligen Datentypen ist, dass es keinerlei Ungenauigkeiten bei der Umwandlung vom dezimalen ins duale Zahlensystem gibt, jede Zahl wird exakt dargestellt. Das sieht bei den Datentypen für

6.6 Gleitpunktzahlen

ganz anders aus. Eine Gleitpunktzahl entspricht mathematisch einer gebrochen rationalen Zahl oder *reellen Zahl*. Man erkennt sie leicht am Dezimalpunkt **(Achtung: Zahlen werden in englischer Schreibweise dargestellt, also mit einem Dezimalpunkt statt Komma, unabhängig von der Darstellung durch das Betriebssystem)**. Auch hier gibt es unterschiedliche Datentypen für unterschiedliche Wertebereiche:

Datentyp	Speicherbedarf	Wertebereich (Genauigkeit)
`float`	4 Byte	$\pm 3.4*10^{+38}$ bis $\pm 1.2*10^{-38}$ (bei 6 Stellen)
`double`	8 Byte	$\pm 1.7*10^{+308}$ bis $\pm 2.3*10^{-308}$ (bei 15 Stellen)
`long double`	10 Byte	$\pm 1.1*10^{+4932}$ bis $\pm 3.4*10^{-4932}$ (bei 19 Stellen)

Die Genauigkeit wird in Anzahl der Dezimalstellen (nicht Anzahl der Nachkommastellen!) angegeben. So bedeutet *6 Stellen genau,* dass zwei Gleitpunktzahlen, die sich innerhalb der ersten sechs Dezimalstellen unterscheiden, auch verschieden gespeichert werden. Die Ziffern 12.3456 und 12.345678 sind bei sechs Stellen Genauigkeit also gleich groß. Intern werden von den verfügbaren 32 Bit (bei `float`) 23 Bit für die Mantisse, 8 Bit für den Exponenten und 1 Bit für das Vorzeichen verwendet. Trotz diverser mathematischen Tricks lässt sich aber eine gewisse Ungenauigkeit bei der Umwandlung von dezimalen Gleitpunktzahlen in die interne Darstellung nicht vermeiden.

6.7 Wahrheitswerte (bool)

Hierbei handelt es sich um den Datentyp mit dem geringsten Wertebereich. Gerade mal zwei Zustände kann ein Element vom Typ `bool` annehmen: `true` oder `false`. Da C dafür keinen echten, eigenen Datentyp kannte, stand einfach eine 0 für `false` und jeder andere Wert für `true`. Dieses hat sich auch bei C++ nicht geändert, alles außer 0 ist `true`. Diese Tatsache nutzen manche Programmierer gnadenlos aus, um schwer nachvollziehbare Bedingungen zu formulieren.

6.8 Aufzählungstypen (enum)

Ganz anders dagegen bei diesem Datentyp, hier kann der Programmierer selbst nach Herzenslust den Wertebereich festlegen und nach seinem Gutdünken gestalten (na ja, zumindest sehr frei). Man muss einfach nur eine Folge von Begriffen (im weitesten Sinne Bezeichner) aufschreiben und schon ist man fertig.

Intern werden die aufgeführten Begriffe einfach durchnummeriert, beginnend bei 0. So ist es möglich, einer `int`-Variablen jederzeit einen Wert des Aufzählungstyps zuzuweisen, aber leider nicht umgekehrt. Das sieht dann so aus:

```
enum Farben {
            rot, gelb, gruen, blau, orange, cyan, magenta, braun, weiss,
            schwarz, lila, violett
      };


      Farben pinsel = blau;
      int i = 0;
      i = rot + gelb;   // Aber nicht: pinsel = rot + gelb;
      pinsel = weiss;   // Aber nicht: pinsel = i;
      cout << i << endl;
      cout << pinsel << endl;
```

Nur eines funktioniert leider nicht wie man das erwartet: man kann sie nicht einfach so ein- oder ausgeben. Auf dem Bildschirm erscheint nicht etwa „weiss“, wie man es nach der letzten Anweisung erwarten würde, sondern eine einfache 8 (zählen Sie nach!). Wie man das ändert, finden Sie auf Seite 190.

Aufzählungstypen benutzt man häufig, um Zustände eines Systems zu beschreiben. Eine Ampelsteuerung liest sich viel angenehmer, wenn die Ampel rot, gelb oder gruen zeigt und nicht 0,1 oder 2.

6.9 Variablen und Konstanten

Nachdem wir jetzt die wichtigsten Datentypen kennen gelernt haben können wir endlich daran gehen, auch mit Werten zu arbeiten. Im Abschnitt Ausgabe haben wir alles Wichtige zur Ausgabe gesehen, jetzt kümmern wir uns um das Gegenteil, die *Eingabe* von Werten.

In C++ unterscheiden sich Ein- und Ausgabe nicht sonderlich, auch intern arbeiten sie sehr ähnlich. Das macht es für uns einfacher. Das Ziel der Eingabe ist aber hier vom Programmierer anzugeben, nämlich der Name der Speicherstellen, in denen die Eingabe landen soll. Und zu diesem Namen und seinem Datentyp müssen die Daten passen, die von der Tastatur geliefert werden.

Wir müssen also Variablen *deklarieren und definieren*, also einem Speicherbereich einen Datentyp und einen Namen geben. Das passiert sinnvollerweise *bevor* man den Speicherplatz dann benötigt.

```
int main() {
  int ganzeZahl, kleineZahl;
  long lange_Zahl;
  float reelleZahl, pi;
  double grosseZahl;
  // hier kommt das Programm
  return 0;
}
```

Die Struktur ist selbsterklärend: erst kommt der *Datentyp*, danach eine durch *Komma* getrennte *Liste* der Variablen dieses Typs. Am Ende der Liste steht ein *Semikolon*, wie (fast) immer in C++. Natürlich kann man auch jede Variable in eine eigene Zeile packen, dann hat man die Möglichkeit direkt dahinter mit einem *Kommentar* die Bedeutung der Variablen zu erläutern.

In C++ wird häufig nicht zwischen *Deklaration* und *Definition* einer Variablen unterschieden. Streng genommen macht die Deklaration lediglich den Namen und Typ einer Variablen bekannt, bei der Definition wird zusätzlich der dafür auch benötigte Speicher reserviert.

Unter *Initialisierung* einer Variablen versteht man dann die Zuweisung eines Wertes, erst danach kann sinnvoll mit der Variablen gearbeitet werden

In vielen Programmen benötigt man Werte, die zu Beginn des Programms bereits feststehen und sich auch während der Programmausführung **nicht** ändern sollen oder dürfen, die aber zur Berechnung oder Verarbeitung benötigt werden. Diese Werte nennt man *Konstanten*, die Klassiker wie PI oder andere naturwissenschaftlichen Größen sind ja allgemein bekannt. Solche Speicherstellen müssen bei der Deklaration auch gleich mit Inhalt gefüllt werden, eine spätere Änderung ist nicht mehr möglich, weder mit Absicht, noch aus Versehen.

```
int ganzeZahl;
const int NMAX=20;
double p;
const double PI=3.1415926;
```

Konstanten haben also neben einem Datentyp auch noch sofort einen *unveränderlichen Wert.* Es hat sich eingebürgert, die Namen von Konstanten komplett in *Großbuchstaben* zu schreiben.

Schließlich gibt es noch die Möglichkeit, jeder Variablen einen *Anfangswert* zu geben. Das hat den Vorteil, dass sich Fehler schneller eingrenzen lassen, weil sonst nämlich jede Variable einen zufälligen Wert enthält (der Speicher wird beim Einschalten des Rechners nicht komplett gelöscht, sondern enthält irgendwelche Bit-Werte, die dann entsprechend des angegebenen Datentyps interpretiert werden). Die Deklaration ist nicht überraschend:

```
int ganzeZahl;
int max=20;
long sehr_grosse_zahl=2147000999;
```

Sie ist aufgebaut wie eine Konstantendeklaration nur eben ohne das Wörtchen `const`.

Für die Eingabe steht in C++ die Anweisung `cin` und der Eingabeoperator >> zur Verfügung. Alles andere sieht verdächtig nach Ausgabe aus:

```
cin >> lange_Zahl;
cin >> ganzeZahl >> kleineZahl >> grosseZahl;
```

Ganz analog zur Ausgabe kann man mit einer Eingabe gleich mehrere Werte eingeben, die Trennung bei der Eingabe erfolgt dabei durch mindestens ein Leerzeichen. Das Ende der Eingabe wird mit der `Enter`-Taste angezeigt.

Wer sich die Mühe macht, und aus diesen beiden Fragmenten ein Programm bastelt wird mit Erschrecken feststellen, dass auf dem Bildschirm nur ein kleiner blinkender Strich erscheint und sonst nichts. Benutzerfreundlich ist das nicht.

6.10 Benutzerfreundliche Eingaben gestalten

Wer mehr als nur einen blinkenden Strich (Cursor) will, muss das von Hand erledigen und schlicht in sein Programm hinein schreiben. Es gilt hier ein elementarer Grundsatz: **Was auch immer ein Programm irgendwann einmal tun soll, muss vorher in dieses Programm hinein programmiert werden.**

In unserem obigen Fragment weiß der Anwender des Programms überhaupt nicht, was er mit dem Cursor anfangen soll. Selbst wenn er darauf kommt, dass jetzt eine Eingabe erwartet wird, weiß er nicht, welche Eingaben in welcher Form erfolgen sollen. Damit sind Eingabefehler garantiert.

Es bietet sich also an, vor jeder Eingabe zunächst einen erklärenden Text auszugeben und dahinter oder danach die Eingabe zu erwarten. Es findet also ein ständiger Wechsel zwischen Aus- und Eingabe statt:

```
  cout << "Bitte eine ganze Zahl eingeben: ";
  cin >> lange_Zahl;
  cout << "Bitte zwei ganze Zahlen und eine Gleitpunktzahl eingeben: ";
  cin >> ganzeZahl >> kleineZahl >> grosseZahl;
```

Das sieht auf dem Bildschirm dann so aus:

```
Bitte eine ganze Zahl eingeben: 5
Bitte zwei ganze Zahlen und eine Gleitpunktzahl eingeben: 4 5 22.6
```

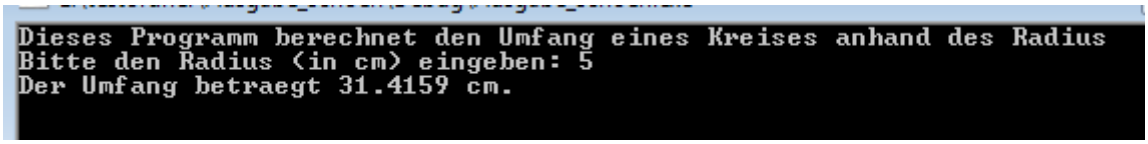
Man beachte: bei den `cout` steht nirgends ein `endl`, trotzdem beginnt die nächste Ausgabe in einer neuen Zeile. Dafür sorgt die `Enter`-Taste nach der Eingabe. Und hier lauert ein häufiger Fehler:

`cin >> var1 >> endl;` ist verboten!!! Hier wird Ein- und Ausgabe gemischt, das geht schief.

An diesem Beispiel sieht man aber auch, wie wichtig es ist, seinen Variablen sprechende Namen zu geben. Das folgende Programm verdeutlicht, wie es richtig geht:

```
int main() {
  double radius;
  const double PI = 3.1415926; // besser: const double PI = 4*atan(1);
                               // das liefert PI in Rechnergenauigkeit
  cout << "Dieses Programm berechnet den Umfang eines Kreises anhand des Radius" <<
endl;
  cout << "Bitte den Radius (in cm) eingeben: ";
  cin >> radius;
  cout << "Der Umfang betraegt " << 2*PI*radius << " cm." << endl;
  return 0;
}
```

und liefert folgende Ausgabe:



6.11 Arithmetische Operatoren

Einfache Rechenoperationen haben wir ja bereits eingesetzt, jetzt kennen wir aber auch viele Datentypen und können uns um die möglichen *Operationen* darauf kümmern. Es sind mehr wie man erwartet!

Der bisher behandelte Fall geht von zwei *Operanden* (linker und rechter *Operand*) und dem dazwischen stehenden *Operator* aus. Die vier Grundrechenarten fallen in diese Kategorie: a+b, 22-7 usw. Wenn mehr als zwei Operanden und mehr als ein Operator beteiligt sind (*Ausdrücke*, *Terme*), wird immer stückweise von links nach rechts gearbeitet. Auch im PC gelten die üblichen *Vorrangsregeln* wie *Punktrechnung* geht vor *Strichrechnung*. Und wenn man das anders braucht muss man *Klammern* setzen, alles wie gehabt: 3+a-(47+42)*4. Hier die mathematischen Operatoren in der Übersicht:

+	Addition
–	Subtraktion
/	Division
*	Multiplikation
%	Modulo (ergibt den Rest bei einer ganzzahligen Division)

Neben diesen *binären* Operatoren gibt es aber auch einige *unäre* Operatoren, die mit nur einem Operanden arbeiten. Auch hier die Übersicht:

+	Vorzeichen, positiv
–	Vorzeichen, negativ
++	Inkrementoperator (Wert um 1 erhöhen)
––	Dekrementoperator (Wert um 1 erniedrigen)
&	Adressoperator (Adresse von...)
*	Verweis (zeigt auf...)
!	Nicht / not

Die *Vorzeichenoperatoren* + und - müssen nicht extra erläutert werden, der *Inkrement*- und *Dekrementoperator* schon. Diese tauchen immer in Verbindung mit einer Variablen auf (nicht mit Konstanten oder festen Werten!) und *erhöhen* (bzw. *erniedrigen*) den Wert der Variablen um 1.

Das Besondere dran ist, dass diese Operatoren vor (*Präfix*) oder nach (*Postfix*) der Variablen stehen und dann unterschiedliche Bedeutung haben. Handelt es sich um die Präfixversion (also `++var`) wird bei der Programmausführung der aktuelle Wert der Variablen var aus dem Speicher geholt, um 1

erhöht/erniedrigt und dann erst an das Programm zur weiteren Verarbeitung übergeben. Danach wird der geänderte Wert unter var wieder abgespeichert.

Steht der Operator hinter der Variablen (Postfix, also `var++` oder `var--`) wird bei der Programmausführung der Wert der Variablen aus dem Speicher gelesen und unverändert an das Programm zur Verarbeitung weiter gegeben. Danach wird der Wert erhöht/erniedrigt und dieser geänderte Wert wieder im Speicher abgelegt.

Nach einer Anweisung mit Inkrement/Dekrement ist die verwendete Variable auf jeden Fall um 1 höher/niedriger, aber je nach Art des Operators (Präfix oder Postfix) wurde bereits der geänderte Wert oder der originale Wert zur Berechnung verwendet

Das zeigt das folgende Programm:

```
int main() {
  int i = 2;
  int j = 8;
  cout << i++ << endl;
  cout << i << endl;
  cout << j-- << endl;
  cout << --j << endl;
  return 0;
}
```

mit folgenden Ergebnissen:

Um die Operatoren vollständig zu kennen, fehlen noch einige, die so nicht in der normalen Mathematik vorkommen, sondern sich aus der technischen Abbildung von Daten in elektronische Schaltkreise ableiten:

a << b	Schiebeoperator links (schiebt die Bit-Darstellung von a um b Stellen nach links und fügt für jeden Schritt eine 0 an, das entspricht einer Multiplikation mit 2^b)
a >> b	Schiebeoperator rechts (schiebt die Bit-Darstellung von a um b Stellen nach rechts und fügt vorne jeweils eine 0 an, das entspricht einer Division durch 2^b)
a = b	Zuweisungsoperator (a ergibt sich zu b)
a += b,a -= b, a *= b, a /= b	Kurzschreibweise für a=a+b, a=a-b, a=a*b und a=a/b
a & b	Bitweise UND-Verknüpfung
a ^ b	Bitweise EXCLUSIV ODER Verknüpfung
a | b	Bitweise ODER Verknüpfung
~ a	Bitweise NEGATION

Wie man hier auch wieder sehen kann, werde manche Operatoren unterschiedlich verwendet, die Bedeutung hängt dann immer von der aktuellen Position (dem Kontext) ab.

Im nächsten Kapitel werden die folgenden Vergleichsoperatoren eine wichtige Rolle spielen:

= =	Gleich (identisch)
!=	Ungleich (nicht gleich)
>	Größer
>=	Größer gleich
<	Kleiner
<=	Kleiner gleich
&&	Logisches UND (von Ausdrücken)
||	Logisches ODER (von Ausdrücken)
!	Logisches NICHT (bei Ausdrücken)

Last but not least die Operatoren, die es in C++ auch gibt, die wir aber hier nur der Vollständigkeit erwähnen, ohne sie weiter zu betrachten[11].

`::  []  .  ->  .*  ->*`	Zugriffsoperatoren
`(typ)  dyamic_cast<>  static_cast<> const_cast<>  reinterpret_cast<>`	Cast-Operatoren
`new  new[]  delete  delete[]`	Operatoren zur Speicherverwaltung
`?:  &  name()  type()  sizeof() typeid()`	Sonstige Operatoren

Operatoren sind grundsätzlich *reservierte* Begriffe und sollten nicht mit eigenem Inhalt belegt werden, das kann zu großen Missverständnissen führen. Vermeiden Sie also nach Möglichkeit, eine Variable `new` oder `delete` zu nennen.

6.12 Weitere elementare Datentypen

Auch wenn Zahlen die ursprüngliche Domäne von Computern sind, sie können auch mit Texten und anderen Zeichen umgehen.

11 Einige dieser Operatoren werden wir später in den Kapiteln über dynamischen Speicher und Zeiger wiedersehen.

Um ein einzelnes Zeichen in einer Variablen abzuspeichern gibt es den Datentyp `char`. Einzelne Zeichen werden in einfache Hochkomma eingeschlossen, um sie von Variablen mit einem gleichlautenden Namen zu unterscheiden. Eine Definition sieht dann so aus:

```
char ch = 'x';
```

Und alles, was weiter oben über die Ein- und Ausgabe von Zahlen gesagt wurde gilt auch hier. Eine Eingabe sieht also so aus:

```
cin >> ch;
```

Bei der Eingabe werden die Hochkommata natürlich nicht mit eingegeben.

Intern werden alle Zeichen als ASCII-Wert gespeichert, sind also ganze Zahlen. Daher kann der Datentyp `char` auch mit `int` hin und her gewandelt werden. Auch Arithmetik ist mit Zeichen möglich! Vorsicht ist bei solchen Aktionen geboten, weil der Datentyp `char` je nach Compiler mal mit und mal ohne Vorzeichen dargestellt wird. Das gibt dann Warnungen beim Übersetzen.

Geht es um ganze Worte (also mehrere Zeichen am Stück) verwendet man den Datentyp `string`. Dabei handelt es sich aber nicht um einen elementaren Datentyp sondern um ein Konstrukt aus mehreren `char`. Der Einsatz erfolgt aber ganz analog zu den `char`, hier wird zur Unterscheidung allerdings das Anführungszeichen (oder auch doppeltes Hochkomma genannt) benutzt.

```
string s = "Hallo Welt!";
cout << s;
```

Um diesen Datentyp verwenden zu können, muss man am Anfang des Programms mit `#include <string>` die passende Bibliothek einbinden. Die genauere Betrachtung dieses Datentyps[12] erfolgt im Abschnitt 9.2 auf Seite 101.

6.13 Zusammenfassung

Wir kennen jetzt alle Anweisungen, um C++-Programme zu schreiben, die aus mehr oder weniger beliebigen Eingaben nach einfachen Rechenvorschriften Ausgaben ermitteln. Das EVA-Prinzip (Eingabe-Verarbeitung-Ausgabe) wird beachtet. Die optische Gestaltung der Ausgaben ist kein Buch mit sieben Siegeln mehr. Das wird jetzt geübt.

6.14 Übungen

Kreisring

Schreiben Sie ein Programm, dass nach Eingabe des Radius r den Umfang und die Fläche eines Kreises berechnet. Erweitern Sie das Programm dann so, dass nach Eingabe eines zweiten Radius r2 die Fläche des Kreisrings zwischen r und r2 ausgegeben wird (r2 sollte kleiner wie r sein).

Temperaturumrechnung

Schreiben Sie ein Programm, dass eine in Grad Celsius einzugebende Temperatur in Grad Fahrenheit umrechnet. Die Formel dazu lautet $F = C * \frac{5}{9} + 32$

12 Streng genommen handelt es sich hier um eine Klasse und nicht um einen Datentyp, der Unterschied spielt aber hier keine Rolle. Darüber hinaus gibt es noch die aus der Programmiersprache C klassische Zeichenkette, die aber heute nur noch in Sonderfällen eine Rolle spielt

Kap. 6

Schreiben Sie das analoge Programm, welches aus einer Eingabe in Grad Fahrenheit die Temperatur in Grad Celsius berechnet.

Aufmerksamkeit erzeugen

Schreiben Sie ein Programm , dass folgende Ausgaben macht:

```
Ungueltiger Dateiname: D:\testordner\Kapitel_var\ueb's.cpp
```

Danach soll eine Eingabe erfolgen, die sich lautstark (per Beep) bemerkbar macht:

```
Ihre Eingabe:
```

Variablen definieren

Für die Berechnung eines Rechteckes sind die Variablen `seitenlaenge`, `umfang` und `flaeche` erforderlich. Definieren Sie die Variablen und initialisieren Sie `seitenlaenge` mit dem Wert 1.0.

ASCII-Werte

Schreiben Sie ein Programm, das die Zeichen mit dem ASCII-Code 74 und 83 ausgibt.

7 Ablaufsteuerung

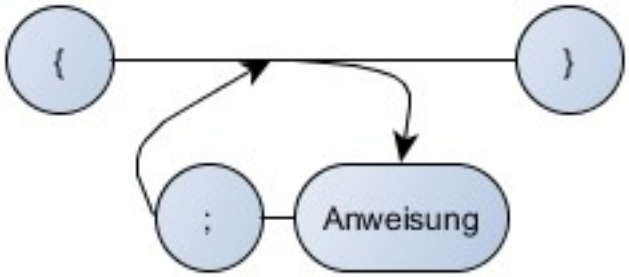
Allen Programmen, die wir bisher gesehen und selbst geschrieben haben ist eines gemeinsam. Man steckt etwas in sie hinein und bekommt ein Ergebnis. Jegliche weitere Einflussnahme ist ausgeschlossen. Man spricht in diesem Zusammenhang von linearen Programmen. Das ist nicht nur langweilig, sondern auch wirklichkeitsfremd. Ein Programm muss mehr als eine Eingabe verkraften und dann auch verschiedene Ergebnisse liefern können.

Neben diesem linearen Programmablauf muss es also noch etwas anderes geben.

Diese Ablaufstrukturen lernen wir jetzt kennen.

7.1 Block

Eine Folge von Anweisungen bildet nach außen einen sogenannten Block und verhält sich dann wie eine einzige Anweisung. Grafisch sieht das so aus:



Jeder Block beginnt immer mit einer öffnenden (linken) geschweiften Klammer und endet im einfachsten Fall sofort wieder mit einer schließenden (rechten) geschweiften Klammer. Damit enthält der Block nichts. Darüber darf man sich nicht wundern, ein leerer Block ist in manchen Fällen eine echte Notwendigkeit und hilft einem beim Programmieren aus der Klemme. Besonders bei größeren Programmen kann man damit so tun, als ob gewisse Teile schon fertig wären und den Rest des Programms schon testen.

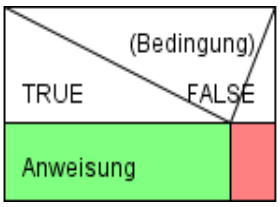
Interessanter ist der längere Weg: nach der linken Klammer kommt eine Anweisung, ein Semikolon und bei Bedarf wiederholt sich dies beliebig oft. Hier sind jetzt alle Arten von Anweisungen erlaubt. Das klingt etwas trivial, hat aber eine große Bedeutung: so darf ein Block selbst natürlich wieder einen Block enthalten usw. Den Abschluss bildet auch hier die schließende Klammer.

Der Vollständigkeit halber sei es an dieser Stelle auch erwähnt: Es gibt auch eine leere Anweisung. Die ist grafisch schwer darstellbar, denn sie besteht aus nichts. Ihren häufigsten Einsatz hat sie unabsichtlich, wenn man mal ein Semikolon zu viel setzt. Dank leerer Anweisung ist das dann kein Fehler.

Betrachten wir jetzt unsere bisher geschriebenen Programme noch einmal unter diesen Gesichtspunkten sehen wir, dass alle diese Programme (und auch alle zukünftigen) lediglich aus einem Block und einem Namen (bisher `main`) bestehen.

7.2 Bedingte Anweisung

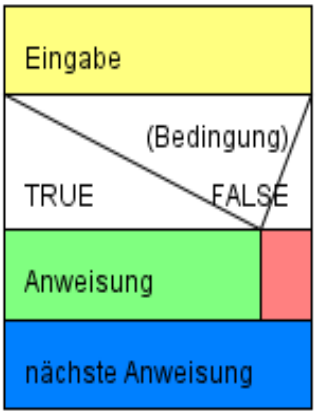
Eines der wichtigsten Konstrukte ist die einfache Entscheidung, von der dann abhängt, ob ein Programmteil ausgeführt wird oder nicht. Zentrales Element ist dabei die Bedingung, deren Ergebnis stets



entweder wahr (`true`) oder unwahr (`false`) ist, dazwischen gibt es nichts. Im Struktogramm sieht das Ganze so aus wie in der Abbildung.

Unser Programmablauf geht ja grundsätzlich von oben nach unten, aber hier haben wir jetzt den Fall, dass es plötzlich zwei parallele Wege nach unten gibt. Der rechte Weg (unter `false`) ist leer, nur der linke Zweig (unter `true`) enthält eine Anweisung. Dieses ist die bedingte Anweisung, die nur dann ausgeführt wird, wenn die Bedingung (in der runden Klammer) erfüllt ist, also wahr (`true`) liefert.

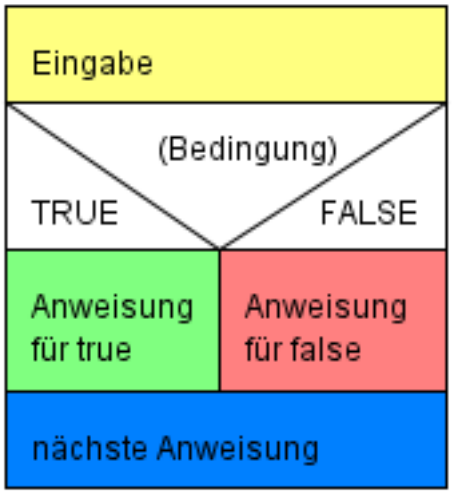
Ist die Bedingung unwahr (also `false`), wird diese Anweisung übersprungen und es wird die nächste Anweisung ausgeführt.



Der Ablauf sieht also so aus: Zuerst wird die Anweisung, die hier als Eingabe bezeichnet wird, ausgeführt. Dann kommt unser neues Konstrukt. Die Bedingung (zwischen den runden Klammern) wird ausgewertet (also geprüft). Nur wenn diese Auswertung zum Ergebnis wahr (`true`) führt, wird die grün hinterlegte Anweisung ausgeführt. Bei unwahr (`false`) passiert nichts (rot hinterlegt). Und danach geht es in jedem Fall mit der (blau hinterlegten) nächsten Anweisung weiter.

Die (grün hinterlegte) Anweisung wird also nur durchgeführt, wenn die Bedingung wahr ist.

7.3 Zweiseitige Auswahl

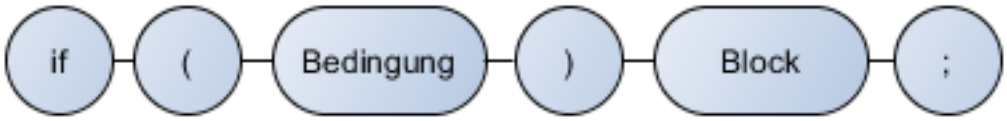


Das Struktogramm oben zeigt ja schon sehr deutlich, dass es im Fall unwahr (`false`) sicher auch möglich sein muss, eine Anweisung abzuarbeiten.

Auch hier wird zuerst die Anweisung, die hier als Eingabe bezeichnet wird (gelb), ausgeführt. Dann kommt unser neues Konstrukt. Die Bedingung (zwischen den runden Klammern) wird ausgewertet (also geprüft). Nur wenn diese Auswertung zum Ergebnis wahr (`true`) führt, wird die grün hinterlegte Anweisung für `true` ausgeführt. Bei unwahr (`false`) wird jetzt die Anweisung für `false` ausgeführt. Man hat also in diesem Fall ein echtes entweder-oder programmiert. Und danach geht es in jedem Fall mit der (blau hinterlegten) nächsten Anweisung weiter.

In C++ basieren beide Varianten auf dem selben Konstrukt. Im ersten Fall bleibt ein Zweig leer, im zweiten Fall ist dieser Zweig gefüllt. Deswegen erfolgt die Beschreibung auch an dieser Stelle und die bedingte Anweisung ist ein Spezialfall der zweiseitigen Auswahl.

Schauen wir uns die bedingte Anweisung noch einmal in einer anderen Darstellung an:



Wir finden mit dem Block einen alten Bekannten wieder, das Semikolon und die runden Klammern kennen wir auch schon. Neu ist das Schlüsselwort `if` und die `Bedingung`. Unter `Bedingung` verstehen wir einen Ausdruck, der nach der Auswertung entweder `true` oder `false` liefert. Klassisch ist das jede

Art von Vergleich (größer, kleiner gleich, ...) aber auch einfache Variablen vom Typ `bool` sowie alle logischen Verknüpfungen sind denkbar.

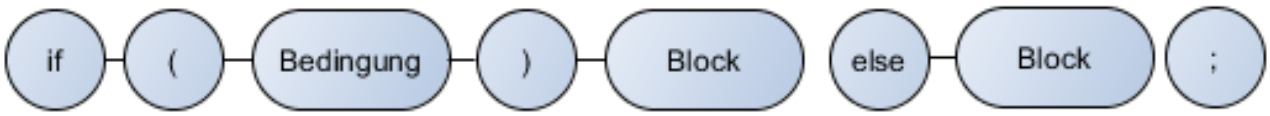
Versuchen wir einmal, daraus etwas in unsere Entwicklungsumgebung zu übernehmen.

```
if (i > maximum)
        {
            maximum = i;
        }
```

In diesem Fragment bildet `(i > maximum)` die Bedingung, die von Klammern eingefasst wird. Wenn der Inhalt der Variablen `i` größer als der Inhalt der Variablen `maximum` ist, ist die Bedingung erfüllt (sie liefert also den Wert `true` zurück). Damit wird die Anweisung im Block `maximum = i;` ausgeführt. Ist die Bedingung nicht erfüllt passiert nichts.

Egal wie der Vergleich ausgegangen ist und welches Ergebnis die Bedingung liefert, danach wird mit der Anweisung weitergemacht, die als nächstes folgt (hier aber nicht mehr aufgeführt ist).

Die zweiseitige Auswahl bietet die Möglichkeit, auch für den Fall, dass die Bedingung nicht erfüllt ist, eine Anweisung (bzw. einen Block von Anweisungen) auszuführen. Grafisch sieht das dann so aus:



Und in C++:

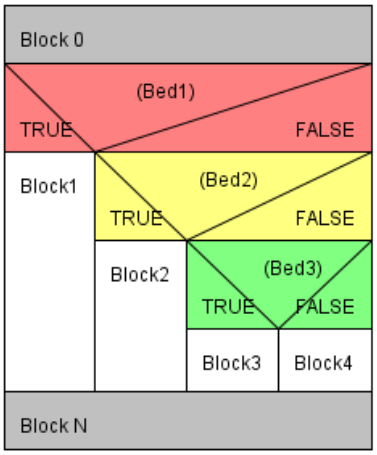
```
if (0 > diskriminante)
        {
            retValue = NEGATIV;
        }
        else
        {
            retValue = POSITIV;
        }
```

Neu dazu gekommen ist das Schlüsselwort `else` zum Einleiten des alternativen Blocks, der ausgeführt werden soll, wenn die Bedingung **nicht** erfüllt ist.

Wenn man diesen alternativen Block einfach leer lässt (was ja nicht verboten ist) hat man den Sonderfall einseitige Auswahl wieder hergestellt. Es gibt Dozenten, die bestehen darauf *„zu jedem if gehört ein else“*. Aus didaktischen Gründen durchaus verständlich, weil es dann bei den geschachtelten Bedingungen keine falschen Zuordnungen von `else` und `if` gibt. Die aktuellen Entwicklungsumgebungen unterstützen aber bei der Programmentwicklung durch Einrückungen und das Hervorheben der zusammengehörigen Klammern derart gut, dass man auf diese Sicherheitsmaßnahme verzichten kann.

Zusammenfassend: Die bedingte Anweisung oder zweiseitige Auswahl entscheidet anhand eines *logischen Ausdrucks* (`true` oder `false`) darüber, ob eine Anweisungsfolge (Block) ausgeführt wird oder nicht bzw. ein alternativer Block. Als Bedingung ist alles erlaubt, was entweder `true` oder `false` zurück liefert. Wenn Sie sich jetzt zurück an den Datentyp `bool` erinnern, wissen Sie auch noch, dass der Wert 0 als `false` betrachtet wird und jeder andere Wert als `true` interpretiert wird. Das führt oft zu syntaktisch korrekten Bedingungen, die aber nicht das widerspiegeln, was gemeint war.

7.4 Geschachtelte Bedingungen



Wenn ein Block in einer Verzweigung (zweiseitigen Auswahl) selbst wieder eine Verzweigung ist, spricht man von einer geschachtelten Bedingung. Die Abbildung links zeigt den prinzipiellen Aufbau. Nach den einleitenden Anweisungen von Block 0 wird die erste Bedingung Bed1 geprüft. Ist die nicht erfüllt (`false`) kommt als weitere Bedingung Bed2 zum Tragen. Ist auch die nicht erfüllt, greift Bed3.

Je nachdem, welche Bedingung davon erfüllt (also `true`) ist, wird der zugehörige (weiße) Block unter TRUE abgearbeitet. Wenn keine der drei Bedingungen erfüllt ist, bleibt nur der Weg über Block 4. Egal welcher Block (1-4) durchlaufen wurde, danach kommt der abschließende Block N. **Es wird aber auf jeden Fall nur einer der Blöcke 1 bis 4 durchlaufen!**

In C++ Notation:

```
{
      BlockN;
      if (Bed1)
      {
         Block1;
      }
      else if (Bed2)
      {
         Block2;
      }
      else if (Bed3)
      {
         Block3;
      }
      else
      {
         Block4;
      }
      BlockN;
}
```

```
{
      BlockN;
      if (Bed1)
      {
         Block1;
      }
      else if (Bed2)
            {
               Block2;
            }
            else if (Bed3)
                  {
                    Block3;
                  }
                  else
                  {
                     Block4;
                  }
      BlockN;
}
```

Beide Programmteile sind identisch und unterscheiden sich nur in der optischen Darstellung und der automatische Einrückung durch die verwendete Entwicklungsumgebung.

Die geschachtelte Bedingung ist dann besonders gut geeignet, wenn es sich bei den Bedingungen um Intervalle oder Bereiche handelt. So könnte z.B. eine Rabattstaffelung berechnet werden:

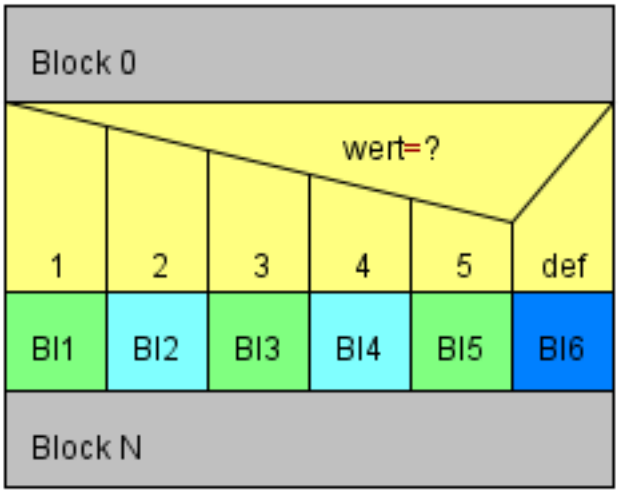
```
{
      if (waren_wert > 10000)
            rabatt_in_prozent = 20;
      else if (waren_wert > 7500)
            rabatt_in_prozent = 15;
      else if (waren_wert > 5000)
            rabatt_in_prozent = 10;
      else if (waren_wert > 2500)
            rabatt_in_prozent = 5;
      else
            rabatt_in_prozent = 0;
}
```

Die Reihenfolge der Bedingungen ist hier sehr wichtig, da ja die erste erfüllte Bedingung abgearbeitet wird und die restlichen Bedingungen nicht mehr geprüft werden.

7.5 Mehrseitige Auswahl

Es gibt Fälle, da hilft eine geschachtelte Bedingung selbst mit sehr hoher Schachtelungstiefe nicht oder nur schwer weiter. Von der damit einhergehenden Unübersichtlichkeit mal ganz zu schweigen. Dafür wurde die mehrseitige Auswahl erfunden.

Hier hängt die Auswahl und damit der weitere Ablauf eines Programms nicht an einer *logischen* Bedingung sondern an einem *numerischen* Wert. Die einzelnen Programmzweige werden quasi durchnummeriert und der Inhalt einer (numerischen) Variable entscheidet, welcher Zweig genommen wird.



Numerisch in diesem Zusammenhang bedeute dass der Inhalt der Variable zu einem abzählbaren Datentyp gehören muss, Fließkommazahlen sind damit nicht dabei. Aber die Datentypen `char`, `int`, `byte` und alle selbst definierten `enum`-Typen gehören dazu. Mit anderen Worten: alles, was im Rechner exakt ohne Rundungsfehler dargestellt werden kann und einen genau definierten Vorgänger und/oder Nachfolger hat.

Nach dem einleitenden Block 0 wird einmalig der Inhalt der Variablen in der Bedingung (hier heißt sie wert) geprüft und bestimmt. In Abhängigkeit dieses Inhaltes wird dann der Zweig durchlaufen, dem dieser

Wert als Konstante zugeordnet ist. Hat also die Variable den Inhalt 3 wird Block Bl3 durchlaufen, hat sie den Wert 5 wird stattdessen Block Bl5 abgearbeitet.

Eine Besonderheit gibt es mit dem Zweig `default`. Dieser Zweig ist optional (kann also entfallen) und kommt dann zum Zuge, wenn keiner der anderen Fälle zutrifft.

Es ist nicht vorgeschrieben, dass die Werte fortlaufend (lückenlos) oder in aufsteigender Reihenfolge angegeben sein müssen, es ist auch erlaubt, mehrere Werte für einen Fall zu definieren. Umgekehrt darf es aber keinen Wert doppelt geben, das findet der Compiler nicht so gut.

Egal, welcher Zweig durchlaufen wurde, es geht danach mit Block N weiter.

Anwendung findet dieses Konstrukt häufig bei Menü-Auswahlen oder bei der Verarbeitung von Zuständen. Auf den Syntaxgraphen verzichte ich an dieser Stelle, er wird wegen der vielfältigen Möglichkeiten einfach zu komplex. Wir betrachten den Aufbau einfach anhand eines Beispiels:

```
{
      Block_0;
      switch (wert)
      {
      case 1: Block1;
            break;
      case 2: Block2;
            break;
      case 3: Block3;
            break;
      case 4: Block4;
            break;
      default: Block6;
      }
      Block_N;
}
```

Upps, das sieht ja doch etwas anders aus, wie das Struktogramm vermuten lässt und bedarf einer näheren Betrachtung.

7.5.1 switch

Das Schlüsselwort `switch` leitet die mehrseitige Auswahl ein. In (runden) Klammern wird dahinter der Ausdruck angegeben, der zur Entscheidung herangezogen werden soll. Dieser Ausdruck muss einen ganzzahligen Wert liefern, im einfachsten Falle ist das der Inhalt einer Variablen, es kann aber auch eine beliebige Berechnung sein.

Daran schließt sich zwingend ein Block in einem bestimmten Format an, die geschweiften Klammern müssen also stehen. Dieser Block ist so aufgebaut:

Für jeden Fall (Verzweigung) steht das Schlüsselwort `case`, dahinter eine Konstante (also im einfachsten Fall eine Zahl, Buchstabe oder ein `enum`-Wert), gefolgt von einem Doppelpunkt. Hinter dem Doppelpunkt kommt ein normaler Block (ohne geschweifte Klammern!)[13], der abgearbeitet wird, wenn der Wert des Ausdrucks dem Wert der Konstanten vor dem Doppelpunkt entspricht. Den Abschluss bildet das (optionale) Schlüsselwort `break`, dann folgt der nächste Fall.

So einfach liegt der Fall aber leider nicht immer. Möchte man den selben Anweisungsblock bei mehreren Werten auswählen, muss man diese einzelnen Fälle auch einzeln aufführen, und zwar hintereinander. Das kommt sehr häufig bei Menüs vor, die zur Auswahl Buchstaben verwenden. Da ist es ja durchaus sinnvoll, dass bei Verwendung eines Großbuchstabens dieselbe Aktion startet, die auch beim Kleinbuchstaben erfolgen würde. Das löst man dann so:

```
switch (wert)
            {
            case 'a':
            case 'A': Block_a;
                  break;
            case 'x':
            case 'X': Block_X;
                  break;
            default: Block6;
            }
```

Jetzt wird bei 'a' und 'A' sowie 'x' und 'X' jeweils die gleiche Anweisungsfolge ausgeführt.

Die Bedeutung von `break` lässt sich hier bereits erahnen, es stoppt die weitere Ausführung. Und damit ist auch das hier möglich:

```
switch (wert)
{
case rot: Block_rot;
case gelb: Block_gelb;
      break;
case gruen:
case blau: Block_blaugruen;
      break;
default: Block_weiss;
}
```

Es werden alle Anweisungen ausgeführt, die nach der berechneten Konstanten stehen bis zum Auftreten des ersten `break`. Falls die Variable `wert` den Inhalt `rot` hat, wird erst `Block_rot` und dann auch noch `Block_gelb` ausgeführt, dann greift das `break`.

Hat die Variable `wert` den Inhalt `gelb`, wird nur der `Block_gelb` ausgeführt (der Einsprung erfolgt ja nach dem `Block_rot`)..

Stehen `blau` oder `gruen` in der Variablen wert, wird nur der `Block_blaugruen` ausgeführt, bei allen anderen `Block_weiss`.

13 Natürlich sind auch hier geschweifte Klammern erlaubt, weil sie überall um Anweisungen herum eingesetzt werden dürfen, um einen echten Block anzulegen. Hinter einer `case`-Anweisung sollte man auf die geschweiften Klammern aber verzichten, weil dies häufig zu unerwünschten Seiteneffekten führt

Der Begriff „Konstante“ im Zusammenhang mit der `switch`-Anweisung bedeutet einen festen, unveränderlichen Wert als Kriterium für jeden einzelnen Auswahlzweig. Das soll den Unterschied zu einer Variablen verdeutlichen, die ja hier gerade nicht verwendet werden darf. Die variable Größe steht hinter dem `switch`, hinter den einzelnen `case` dürfen nur feste (=konstante) Werte stehen, die die Variable oben annehmen kann

7.5.2 default

Der `default`-Zweig ist der Rettungsanker, wenn kein anderer Fall greift, er ist nicht zwingend vorgeschrieben und darf daher auch entfallen. Wenn er da ist, dient er im wesentlichen zur Fehlerbehandlung, denn dieser Fall wird ja immer dann aufgerufen, wenn keiner der erwünschten und vorgesehenen Fälle greift. Es ist also ein Fall eingetreten, den der Programmierer so nicht vorgesehen hat oder der schlicht nicht vorkommen sollte. Bei einem Menü ist das relativ leicht vorstellbar, wenn der Benutzer einen Menüpunkt eingibt, der nicht in der Liste vorkommt und für den es daher auch kein Programm gibt. Dann erhält er hier eine passende Fehlermeldung.

Denkbar sind aber auch Programme, in denen der `default`-Fall der Normalfall ist und die anderen Fälle eine andere Behandlung erfordern. Das ist z.B. bei der Umwandlung von Kleinbuchstaben in Großbuchstaben der Fall (der ASCII-Wert ist um 32 kleiner), außer bei den deutschen Umlauten, die weichen teilweise von dieser Regel ab.

Die Programmanweisungen zwischen `case:` und `break;` sehen aus wie ein Block, wurden wie ein Block erläutert und funktionieren wie ein Block. Mit einem klitzekleinen Unterschied: es gibt keine lokalen Variablen!

Werden in diesem Bereich Variablen deklariert, die nur in diesem Zweig verwendet werden, sind das trotzdem globale Variablen. Und daran stört sich der Compiler, weil er dann das Problem hat, dass diese Variablen nur in manchen Fällen definiert werden und in den anderen Zweigen nicht. Und damit kann er verständlicherweise nicht umgehen, er wirft also eine Fehlermeldung aus.

Abhilfe schafft es, diesen Bereich dann zusätzlich in geschweifte Klammern zu setzen: `case:` { Anweisungen} `break;` Jetzt ist die Variable eine echte lokale Variable und der Compiler ist zufrieden. Ihr Betreuer im Praktikum findet diese Lösung eventuell nicht so gut....(weil er Ihnen aus guten Grund beigebracht hat, alle benötigten Variablen an einem speziellen Ort zu Beginn der Funktion zu definieren)

7.6 Übungen

Währungsrechner

Ziel ist ein Programm, dass die Währungen Dollar, Yen und Pfund in Euro umrechnet. Die Ausgangswährung ist per Menü auszuwählen (den Umrechnungskurs finden Sie im Internet, sie können aber Ihrer Phantasie auch freien Lauf lassen).

Schreiben Sie eine Version, die mit bedingten Verzweigungen (if) arbeitet und eine zweite, die mit switch funktioniert.

Erweitern Sie das Programm um weitere Währungen (Rubel, Zloty, Rand usw.). Welche Version (if oder switch) ist besser lesbar und leichter verständlich?

(nach Bjarne Stroustrup)

Steuerberechnung

Schreiben Sie ein Programm, das aufgrund des Bruttoeinkommens die Steuerlast berechnet. Der Steuersatz bestimmt sich nach dieser (frei erfundenen) Tabelle:

unter 17.000 €	steuerfrei
bis 34.000 €	12%
bis 48.000 €	16%
bis 56.000 €	22%
über 92.000 €	30%

Zweitgrößte Zahl

Schreiben Sie ein Programm, dass von fünf eingegebenen Gleitkommazahlen die zweitgrößte ermittelt. Die fünf Zahlen werden nach und nach eingegeben (Die Schleife kennen wir noch nicht, sie darf also an dieser Stelle nicht verwendet werden). Überlegen Sie im Vorfeld, welche Vergleiche notwendig sind!

Kleinbuchstaben umwandeln

Schreiben Sie ein Programm, das einen eingegebenen Kleinbuchstaben in den korrespondierenden Großbuchstaben verwandelt. Tipp: der Datentyp `char` kann problemlos und ohne Komplikationen in den Datentyp `int` (und zurück) konvertiert werden. Man kann also direkt mit den ASCII-Werten arithmetische Rechnungen durchführen. Eine ASCII-Tabelle finden Sie im Anhang oder im Internet.

7.7 Wiederholungen/Schleifen

Alle Programme, die wir bisher geschrieben haben, holten sich nach dem Start eine oder mehrere Eingaben, rechneten daraus etwas aus und waren dann zu Ende. Dabei wäre es doch sehr praktisch, wenn der Währungsrechner nicht nach jeder Umrechnung neu gestartet werden müsste, sondern wir einfach den nächsten Betrag umrechnen könnten.

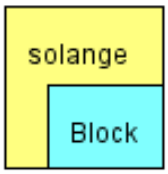
Natürlich gibt es auch dazu das passende Konstrukt, die Wiederholungsanweisung oder umgangssprachlich die Schleife.

Eine Schleife ist nichts anderes, als eine Bedingung, die darüber entscheidet, wie oft ein Block durchlaufen wird. In der Bedingung muss irgendetwas überprüft werden, das sich im Schleifenblock ändert oder zumindest ändern kann, sonst hat man eine Endlosschleife (was unter gewissen Umständen durchaus erwünscht sein kann). Die Bedingung ist eng mit der bedingten Anweisung verknüpft und liefert einen Wert vom Typ `bool` (`true` oder `false`) zurück.

Die verschiedenen Schleifen unterscheiden sich nur in der Position und Art der verwendeten Bedingung und können mit etwas Übung leicht ineinander überführt oder umgewandelt werden.

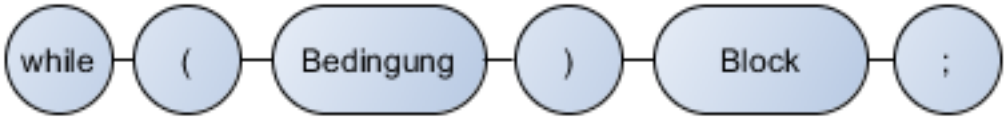
7.8 kopfgesteuerte Schleifen

Die kopfgesteuerte Schleife trägt ihren Namen, weil sie zuerst die Bedingung prüft und dann entscheidet, ob der Schleifenkörper durchgeführt wird und wenn ja, wie oft. Es kann hier also passieren, dass der Block überhaupt nicht ausgeführt wird, deswegen wird dieser Schleifentyp auch gerne abweisende Schleife genannt.



7.8.1 while

Eine `while`-Schleife ähnelt in ihrer Funktionalität als auch ihrem Aussehen nach stark der `if`-Bedingung. Und in der Tat unterscheiden sich die Syntaxgraphen nur im verwendeten Schlüsselwort:



Das englische Wort *while* bedeutet im Deutschen *solange*, und damit ist auch die Wirkung dieses Konstrukts klar: solange die Bedingung erfüllt ist, wird der Anweisungsblock abgearbeitet.

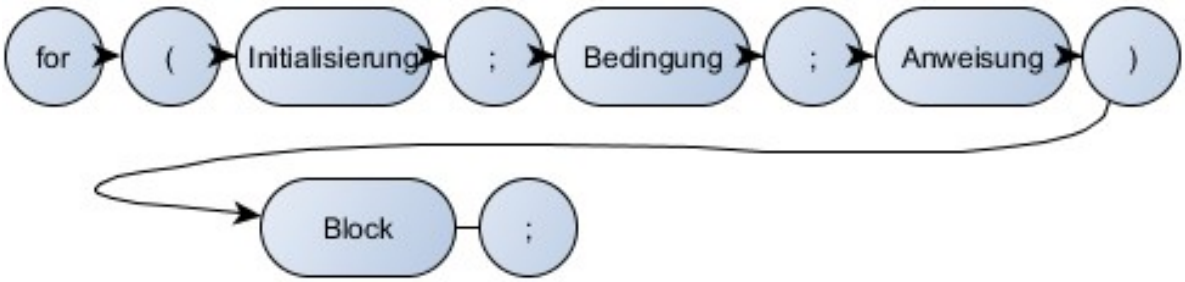
Nach dem Block wird die Bedingung erneut geprüft und neu entschieden. Es kann also sein,. dass der Block **niemals** durchlaufen wird, weil die Bedingung von Anfang an nicht erfüllt ist. Daher der Begriff *abweisende Schleife*.

Um aus der Schleife irgendwann einmal heraus zu kommen, muss es also im Anweisungsblock möglich sein, etwas so zu ändern, dass die Bedingung nicht mehr erfüllt, also `false`, wird. Wird das vergessen, hat man eine Endlosschleife.

7.8.2 for

Die `for`-Schleife ist ein Spezialfall der kopfgesteuerten Schleife und wird immer dann eingesetzt ist, wenn von Anfang an klar ist, wie oft die Schleife durchlaufen werden soll. Hier gibt es nicht nur eine Bedingung, die zum Abbruch führt, sondern auch gleich dazu die Anweisung, die diese Bedingung ändert. Und zusätzlich eine Initialisierung, damit auch klar ist, mit welchen Werten die Schleife startet.

Im Syntaxgraphen sieht das so aus:



Die `for`-Schleife ist eine Zählschleife, die im wesentlichen eine Variable (diese wird häufig *Laufvariable* genannt) nach gewissen Regeln ändert. Welche Variable das ist, gibt die Initialisierung an. Hier wird die Variable mit einem Startwert versehen (und in den meisten Fällen wird sie an dieser Stelle auch gleich noch angelegt, dann hat man nämlich eine lokale Variable, die nur innerhalb der Schleife existiert und ändert nicht aus Versehen eine Variable, die an anderer Stelle mit anderen Aufgaben verwendet wird). Man findet hier also Anweisungen, die genau wie eine Variablendeklaration aussehen: `i=0;` (für eine

bereits existierende Variable) oder auch `int i=0;` (dann wird diese Variable i lokal angelegt). **Achtung**: es muss sich hier um einen Datentyp handeln, der gut gezählt werden kann, also alles was ganzzahlig oder anderweitig abzählbar ist (z.B. alle `enum`-Typen), keine Fließkommazahlen! Als Startwert ist alles erlaubt, was in diesem Datentyp möglich ist.

Hinter dem Semikolon folgt analog zur `while`-Schleife die `Bedingung`, die kontrolliert, ob der Block ausgeführt wird oder nicht. Wird nur eine reine Zählschleife benötigt, steht hier oft etwas wie `i<10;` oder `i*3<22;` also eine Bedingung, die in irgendeiner Form die eben gesetzte Variable enthält. Das muss aber nicht so sein, die Bedingung kann sich auch auf etwas ganz anders beziehen! (und dann gilt genau das, was auch bei der `while`-Schleife steht: wenn sich im Schleifenkörper nichts findet, was die Bedingung irgendwann einmal zum Ändern bringt, hat man eine Endlosschleife.)

Die letzte `Anweisung` im Schleifenkopf schließlich ist dafür gedacht, die verwendete Laufvariable zu ändern. Je nach Absicht findet man hier also eine Anweisung, die die Laufvariable erhöht oder erniedrigt, z.B. `i++;` oder `i--;` wenn in Einerschritten rauf oder runter gezählt werden soll. Natürlich ist auch jede andere Anweisung denkbar, die andere Schrittweiten bestimmt, also etwa `i=i+3;` oder `i-=2;`

Schauen wir uns das noch einmal an einem Programmstück an:

```
for (int i = 1; i <= 10; i++)
{
      cout << "i= " << i << "  " << i*i << endl;
}
```

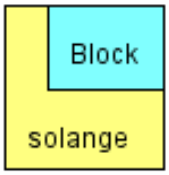
Im Kopf der Schleife wird die Variable i angelegt und mit 1 initialisiert. Dann wird die Bedingung i<=10 geprüft und für `true` befunden (i ist ja gerade mal auf 1 gesetzt worden). Also wird der Anweisungsblock ausgeführt und i als auch i*i (also i^2) ausgegeben.

Danach kommt die Anweisung i++ zum Zuge, also wird i um 1 erhöht und hat jetzt den Wert 2. Und jetzt beginnt die Schleife wieder von vorne, die Bedingung wird geprüft, für `true` befunden, i und i^2 werden ausgegeben.

Das alles wiederholt sich bis i den Wert 11 hat. Jetzt ist die Bedingung nicht mehr erfüllt, also wird der Block auch nicht ausgeführt. Die letzte ausgegebene Zeile ist also mit dem Wert 10 in i geschrieben worden. Wenn Sie das Programm selbst schreiben, sollten sie eine Liste der Quadratzahlen von 1 bis 10 auf ihrem Monitor sehen.

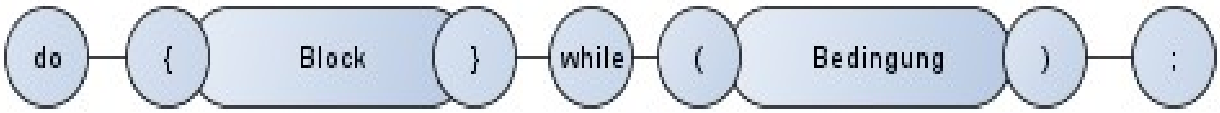
Die `for`-Schleife wird sehr häufig in Verbindung mit Feldern (array), Aufzählungen (`enum`) und anderen, Reihen-ähnlichen Daten(-strukturen) verwendet, wenn eine bestimmte Operation auf alle oder einige, aber leicht zu bestimmende, Daten ausgeübt werden soll. Bei der Vorstellung dieser Datentypen wird sie uns daher wieder begegnen.

7.9 Fußgesteuerte Schleife (do...while)



Es ist offensichtlich, dass es neben einer kopfgesteuerten auch eine fußgesteuerte Schleife geben muss, die dann auch annehmende Schleife genannt wird. Warum das so ist, sieht man schon am Struktogramm, der Block liegt **vor** der Bedingung.

Eingeleitet wird die fußgesteuerte Schleife durch das Schlüsselwort `do` am Kopf der Schleife. Dann kommt der Block und danach die Bedingung. Der Block wird also mindestens einmal durchlaufen, egal was die Bedingung danach ergibt.



Alles, was bei der `while`-Schleife gesagt wurde, gilt auch hier: innerhalb des Blocks muss es eine Anweisung geben, aufgrund der die Bedingung irgendwann einmal `false` wird. Hier lauert eine Falle durch die Nachlässigkeit der deutschen Sprache, die `do`..`while`-Schleife läuft solange wie die Bedingung erfüllt also `true`) ist. Im Sprachgebrauch hat sich aber der Begriff *solange bis* eingenistet, man muss also bei der Formulierung der Bedingung aufpassen.

Ein häufiger Anwendungsfall für diese Schleife ist die Überprüfung von Eingaben, wenn diese bestimmten Kriterien genügen sollen. Dazu wird die Eingabe (und idealerweise auch die Eingabeaufforderung) solange wiederholt, wie die Eingabe falsch erfolgte (und nicht: solange bis die Eingabe richtig ist). Als Programm sähe das dann so aus:

```
do
{
      cout << "Bitte eine Zahl zwischen 1 und 100 eingeben: ";
      cin >> zahl;
} while (zahl<1 || zahl >100);
```

Es reicht also aus, wenn mindestens eine der beiden Oder-verknüpften Teilbedingungen erfüllt ist, damit die Schleife wiederholt wird. Sie wird nur beendet, wenn beide Bedingungen `false` sind. Alternativ wäre auch diese Formulierung denkbar: `while (!(zahl>=1 && zahl <=100));`

Man beachte beim Wechsel von Oder- auf Und-Verknüpfung auf den Wechsel bei den Bedingungen, hier greifen die Gesetze von De Morgan, Näheres erfahren Sie bei Interesse unter https://de.wikipedia.org/wiki/De_Morgansche_Gesetze

7.10 Welche Schleife für welchen Zweck?

Man kann zeigen, dass es möglich ist, alle Programmieraufgaben mit nur einer Schleifenart zu lösen. Daher ist es auch möglich, eine beliebige Schleifenart durch eine andere zu ersetzen, ohne dass die Funktionalität des Programms sich dadurch ändert. Aber nur, weil etwas theoretisch machbar ist, muss man das noch lange nicht auch machen. Jede Schleifenart hat ihre individuellen Vor- und Nachteile und damit einen speziellen Einsatzzweck, für den sie besonders gut geeignet ist.

Die for-Schleife spielt ihre Vorteile aus, wenn es darum geht, eine zu Beginn der Schleife bereits bekannte Anzahl von Durchläufen zu absolvieren. Das ist in der Regel der Fall beim Durchlaufen von Feldern (Arrays, Reihentyp), deren Länge (=Anzahl der Elemente) bekannt ist. Dann hat man nämlich alles, was der Kontrolle der Schleife dient, in einer Zeile kompakt zusammen.

Die while-Schleife kommt häufig dann zum Einsatz, wenn eine unbekannte Anzahl von Durchläufen zu erwarten ist. Ein idealer Anwendungsfall ist das Lesen von Daten aus einer Textdatei, hier ist irgendwann Schluss mit Daten, und genau dann ist auch Schluss mit der Schleife. Das Ende der Schleife wird also oft durch ein Ereignis bestimmt, dessen Auftreten vorher nicht genau bekannt ist. Analog verhält sich dies beim Abarbeiten von dynamischen Datenstrukturen, auch bei denen ist plötzlich die Liste zu Ende.

Die do...while-Schleife schließlich ist die Idealbesetzung, wenn der Schleifenkörper mindestens einmal durchlaufen werden soll. Das ist regelmäßig bei Eingaben der Fall: der Benutzer des Programms soll eine Eingabe tätigen, danach wird überprüft, ob diese die gestellten Bedingungen erfüllt. Ist dem so, geht es im Programmablauf einfach weiter, wenn nicht, wird einfach die gesamte Eingabe solange wiederholt, wie die Eingabe nicht den Forderungen entspricht. Es ist übriges keine gute Idee, nach einer falschen

Eingabe die Eingabe erneut zu verlangen, diesmal aber auf die Überprüfung zu verzichten und einfach im Programm weiter zu machen (wird aber oft von Anfängern so gemacht).

7.11 Sprünge

Sprünge sind mit der strukturierten Programmierung ziemlich aus der Mode gekommen, ein Struktogramm bietet an sich zunächst auch kein Element für einen Sprung. Die Sprache C kannte Sprünge aber, genauso wie viele alte (wirklich alt, also aus den 60er und 70er Jahren) Programmiersprachen. Dabei handelt es sich um Relikte aus der maschinennahen Programmierung und aus der Zeit vor der Einführung von echten Schleifen.

Einen Sonderfall des Sprungbefehls haben wir bei der `switch`-Anweisung kennengelernt, das `break`; Damit wird die Verarbeitung einer Anweisungsfolge sofort abgebrochen und die Ausführung hinter der `switch`-Anweisung fortgesetzt. Dieser Sprung taucht im Struktogramm nicht auf, weil es sich ja um die mehrseitigen Auswahl handelt[14]. In anderen Programmiersprachen, für die solche Struktogramme ja unverändert auch gelten, besteht auch keine Notwendigkeit für einen Sprung, da gibt es dann aber die Möglichkeit, eine Anweisungsfolge direkt mit mehreren Konstanten zu versehen. Genau genommen handelt es sich hier also nicht um einen Sprung, sondern um die konkrete Umsetzung durch die Sprache C++. Hätte man ein anderes Schlüsselwort definiert (z.B. `endcase`), käme niemand auf die Idee, hier einen Sprung anzunehmen.

Mit `break` sind aber auch Anweisungen mit echten Sprüngen möglich:

```
fehler = 0;
while (a != b)
{
      // …
      b = 4;
      if (fehler == desaster)
      {
            break;
      }
      a++;
      //…
}
```

Diese Konstruktion ist sehr angenehm, im Falle eine Desasters kann man direkt an das Ende der Schleife springen. Das bedeutet aber auch, dass die Variable a nicht in jedem Fall hochgezählt wird, auch wenn dies aufgrund der Einrückung so aussieht.

Ein `break`; in dieser Bedeutung zeugt fast immer von schlechtem Programmierstil, es lässt sich nämlich einfach vermeiden:

```
fehler = 0;
while (a != b && fehler!= desaster)
{
      // .…
      b = 4;
      a++;
```

14 Eine `switch`-Anweisung ist ähnlich aufgebaut wie ein Bahnhof mit mehreren Bahnsteigen: Der Zug kommt auf dem einzigen Einfahrtsgleis an, wird über mehrere Weichen (=`switch`) auf ein bestimmtes Gleis (=`case`) geführt und verlässt den Bahnhof über das gemeinsame Ausfahrtsgleis

```
        //...
    }
```

Diese Schreibweise ist nicht nur kürzer, sie ist auch viel klarer und leichter zu verstehen. Wenn es trotzdem einmal unvermeidlich sein sollte ein `break` zum Verlassen einer Schleife einzusetzen, muss diese durch eine entsprechende Kommentierung ganz klar erläutert und dokumentiert werden (ich bitte um eine E-Mail, wenn Ihnen ein solcher Fall unterkommt, mir fällt nämlich keiner ein).

Ähnlich gelagert ist der Fall bei `continue`, hier wird die Schleife nicht beendet, sondern nur der aktuelle Durchlauf. Der Teil innerhalb des Anweisungsblockes nach dem `continue`; wird also übersprungen. Dabei kann man diesen Sprung noch leichter vermeiden als bei `break`.

```
Fehler = 0;
while (a != b )
{
        // ....
        b = 4;
        if (fehler == desaster)
        {
                continue;
        }
        a++;
        //...
}
```

```
fehler = 0;
while (a != b )
{
        // ....
        b = 4;
        if (fehler != desaster)
        {
                a++;
        }
        //...
}
```

Es genügt, die Bedingung zu negieren („umzudrehen“) und die zu überspringende Anweisung an diese Bedingung zu knüpfen. Damit sollte diese Anweisung in ihrem studentischen Programmierübungen niemals auftauchen.

Nur der Vollständigkeit halber: es gibt auch noch den Befehl `goto`, der aber in einem Programm nach heutigen Maßstäben schlicht nichts verloren hat und deshalb hier nicht weiter erwähnt wird.

7.12 Übungen

Summieren

Schreiben Sie ein Programm, das die Summe der ersten 100 (1000, 10.000) Zahlen berechnet und ausgibt. Es gibt neben der Lösung mit einer Schleife auch eine analytische Variante (Hinweis: addieren Sie die erste und die letzte Zahl, addieren Sie die zweite und die vorletzte Zahl, usw. Was stellen Sie fest?).

Stellen Sie beide Programme gegenüber und versuchen Sie, die Laufzeit zu ermitteln. Merken Sie einen Unterschied? (Tipp: eine „Stoppuhr für Programme“ finden Sie im Kapitel 15)

Maximum/Minimum

Schreiben Sie ein Programm, das so lange ganze Zahlen von der Tastatur einliest, bis die Zahl 0 eingegeben wird. Danach soll die größte (und/oder kleinste) eingegebene Zahl angezeigt werden

Erweitern Sie das Programm so, dass neben der größten (kleinsten) auch die zweitgrößte (zweitkleinste) Zahl ausgegeben wird.

Primzahlen

Der Klassiker schlechthin für Schleifen. Schreiben Sie ein Programm, dass alle Primzahlen in einem Intervall 0 bis *MaxPrim* ausgibt.

Zahlendrehen

Schreiben Sie ein Programm, das die Ziffernfolge einer ganzen Zahl rückwärts ausgibt. (Tipp: der Modulo-Operator % hilft weiter). Und wenn Sie die Zahl schon in ihre Ziffern zerlegt haben, berechnen Sie doch auch gleich noch die Quersumme der Zah!)

Hagelkorn-Folge

Im Buch „Gödel. Escher, Bach" beschreibt Douglas Hofstaedter diese Regeln für eine Zahlenfolge aus einer positiven, ganzen Zahl:

- Ist die Zahl = 1 ist das Ende der Folge erreicht, das Programm endet
- Ist die Zahl gerade, wird sie durch 2 geteilt
- Ist die Zahl ungerade, wird sie mit drei multipliziert und 1 addiert

Schreiben Sie ein Programm, das diese Zahlenfolge zu einer eingegebenen Zahl ausgibt.

Berechnung von π

Leibniz hat für die Berechnung der Kreiszahl π schon vor über 300 Jahren diese Reihe postuliert:

$$\pi = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} \pm \cdots$$

Schreiben Sie ein Programm, das π durch die ersten 10.000 Glieder dieser Reihe bestimmt. Vergleichen Sie das Ergebnis mit dem Ergebnis von $4 * atan(1)$ (dazu muss die Bibliothek `<cmath>` in das Programm eingebunden werden).

8 Strukturierte Programmierung

Wir haben jetzt alle Elemente von C++ kennengelernt, die zum Programmieren auch größerer Probleme ausreichen. Allerdings werden unsere Programme schnell sehr lang und unübersichtlich. Insbesondere haben wir noch keine Möglichkeit vorgestellt, identische Programmteile, die an verschiedenen Stellen benötigt werden, nur einmal schreiben zu müssen, statt mehrmals (das macht das Programm ja nur länger, aber nicht besser).

Aber C++ wäre keine moderne Programmiersprache, wenn es nicht auch dafür eine Lösung gäbe. Sie nennt sich Funktionen.

8.1 Funktionen

Eine Funktion ist nichts weiter als ein in sich abgeschlossenes Stück Programm, das unter einem eigenen Namen auftritt. Der geschickte Einsatz dieser Strukturelemente ist die wahre Kunst des Programmierens.

Funktionen kennt man aus der Mathematik und in C++ ist es ganz ähnlich. *a=sin (alpha)* berechnet den Wert der Winkelfunktion *Sinus* für den Winkel *alpha*. Wie das konkret gemacht wird, ist dabei für den Anwender dieser Funktion völlig belanglos, er verlässt sich auf die korrekte Berechnung durch die Funktion.

Funktionen werden auch manchmal Unterprogramme genannt, weil sie sich von allen Stellen in einem Programm aufrufen lassen, ihre Arbeit erledigen und danach die Programmausführung an der Stelle fortsetzen, an der die Funktion aufgerufen wurde.

Eine Funktion sollte so gebaut werden, dass sie eine konkrete Aufgabe erledigt, nicht mehr, aber auch nicht weniger. So ist eine Mischung aus Berechnung (z.B. einer mathematischen Funktion) und der Ausgabe dieses Wertes durch die Funktion nicht so geschickt, weil der Programmierer mit dem Ergebnis weiter rechnen möchte und die Ausgabe sein mühsam gebasteltes Bildschirmlayout kaputt macht. Außerdem erhöht es die Wiederverwendbarkeit dieser Funktion, wenn sie nur das tut, was ihr Name aussagt.

Eine Funktion liefert typischerweise einen Rückgabewert, weiter oben war das der Sinuswert. Die `return`-Anweisung innerhalb der Funktion sorgt dafür. Alle unsere Programme bisher enthielten genau eine Funktion, die `main()`-Funktion. Und auch die gibt über eine `return`-Anweisung, üblicherweise am Ende, einen Wert zurück, im Normalfall ein `int`. Diesen Wert übernimmt der Aufrufer (bei der main()-Funktion ist das das umgebende Betriebssystem) und wertet ihn aus. Bei `0` war alles ok, jede andere Zahl wird als Fehler interpretiert (Welcher Wert welchem Fehler entspricht muss dokumentiert werden, hier gibt es keine Regelung.).

Werden Funktionen aus unseren Programmen aufgerufen, kann der Rückgabewert einer Variablen zugewiesen werden. Der Funktionsaufruf darf aber auch überall dort stehen, wo ein Ausdruck erlaubt ist, also auch direkt in Berechnungen, in Bedingungen und ähnlichen Stellen.

Formal sieht eine Funktion so aus:

Es beginnt mit dem Rückgabetyp der Funktion, dann folgt der Name der Funktion, eine öffnende runde Klammer, eine Liste von Parametern (kann leer sein), die schließende runde Klammer und dann der Block, der die Anweisungen für diese Funktion enthält.

Der Rückgabewert darf jeden Datentyp haben, nur ein Array ist verboten. Hat die Funktion keinen Rückgabewert steht hier das Schlüsselwort `void`. (Dann gibt es auch kein `return` in der Funktion.)

Für den Funktionsnamen gelten die gleichen Regeln wie für alle anderen Namen auch. Im Zweifelsfall den längeren Namen wählen, der aber mehr aussagt. Es hat sich eingebürgert, Funktionen mit Verben zu benennen, weil sie ja eine konkrete Aufgabe erfüllen. Bei Funktionen mit dem Rückgabewert `bool` fangen die Namen oft mit *ist* (im englischen dann mit *is*) an, weil sich das in logischen Ausdrücken sehr verständlich formulieren lässt.

Eine Funktion kann zwischen Null und nahezu beliebig vielen, durch Komma getrennte, Parameter haben. Ein Parameter hat eine sehr große Ähnlichkeit mit einer Variablendeklaration und erfüllt auch einen sehr ähnlichen Zweck. Alle hier deklarierten Parameter stehen innerhalb der Funktion (also im Block) als lokale Variable zur Verfügung. Beim Aufruf der Funktion werden sie mit den aktuellen Werten aus der Parameterliste des Aufrufers initialisiert.

Aufgerufen wird die Funktion durch ihren Namen und ein Klammerpaar, in dem die notwendigen Parameter stehen. Dabei müssen Typ, Anzahl und Reihenfolge beim Aufrufer und bei der Funktionsdeklaration übereinstimmen (nicht die Namen!!!).

Die folgende Funktion hat weder einen Rückgabewert noch Parameter und kann damit also als reine Maßnahme zur Erhöhung der Übersichtlichkeit missverstanden werden:

```
void sterne() {
      cout << "**********************" << endl;
}
```

Dies Funktion gibt also bei jedem Aufruf eine Reihe Sternchen aus [15] Angewendet als vollständiges Programm sieht das dann etwa so aus:

```
void sterne() {
      cout << "**********************" << endl;
}
int main() {
      sterne();
      cout << "Weihnachtsgeldberechnung:" << endl;
      sterne();
}
```

Wichtig: Die Definition der Funktion muss vor ihrer ersten Verwendung erfolgen, sonst hagelt es Fehlermeldungen. Der Compiler kann ja nicht wissen, ob noch eine passende Funktion kommt oder nicht, da macht er es sich einfach und lässt den Programmierer streng nach Vorschrift arbeiten. Wie man dies zu Gunsten der Übersichtlichkeit ändert, sehen wir später noch.

Auch wenn es bei diesem kleinen Beispiel noch nicht so klar sichtbar ist: Funktionen lohnen sich eigentlich immer, selbst dann, wenn sie nur einmal aufgerufen werden. Sie bringen ganz automatisch deutlich mehr Übersicht ins Programm und sollten daher möglichst oft eingesetzt werden.

15 Diese Zeilen entstanden am 21.12. und sind daher weihnachtlich motiviert

Tipp: Insbesondere bei komplexen Aufgabenstellungen lohnt es sich, die einzelnen Verfeinerungen beim Top-Down-Entwurf jeweils in eine eigene Funktion zu packen. So wird kein Arbeitsschritt vergessen und die Komplexität wird schrittweise sichtbar reduziert.

8.2 Rückgabewerte und Rückkehr

Man kann sich eine Funktion vereinfacht so vorstellen, dass anstelle ihres Namens einfach der in der Funktion stehende Anweisungsblock ausgeführt wird. In der realen Welt des Compilers läuft es etwas anders, der übersetzte Code des Funktionsblocks steht an einer völlig anderen Stelle im Hauptspeicher des Rechners wie der Code des Hauptprogramms. Trifft der Programmablauf auf einen Funktionsaufruf, „rettet" er die aktuelle Situation (also seinen aktuellen Zustand, den Inhalt der Parameter und noch so ein paar Kleinigkeiten, die für den korrekten Programmablauf erforderlich sind) in einem speziellen Speicherbereich, dem *Stack*. Diesen *Stack* kann man sich wie einen Stapel Papier vorstellen, auf jedem Blatt steht genau so ein Rettungspaket.

Endet eine Funktion, wird das oberste „Blatt" genommen und alle betriebswichtigen Daten wieder da eingesetzt, wo sie vorher gestanden haben und die Programmausführung fortgesetzt. Diese Technik stellt sicher, dass es selbst bei den tief verschachtelten Funktionsaufrufen etwa in einer Rekursion (siehe Seite 87) immer an der richtigen Stelle weiter geht und man Funktionen aus anderen Funktionen oder sogar aus sich selbst heraus aufrufen kann.

8.3 Parameter

Funktionen können Parameter haben, um Werte auf einem eleganten Weg in die Funktion hinein zu bekommen. Die Verwendung von globalen Variablen wäre auch möglich, hat aber den entscheidenden Nachteil, dass erst durch intensives Untersuchen des Quelltextes heraus kommt, welche globalen Variablen verwendet werden und welche nicht. Übergibt man alle notwendigen Informationen als Parameter, ist mit einer Zeile klar, was in die Funktion hinein und was aus ihr heraus geht. Man gewinnt dadurch also zum einen an Übersichtlichkeit und zum anderen an Flexibilität.

Die Parameterliste sieht genauso aus wie eine Variablendeklaration und verhält sich auch genauso. Die hier deklarierten Variablen werden für die Dauer der Funktion angelegt und sind innerhalb des Blocks definiert und verfügbar. Am Ende der Funktion (also mit Erreichen der schließenden geschweiften Klammer) werden die Variablen zerstört.

Zur Veranschaulichung erweitern wir unsere Funktion `sterne()` um einen Parameter, der die Anzahl der Sterne angibt, die in der Zeile ausgegeben werden sollen:

```
void sterne(int anzahl)
{
      while (anzahl > 0){
            cout << "*";
            anzahl--;
      }
      cout << endl;
}
```

Der Parameter `anzahl` steht hier als lokale Variable innerhalb der Funktion zur Verfügung. Beim Aufruf der Funktion aus dem Hauptprogramm wird der Wert der Laufvariablen `i` in den Parameter kopiert, die Änderung innerhalb der Funktion hat also keine Auswirkung auf den Wert des Aufrufers:

```
int main()
{
      for (int i = 1; i < 10; i++)
            sterne(i);// Funktionsaufruf
}
```

Können Sie aus dem Listing entnehmen, was dieses Programm macht? Überprüfen Sie ihre Vermutung und schreiben Sie das Programm einfach einmal selbst.

Wichtig: Parameterdeklaration und Aufruf der Funktion müssen zueinander passen, dass heißt, dass Anzahl, Reihenfolge und Typ der Parameter an beiden Stellen übereinstimmen müssen.

8.3.1 Prototypen

Es gibt Fälle, da rufen sich zwei Funktionen gegenseitig auf. Das funktioniert auch prima, dank der Stack-Verwendung gibt das keine Probleme. Das einzige Problem ist der Compiler, der ja darauf besteht, dass eine Funktion vor ihrem Aufruf bekannt sein muss. Das ist in diesem Fall aber schlecht möglich, es kann nur eine erste Funktion geben. Noch einfacher ist der Fall, wenn man die Funktionen hinter der `main()`-Funktion schreiben möchte, weil es einfach übersichtlicher ist, oder der Dozent aus didaktischen Gründen darauf besteht[16]. Selbstverständlich gibt es dafür eine Lösung, den Funktionsprototypen.

Ein Funktionsprototyp stellt nichts weiter wie die erste Zeile (die Kopfzeile) der Funktion dar, statt des Funktionsblock steht hier aber nur ein Semikolon. Unsere Funktion sterne sähe als Prototyp also so aus:

```
void sterne(int anzahl);
```

Diese Zeile reicht dem Compiler, um im späteren Verlauf der Übersetzung den Aufruf und die Verwendung der Funktion `sterne(...)` überprüfen zu können, er hat ja die Parameterliste bereits gesehen.

Schauen wir uns dazu das etwas erweiterte Programm an:

```
void leerzeichen(int anzahl);
void sterne(int anzahl);

void sterne(int anzahl)
{
      while (anzahl > 0){
            cout << "*";
            anzahl--;
      }
      cout << endl;
}

int main()
{
      for (int i = 1; i < 10; i++)
            sterne(i);  // Funktionsaufruf
      leerzeichen(2);
}
```

Wir haben eine zweite Funktion `leerzeichen (int anzahl)` deklariert, die analog zu `sterne()` eine Anzahl Leerzeichen ausgibt und rufen diese im Hauptprogramm auch mit dem Argument 2 auf.

16 Ich persönlich bestehe sogar darauf, dass alle Funktionen in einer eigenen Datei stehen

Der Compiler trifft auf die Deklaration der neuen Funktion und „notiert“ diese auf seiner Liste, die Funktion sterne() wird direkt dahinter vermerkt.

Der folgende Funktionsblock von sterne() wird als Objektmodul erzeugt (also übersetzt) und dem Linker zur Verfügung gestellt. Zum Abschluss wird auch die main()-Funktion übersetzt und als Objektmodul an den Linker übergeben. .

Der Compiler überprüft beim Aufruf jeder Funktion, ob die Parameterliste mit dem Prototypen übereinstimmt. Wer jetzt glaubt, dass der Compiler sich in diesem Fall beschwert, irrt, denn die Aufrufe aller Funktionen stimmen mit den Prototypen überein, also kein Fehler für den Compiler.

Ein Problem taucht erst bei der nächsten Stufe des Übersetzungsvorgangs auf, dem Linker. Der merkt nämlich beim Zusammenbau der einzelnen Objektmodule (jede Funktion ergibt nach dem Übersetzen ein Objektmodul), dass da für eine Funktion zwar der Prototyp existiert, aber kein Objektmodul. Den Compiler hat das nicht interessiert, ein Objektmodul kann ja sonst woher kommen, zum Beispiel von einem anderen Programmierer. Ausbaden muss das Dilemma der Programmierer, der den „Build“-Knopf gedrückt hat, also Sie. Dummerweise sind solche Fehler nicht leicht zu finden, es gibt selten eine Zeilennummer, und die Fehlermeldung ist nicht unbedingt sehr leicht verständlich:

```
1>Source.obj : error LNK2019: unresolved external symbol "void __cdecl
leerzeichen(int)" (?leerzeichen@@YAXH@Z) referenced in function _main

1>E:\testordner\CPP-TestProject4\Debug\CPP-TestProject4.exe : fatal error
LNK1120: 1 unresolved externals

========== Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped ==========
```

Der Verursacher ist die beue Funktion leerzeichen() in unserem Programm, sieist zwar durch den Prototypen in der ersten Zeile angekündigt, wird aber innerhalb dieses Projekts nicht mehr definiert (mit Inhalt gefüllt). Beschränkt man sich auf die rot markierten Teile der Fehlermeldung (die der Linker innerhalb der IDE allerdings nicht farblich markiert!), kann man in etwa erahnen, was das Problem ist, die Ursache suchen und Abhilfe schaffen. Übersetzt bedeutet das in etwa: Es gibt einen Bezeichner „leerzeichen“ (external symbol) mit einem int-Parameter, der in der main()-Funktion aufgerufen wird (referenced). aber nicht aufgelöst (unresolved) wurde.

Hier ist die Lösung einfach: der fehlende Funktionsrumpf muss her. Oft sind es aber Tippfehler in den Parametern, die kaum auffallen und die Fehlersuche zu einer Geduldsprobe werden lassen.

Es gibt also jetzt drei Stellen, an denen eine Funktion zu Problemen führen kann. Grundsätzlich gilt: Funktionsname, Parametertyp, Parameteranzahl, Parameterreihenfolge und Rückgabewert müssen an allen drei Stellen identisch sein. Jeder Unterschied hier bedeutet für den Compiler die Definition einer neuen Funktion und für den Linker einen potentiellen Fehler.

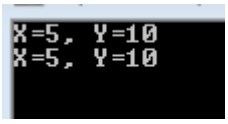
Die Namen der Parameter fehlen hier absichtlich. Sie sind beliebig und dürfen im Prototypen sogar einfach fehlen, für den Compiler reicht die Angabe des Datentyps. Statt `void sterne(int anzahl);` wäre auch `void sterne(int);` erlaubt und im professionellen Umfeld üblich

8.3.2 Referenzen als Parameter

Alle Parameter, die wir bisher in die Funktion hineingegeben haben wurden als Kopie übergeben. Damit gelangen zwar Informationen in die Funktion hinein, aber nicht wieder hinaus. Lediglich der `return`-

Wert steht als Rückkanal zur Verfügung und kann genau einen Wert zurück liefern. Was aber tun, wenn man mehr als einen Wert zurück haben möchte? Betrachten wir folgendes Problem:

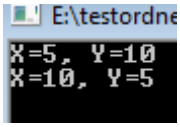
```
void tausche(int a, int b)
{
	int temp = a;
	a = b;
	b = temp;
}
int main()
{
	int x = 5, y = 10;
	cout << "X=" << x << ", Y=" << y << endl;
	tausche(x, y);
	cout << "X=" << x << ", Y=" << y << endl;
}
```



Die Funktion `tausche(a,b)` hat zwei Parameter, die innerhalb der Funktion korrekt getauscht werden. Da wir aber bereits wissen, dass Parameter als Kopie übergeben werden, und sämtliche Änderungen innerhalb der Funktion beim Aufrufer keinerlei Spuren beim Aufrufer hinterlassen, wundert das Ergebnis nicht besonders. Der Aufruf der Funktion war also so völlig sinnlos.

Natürlich gibt es für dieses Problem eine Lösung, sie ist so dezent und unauffällig, dass Sie sie beim flüchtigen Vergleich der beiden Listings vermutlich nicht sofort finden werden:

```
void tausche(int& a, int& b)
{
	int temp = a;
	a = b;
	b = temp;
}


int main()
{
	int x = 5, y = 10;
	cout << "X=" << x << ", Y=" << y << endl;
	tausche(x, y);
	cout << "X=" << x << ", Y=" << y << endl;
}
```



Haben Sie den Unterschied gefunden? Tipp: er befindet sich in der ersten Zeile! Das unscheinbare & an dem Datentyp (genauer: es darf irgendwo zwischen den Datentyp und der Variablen stehen) macht den gewünschten Unterschied. Jetzt werden die so markierten Parameter nicht mehr als Kopie sondern als *Referenz* übergeben. Eine *Referenz* enthält nicht den Inhalt einer Variablen, sondern einen *Verweis* (man könnte auch flapsig von der Adresse sprechen) auf die Variable selbst. Die Funktion erhält also nicht mehr einen Wert als Parameter, sondern einen Verweis auf den Speicherplatz des übergebenen Wertes. Und jetzt wirken sich Änderungen innerhalb der Funktion ganz automatisch auf den Aufrufer aus, weil eben keine Kopie mehr verwendet wird, sondern über die Referenz direkt auf die Originalvariable zugegriffen wird. Voila!

Werden Werte als Referenz übergeben, klappt dies natürlich auch nur, wenn man eine Referenz auf diesen Wert anlegen kann. Mit Variablen ist das ganz klar möglich, aber nicht mit Konstanten. Ein Aufruf von `tausche (4,5)` oder auch von `tausche (a,7)` führt also zu Fehlermeldungen

Natürlich kann man *Referenzparameter* und *Wertparameter* (so heißen die „normalen" Parameter) auch bunt mischen, ganz wie man es braucht. Und analog zu dem bereits über globale Variablen gesagtem gilt auch hier: der Einsatz muss gut begründet (und kommentiert) sein, da er auf den ersten Blick nicht erkennbar ist.

8.3.3 Weitergehend: Zeiger als Parameter

Das gleiche Ergebnis erzielt man auch, wenn man statt einem Wert den Zeiger auf diesen Wert an die Funktion übergibt. Zeiger behandeln wir erst auf Seite 108, hier soll es genügen, dass ein Zeiger ähnlich wie eine Referenz funktioniert, also die Adresse einer Speicherstelle übergeben wird. Der technische Unterschied besteht darin, dass bei der Referenz direkt mit der Adresse der Variablen gearbeitet wird, während bei einem Zeiger eine Kopie der Adresse an eine Zeigervariable übergeben wird. Der Unterschied ist auf den ersten Blick nicht wirklich groß und soll daher hier auch nicht weiter betrachtet werden, verlangt aber ein anderes Vorgehen.

Hier sieht man dann aber bereits im Listing, sowohl bei der Funktionsdeklaration als auch beim Aufruf der Funktion, dass hier die Originalwerte geändert werden, denn an beiden Stellen ändert sich etwas:

```
void tausche(int* a, int* b)
{
      int temp = *a;
      *a = *b;
      *b = temp;
}


int main()
{
      int x = 5, y = 10;
      cout << "X=" << x << ", Y=" << y << endl;
      tausche(&x, &y);
      cout << "X=" << x << ", Y=" << y << endl;
}
```

Die Funktion erwartet jetzt für jeden Parameter einen „Zeiger auf einen `int`-Wert", das bedeutet das Sternchen * am Datentyp. a und b sind also jetzt Variablen, die eine Adresse (einen Zeiger) aufnehmen können, nicht mehr einen Integer!

Auch im Funktionsblock muss sich etwas ändern, damit das Ergebnis wieder stimmt. Die Hilfsvariable `temp` ist nach wie vor vom Typ `int`, um an den Wert zu kommen, der sich hinter dem Zeiger a (der ja eine Kopie auf die Adresse der Variablen x ist) verbirgt, muss der *Dereferenzierungsoperator* `*` benutzt werden. Der macht nichts anderes, als an die Speicherstelle, die in der Zeiger-Variablen a gespeichert ist, zu gehen und den Wert aus dieser Speicherstelle zu holen.

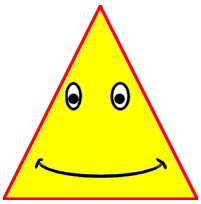
In der Zeile `*a = *b;` wird jetzt doppelt gezeigert. Die rechte Seite ist ja inhaltlich identisch mit der vorherigen Zeile, es wird der Inhalt von der Speicherstelle besorgt, auf den die Zeiger-Variable b zeigt (die ja wiederum eine Kopie der Adresse der Variablen y im Hauptprogramm ist). Und dieser Wert wird an der Stelle abgespeichert, auf den die Zeiger-Variable a zeigt (was ja wiederum eine Kopie der Adresse der Variablen x aus dem Hauptprogramm ist). Damit wäre der erste Teil der Tauscherei abgeschlossen, a zeigt auf den bereits geänderten Wert.

(Überlegen Sie einmal, was passiert, wenn man bei dieser Anweisung die beiden Sterne entfernt (also quasi „kürzt"). Aber erst überlegen, dann ausprobieren!)

Fehlt noch der Abschluss, der gerettete Wert, auf den a zu Beginn der Funktion gezeigt hat, muss ja an die Speicherstelle, auf die b zeigt. Das erledigt die Zeile `*b = temp;`

Die Hilfsvariable `temp` hat keinen *, weil sie ja den Inhalt der angezeigten Speicherstelle erhalten hat und nicht auf eine Speicherstelle zeigt. Daher muss auch nur das Ziel der Zeiger-Variablen b geändert werden und der Tausch ist komplett.

Bei soviel Modifikationen muss sich auch der Aufruf ändern, statt dem Wert der Variablen wird die Adresse übergeben, das bedeutet das & an dieser Stelle (gewisse Ähnlichkeiten zur Referenz sind nicht zufällig, es ist aber technisch ein anderer Vorgang. Hier zeigt sich aber wieder einmal, dass in C++ versucht wird, ein bestimmtes Operatorzeichen immer für ähnliche Operationen einzusetzen, wenn man schon gezwungen ist, Symbole mehrfach für unterschiedliche Zwecke zu verwenden).



Übrigens: es ist nicht verboten (und gar nicht mal so selten), einen Zeiger auf einen Zeiger auf eine Variable zu benutzen. Diese doppelte Verzeigerung endet dann auch mit einem Doppelstern **.

Dieses Konstrukt trifft man regelmäßig an, wenn man Zeiger (z.B. auf den Beginn einer verketteten Liste) mittels einer Funktion ändern möchte und daher als Parameter übergibt. Eine Änderung des Zeigers geht nur, wenn die Zeigervariable als Referenz übergeben wird, das ist dann gleichbedeutend mit **.

8.3.4 Weitergehend: variable Anzahl von Parametern und vorbelegte Parameter

Die Grundregel von Seite 77, dass Anzahl, Reihenfolge und Typ der Parameter bei Aufruf einer Funktion und ihrer Deklaration identisch sein müssen, ist zwar grundsätzlich richtig, es gibt aber für faule Programmierer unter bestimmten Umständen, die Möglichkeit, dieses Prinzip etwas zu lockern.

Man darf nämlich bereits bei der Definition einer Funktion Parameter mit Werten belegen. Wird die Funktion dann aufgerufen, können die so bereits belegten Parameter weggelassen werden, dafür werden dann die bereits vorher definierten Werte verwendet. Auf diese Weise kann man Parameter, die sich selten ändern , sinnvollll vorbesetzen und muss sich dann später nicht mehr darum kümmern:

```
void zeichne_linien(char ch, int laenge = 80, int anzahl = 1)
{
      // Zeichnet anzahl Linien der Länge laenge aus dem Zeichen ch
}
```

Der Parameter `laenge` wird hier mit dem Wert 80 und der Parameter `anzahl` mit 1 vorbesetzt.

Diese Funktion verlangt also zwingend nach dem ersten Parameter, der angibt, welches Zeichen für die Linie verwendet werden soll. Zusätzlich ist es möglich, die Länge anzugeben, wenn einem die 80 Zeichen nicht gefallen. Und wer mehr als eine Linie braucht, kann auch das angeben:

```
zeichne_linien('+'); // liefert 1 Zeile +++++++++++++++(80 Stück)
zeichne_linien(42,40); // liefert:****************************(40 Stück)
zeichne_linien(0x23, 25,3);// drei Zeilen aus #########(25 Stück)
```

Die Parameter dürfen aber immer nur von hinten an weggelassen werden, damit der Compiler eine Chance hat, die Ergänzungen vorzunehmen. (Nebenbei sehen Sie hier mal drei verschiedenen Arten, einen Wert vom Typ `char` zu definieren).

8.3.5 Weitergehend: Überladen von Funktionen

In vielen Programmiersprachen müssen Namen von Funktionen innerhalb eines Projektes eindeutig sein. Nicht so in C++, da ist es ein wenig anders. Hier darf es Funktionen mit gleichen Namen geben, wenn sich diese dafür in der Parameterliste (nicht im Rückgabewert!) unterscheiden[17]. So können inhaltlich gleiche, aber vom Ablauf her unterschiedliche Funktionen unter ein und demselben Namen geführt werden.

```
double flaeche(double a, double b)
{
      return (a*b);
}
double flaeche(double radius){
      return 3.14159*radius*radius;
}
```

Beide Funktionen berechnen die Fläche, die erste Funktion die eines Rechtecks mit den Seiten a und b und die zweite den Inhalt eines Kreises aus dem Radius. Der Compiler erkennt an der Parameterliste, welche Funktion gemeint ist und setzt den richtigen Aufruf ein. Der Unterschied der Funktionen muss in der Parameterliste liegen, ein unterschiedlicher Rückgabetyp reicht nicht aus!

8.3.6 Weitergehend: Arrays als Parameter

Arrays (Felder) werden ja erst im Kapitel 9.1 auf Seite 97 behandelt, daher hier nur der Sonderfall des Arrays als Parameter.

Ein Array ist eine Folge von identischen Daten, die als Gesamtheit das Array bilden, bei dem aber über einen Index auf jedes einzelne Datenelement zugegriffen werden kann. Man kann sich das Array als Kette von einzelnen Variablen vorstellen, der Index gibt an, in welchen Kettenglied der Wert zu suchen bzw. abzulegen ist.

Arrays können sehr groß werden, deshalb haben die Erfinder der Sprache C++ beschlossen, dass bei diesen Datenobjekten grundsätzlich nicht mit Kopien an die Funktion gearbeitet wird. Es muss also anders gemacht werden.

Für unsere Betrachtung definieren wir im Hauptprogramm `main()` ein Array `feld_main` aus fünf `int`-Werten:

17 Man nennt die Kombination aus Funktionsnamen und den Parametern **Signatur**, diese muss für jede Funktion spezifisch sein

```
int feld_main[5];
```

Dazu passend gibt es drei Funktionsprotoypen, die jeweils ein Array als Parameter erwarten:

```
void func_a(int  feld[5]);
void func_b(int  feld[]);
void func_c(int *feld);
```

Bei func_a ist der Fall klar. Es wird ein Array mit fünf Elementen übergeben, das passt zum definierten feld_main. Denkt man, Ist aber leider nicht so. Die fünf Elemente aus dem Feld im Hauptprogramm werden eben leider nicht in die fünf Elemente des Feldes der Parameterliste kopiert, auch wenn das hier von der Datenmenge her kein Problem wäre. Sondern der C++-Compiler hält sich an den C++-Standard und übergibt lediglich die Adresse des ersten Elementes aus dem Array, also den Anfang des Speicherbereichs, den das Array belegt. Und es kommt noch schlimmer, die Funktion erhält keinerlei Information darüber, wie groß das Array tatsächlich ist. Und wenn die Funktion nicht weiß, wo das Array endet, ist der Fehlerfall bereits vorprogrammiert und wird mit 100% Wahrscheinlichkeit auch eintreten. Hier muss also Vorsorge getroffen werden und als guter Programmierer gibt es bei Funktionen mit einem Array als Parameter auch immer einen zweiten Parameter, der die Länge des Arrays auch an die Funktion übergibt. Nur so kann die Funktion sinnvoll mit dem Array arbeiten.

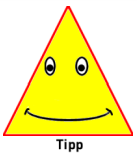
Bei der Funktion func_b verhält es sich ganz genauso. Wenn die Größe des Arrays sowieso nicht übergeben wird, dann muss man sie auch nicht hinschreiben.

Und func_c erwartet per definitionem (siehe Seite. 82) sowieso einen Zeiger, macht dies aber mittels des * auch sehr deutlich. Und diesen Zeiger bekommt sie ganz automatisch.

Unterm Strich: alle drei Funktionsprototypen sind gleichwertig, in der Praxis hat sich die mittlere Schreibweise (func_b) durchgesetzt.

Grundsätzlich wird also bei der Übergabe eines Array immer mit den Originalwerten gearbeitet, entsprechende Vorsicht muss man hier walten lassen. Man kann dies aber durch das zusätzliche Schlüsselwort `const` verhindern:

```
void func_b (const int  feld[]);
```



Unabhängig davon ist es eine gute Idee, die Größe des benutzten Arrays oder die Zahl der vorhandenen (benutzten) Elemente als weiteren Parameter an die Funktion zu übergeben. Sonst passiert es sehr schnell, dass man mit einer Schleife über die Feldgröße hinaus zugreift und so einen Laufzeitfehler provoziert.

8.4 Globale, lokale und statische Variablen

Über den Geltungsbereich von Variablen haben wir bisher schon an einigen Stellen etwas gehört. In Verbindung mit Funktionen ist aber eine ausführlichere Betrachtung erforderlich. Denn eine Funktion ist im Idealfall ein in sich abgeschlossenes Gebilde, dass einzig über die Parameterliste und den `return`-Wert mit seiner Umwelt kommuniziert.

8.4.1 Globale Variablen

Globale Variablen werden außerhalb jeglicher Blöcke definiert und können an jeder Stelle im Programm benutzt werden. Theoretisch werden sie sogar vom Compiler auf Null gesetzt, verlassen sollte man sich darauf aber nicht.

Vermeiden Sie globale Variablen, wenn es irgendwie geht. Und irgendwie geht es eigentlich immer. Und wenn Sie es nicht vermeiden können, lassen Sie sich einen guten Grund dafür einfallen, ihr Betreuer im Praktikum wird sie garantiert danach fragen. In der ersten Übung gilt die Ausrede *„Wir kennen noch nichts anderes“* nur für die ersten Minuten. Denn Sie kennen aus der Vorlesung nur lokale Variablen,die innerhalb der main()-Funktion definiert werden. Neu ist nur der korrekte Begriff dafür.

8.4.2 Lokale Variablen

Jede Variable, die innerhalb eines Blocks definiert wird, ist lokal für diesen Block. Dazu gehören bei Funktionen auch die Variablen aus der Parameterliste, auch wenn diese streng genommen vor dem Block definiert werden. Ihre Lebensdauer beginnt mit der öffnenden geschweiften Klammer und endet mit der schließenden geschweiften Klammer.

Werden in diesem Block neue Blöcke geschachtelt, sind die lokalen Variablen des äußeren Blocks gleichzeitig globale Variable der inneren Blöcke. Es sei denn, der innere Block definiert eine gleichlautende Variable, dann überdeckt diese lokale Variable die Sichtbarkeit der äußeren Variable gleichen Namens.

Die Idee hinter den lokalen Variablen ist das Vermeiden von *Seiteneffekten*, die bei der Verwendung globaler Variablen zwangsläufig auftreten. Stellen Sie sich einfach nur einmal vor, was passiert, wenn Sie aus einer `for`-Schleife mit der Laufvariablen i eine Funktion aufrufen, die selbst auch eine `for`-Schleife mit einer Laufvariablen i verwendet. Wenn ihr Vorstellungsvermögen dazu nicht ausreicht, schreiben Sie ein solches Programm einfach, die Funktion `sterne()` aus Kapitel 8.3 ist ein guter Ausgangspunkt dafür.

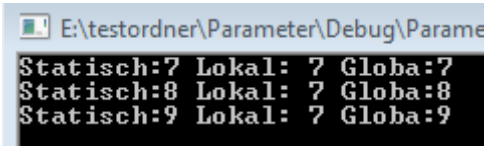
Lokale Variablen werden auf dem Stack angelegt (das ist der Speicherbereich, auf dem auch die Parameter und Rücksprungadressen abgelegt werden), weil dort das Aufräumen sehr einfach geht: einfach den Zeiger (der heißt daher auch Stackpointer) für den unbenutzten Bereich umsetzen, fertig. Und lokale Variablen werden grundsätzlich nicht initialisiert, sie enthalten also nach ihrer Definition keinerlei sinnvolle Werte, geschweige denn eine Null. Die Konsequenz daraus ist, dass diese Variablen mindestens einmal auf der linken Seite einer Zuweisung gestanden haben müssen, bevor sie auf der rechten Seite auftauchen. Das sollte sinnvollerweise gleich bei der Definition geschehen, also z.B. `double d = 0.0;`

8.4.3 Statische Variablen

Eine statische Variable ist die Lösung, wenn Sie den Inhalt einer Variablen doch einmal über die Lebensdauer der Funktion hinaus brauchen. Dazu würde man in anderen Programmiersprachen eine globale Variable verwende, nicht aber in C++.

Wird eine lokale Variable zusätzlich als `static` definiert, bleibt sowohl die Variable als auch ihr Inhalt von einem zum nächsten Funktionsaufruf erhalten. Damit könnte man also beispielsweise zählen, wie oft eine Funktion aufgerufen wird. Außerhalb der Funktion ist diese Variable im Unterschied zu einer globalen Variable nicht sichtbar!

```
int global = 7;

void func()
{
	static int statisch = 7;
	int lokal = 7;
	cout << "Statisch:" << statisch << " Lokal: " << lokal
		<< " Globa:" << global << endl;
	statisch++;
	lokal++;
	global++;
}

int main()
{
	func(); func(); func();
}
```



Das Ergebnis sieht so aus. Das Hochzählen der statischen Variable funktioniert genauso wie das Hochzählen der globalen Variable, allerdings mit dem Vorteil, dass es eben keine globale, sondern eine lokale Variable ist.

Der Versuch, die statische Variable im Hauptprogramm zu ändern würde der Compiler sofort mit einer Fehlermeldung bestrafen. Lediglich die globale Variable darf überall straflos geändert werden, mit allen fatalen Folgen, die das haben kann. Falls Sie die Fehlermeldung einmal sehen möchten, schreiben Sie das Programm einfach selbst. Bei der Gelegenheit können Sie auch das fehlende „l" in der cout-Zeile ergänzen.

8.5 Rekursion

Von einer Rekursion spricht man, wenn eine Funktion sich selbst wieder aufruft, auch dann, wenn der Aufruf indirekt über eine andere Funktion stattfindet. Grundsätzlich lässt sich jedes rekursive Problem auch per Iteration (also durch eine Schleife) lösen, aber in den meisten Fällen geht es eleganter durch die Rekursion.

Als wichtigste Regel gilt: Bei einer Rekursion muss man eine definierte Abbruchbedingung haben, bei der die Rekursion beendet wird, weil sonst eine unendliche Rekursion entsteht. Ist diese Abbruchbedingung noch nicht erreicht, ruft die Funktion sich selbst (oder indirekt über eine andere Funktion) mit einem geänderten Argument selbst wieder auf.

Möglich wird dies nur, weil hier das Konzept der lokalen Variablen greift und bei jedem Funktionsaufruf ein frischer Satz Variablen auf dem Stack erzeugt wird und die Variablen der vorherigen Aufrufe dahinter verdeckt werden.

Das zentrale Beispiel in der Literatur für eine rekursive Berechnung ist die Fakultät einer Zahl n!. Sie ist definiert als n! = 1* 2 * 3 * 4 * 5*....*(n-1) * n. Außerdem ist 0!=1 definiert. Das ist dann auch die Abbruchbedingung, bei der die Rekursion endet, die vor dem erneuten Aufruf zuerst geprüft wird.

In Worten formuliert sieht der rekursive Ansatz so aus:

Solange n größer Null ist, berechnet sich die Fakultät von n aus dem Produkt aus n und der Fakultät von (n-1). Damit ist die Lösung komplett und sieht als Programm so aus:

```
long fakultaet (unsigned int i)
{
      if (i <= 1)
            return 1;
      else return i * fakultaet (i - 1);
}
int main() {
      cout << fakultaet(15);
}
```

Sieht doch gut aus, oder? Zum Vergleich hier die iterative Lösung mittels `for`-Schleife:

```
long fakultaet2(unsigned int i)
{
      long fak = 1;
      for (int k = 2; k <= i; k++)
            fak = fak * k;
      return fak;
}
int main() {
      cout << fakultaet2(15);
}
```

Auch nicht viel länger, für den Rechner sogar einfacher, weil er nur einmal einen Satz Variablen auf dem Stack erzeugen muss, während die rekursive Variante den Stack und damit den Rechner ziemlich fordert.

Es gibt aber Probleme, die möchte man nicht iterativ lösen. Um beispielsweise den Weg aus einem Labyrinth zu finden, kann man dieses Verfahren anwenden: wenn der Weg nicht weiter führt (Sackgasse) wähle an der letzten Abzweigung den linken Weg. Sonst gehe rechts. Die Richtigkeit dieses Verfahrens können Sie leicht mit einem Rätselheft und einem Bleistift überprüfen. Und wenn Sie den Ausgang gefunden haben versuchen Sie sich an einer iterativen Lösung. Diese hätte ich dann gerne für die nächste Version dieses Werkes.

Ihre Stärke haben rekursive Algorithmen beim Suchen und Sortieren, weil jeder Rekursionsschritt das Problem kleiner macht (oftmals auf die Hälfte reduziert).

Grundsätzlich gilt aber, dass man jede Rekursion durch eine Iteration ersetzen kann.

8.6 Aufteilung in mehrere Dateien

Wenn mehrere Programmierer an einem Projekt arbeiten ist es völlig normal, dass jeder seinen Quellcode in eine eigene Datei schreibt. Der Aufwand mit einer Datei für alle wäre doch ein wenig groß.

Man kann dieses Prinzip aber auch bei kleineren Projekten sinnvoll anwenden, in anderen Programmiersprachen wird sogar eine Aufteilung in verschiedene Dateien zwangsweise verlangt. Und spätestens beim Einsatz von Klassen geht es nicht mehr ohne Aufteilung in einzelne Dateien.

8.6.1 Header-Datei

Um ein Projekt übersetzen (nicht linken!) zu können, benötigt der Compiler nicht alle Quelltexte, sondern ihm genügen an den meisten Stellen nur die Kopfzeilen der aufgerufenen Funktionen (also die Prototypen). Damit kann er erfolgreich überprüfen, ob der Aufruf (Anzahl, Typ und Reihenfolge) zueinander

passen und dann mit der Übersetzung des Objektmoduls fortfahren. Erst der Linker benötigt alle Objektmodule, um das Projekt zusammen bauen zu können (wir hatten da schon mal einen solchen Fall auf Seite 80).

Die Kopfzeilen werden gerne auch mit ihrer englischen Benennung **Header** beschrieben, daher sammelt man diese Headerzeilen am besten in einer Header-Datei. Weil das ein sehr häufiger Vorgang ist, bietet der Editor diesen Dateityp auch an, es gibt sogar eine besondere Namenserweiterung (Extension) dafür, nämlich das .h

In der Header-Datei werden die Kopfzeilen (oder auch Prototypen) aller verwendeten Funktionen und die selbst kreierten Datentypen zusammengefasst. Dann muss man nur noch diese Header-Datei mittels der Direktive `#include"Header-Datei"` zu Beginn seines eigentlichen Programms einbinden und schon stehen alle Funktionen zur Verfügung.

Im Unterschied zu den mitgelieferten Header-Dateien (wie z.B.`iostream`) wird der Dateiname einer eigenen Header-Datei in doppelte Anführungszeichen " gesetzt, weil die Datei sich im aktuellen Projektverzeichnis befindet und nicht im Lieferumfang des Compilers.

Um doppeltes Definieren zu vermeiden, werden Header-Dateien üblicherweise mit diesen beiden Zeilen

```
#ifndef Name_der_Datei
#define Name_der_Datei
```

begonnen und enden mit einem

```
#endif
```

Mit `#ifndef` (=if not defined) wird sicher gestellt, dass die Datei nur dann eingebunden wird, wenn sie bisher noch nicht eingebunden war.

Visual Studio kennt zusätzlich die Direktive `#pragma once` die wirksam verhindert, dass eine Header-Datei mehrfach eingebunden wird, wen sie in jeder Datei in der ersten Zeile steht. Obwohl diese Schreibweise nicht im Standard enthalten ist, funktioniert sie auch bei den meisten anderen Entwicklungsumgebungen. Microsoft setzt eben eigene Standards.

8.6.2 Quelltext-Datei

Und wenn man schon einmal alle Header aus dem eigentlichen Programmzeilen heraus hat, kann man einen Schritt weiter gehen und auch alle Funktionen, außer dem Hauptprogramm, ebenfalls in eine eigene Datei auslagern, das entspricht dann einer Bibliotheksdatei.

Auch die muss (in den meisten Fällen jedenfalls) zunächst einmal die zugehörige Header-Datei einbinden. Und dann folgt die Sammlung aller Funktionen, die zu dieser Header-Datei gehören. In dieser Datei allerdings stehen die Funktionen ausprogrammiert drin, also mit allen Anweisungen.

Ausblick: in der objektorientierten Programmierung ist diese Trennung in unterschiedliche Dateien dann verpflichtend, selbst die Namen der Dateien sind dann nicht mehr frei wählbar.

8.6.3 Modularisierung im Detail

Betrachten wir einmal die Modularisierung an ein konkretes Projekt. Dieses Projekt besteht zunächst aus einem Hauptprogramm, das nach und nach um weitere Funktionen erweitert wurde, die Übersichtlichkeit hat darunter ziemlich gelitten. Also wird das Projekt geteilt, das Hauptprogramm (und damit die grundsätzliche Steuerung) bekommt eine eigene Datei `main.cpp`, aus dieser Datei heraus werden die

einzelnen Funktionen aufgerufen, die aber in einer zweiten Datei `funktionen.cpp` abgelegt sind. Diese beiden Dateien enthalten Quelltext und haben daher die Endung `*.cpp`.

Damit das ganze funktioniert, benötigt man zusätzlich noch eine Header-Datei `funktionen.h`, die alle Funktionsprototypen enthält. Diese Datei mit der Endung `*.h` wird in beiden Quelltext-Dateien eingebunden.

Hier die drei Dateien (auszugsweise) im Überblick:

//main.cpp

```
#include "funktionen.h"
int main()
{
	// Variablendeklaration
char menu;
bool programmende = false;
bool schaltjahrOkay = true;
int ostern;
double mittelwert;
double e_hoch_t;
/** weitere Variablen ! **/
do
{
	menu = Menu();
	// Fallunterscheidung
	switch	(menu)
	{
	case '1':
// Ihre Implementierung Schaltjahr

	  schaltjahrOkay = Schaltjahr
			(2016);
	// Ausgabe aller Werte
	break;
	case '2':

	  ostern = Osterdatum (2017);
	break;
// u.s.w
```

//funktionen.h

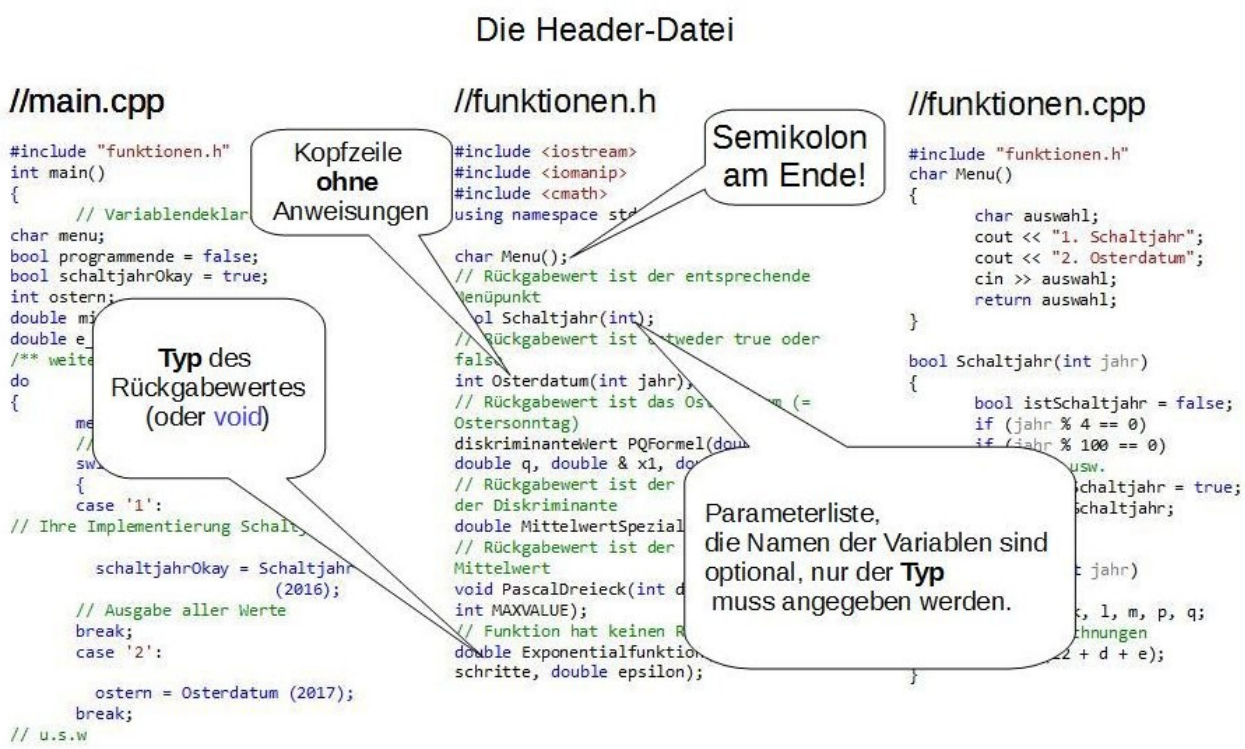
```
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

char Menu();
// Rückgabewert ist der entsprechende
Menüpunkt
bool Schaltjahr(int);
// Rückgabewert ist entweder true oder
false
int Osterdatum(int jahr);
// Rückgabewert ist das Osterdatum (=
Ostersonntag)
diskriminanteWert PQFormel(double p,
double q, double & x1, double & x2);
// Rückgabewert ist der "Bereichswert"
der Diskriminante
double MittelwertSpezial(int[], int);
// Rückgabewert ist der berechnete
Mittelwert
void PascalDreieck(int dreieck[], const
int MAXVALUE);
// Funktion hat keinen Rückgabewert
double Exponentialfunktion(int t, int
schritte, double epsilon);
```

//funktionen.cpp

```
#include "funktionen.h"
char Menu()
{
	char auswahl;
	cout << "1. Schaltjahr";
	cout << "2. Osterdatum";
	cin >> auswahl;
	return auswahl;
}

bool Schaltjahr(int jahr)
{
	bool istSchaltjahr = false;
	if (jahr % 4 == 0)
	if (jahr % 100 == 0)
		// usw.
		istSchaltjahr = true;
	return istSchaltjahr;
}

int Osterdatum(int jahr)
{
	int a, b, k, l, m, p, q;
	//viele Rechnungen
	return (22 + d + e);
}
```

Beginnen wir mit dem Hauptprogramm in der Datei `main.cpp`. Alle zusätzlichen Funktionen sind daraus verschwunden, es bleibt nur die eigentliche `main()`-Funktion übrig. Auch bei den `#include`-Anweisungen hat sich etwas getan, sie sind auf eine einzige `#include`-Anweisung zusammen geschrumpft.
Alle hier definierten Variablen sind lokale Variablen der Funktion `main()`, sie sind also außerhalb dieser Funktion nicht bekannt. Insbesondere kennt keine der Funktionen in der Datei `funktionen.cpp` diese Variablen!

Die Header-Datei (`funktionen.h`) ist das Bindeglied zwischen den beiden Quelltext-Dateien. Hier werden an erster Stelle sämtliche bisher verstreuten `#include`-Anweisungen zentral gesammelt. Vergessene oder zusätzlich erforderliche `#include`-Dateien können also pflegeleicht an nur einer Stelle eingefügt und ergänzt werden.



Nach den `#include`-Anweisungen folgen die Kopfzeilen der Funktionen, die in der zweiten Quelltext-Datei dann beschrieben und definiert werden. Es gibt nur ein paar kleine, aber wesentliche Unterschiede in der Schreibweise.

Da hier nur die Funktionskörper (oder Prototypen) der Funktionen aufgelistet werden, spielt die Reihenfolge keine Rolle (da die #include-Anweisungen ja immer zu Beginn einer *.cpp-Datei stehen sind die Namen der Funktionen auf jeden Fall vor ihrer ersten Verwendung bekannt, egal in welcher Reihenfolge sie in der Header-Datei aufgeführt werden). Ein Prototyp besteht aber nur aus der Kopfzeile, hat also insbesondere keinen Funktionsrumpf oder Funktionskörper mit Programmieranweisungen. Und wenn es keinen Funktionsrumpf gibt, darf es auch keine geschweiften Klammern geben (die bilden ja genau diesen Funktionsrumpf). Deswegen steht das Semikolon auch direkt hinter der Parameterliste und trennt damit die einzelnen Prototypen voneinander.

Die Parameterliste kann auf zwei Arten gestaltet werden, da die Namen der Parameter nicht angegeben werden müssen. Man kann also wählen, ob man die Parameter beim Namen nennt

```
int Osterdatum (int jahr);
```

oder ob man nur die Datentypen auflistet:

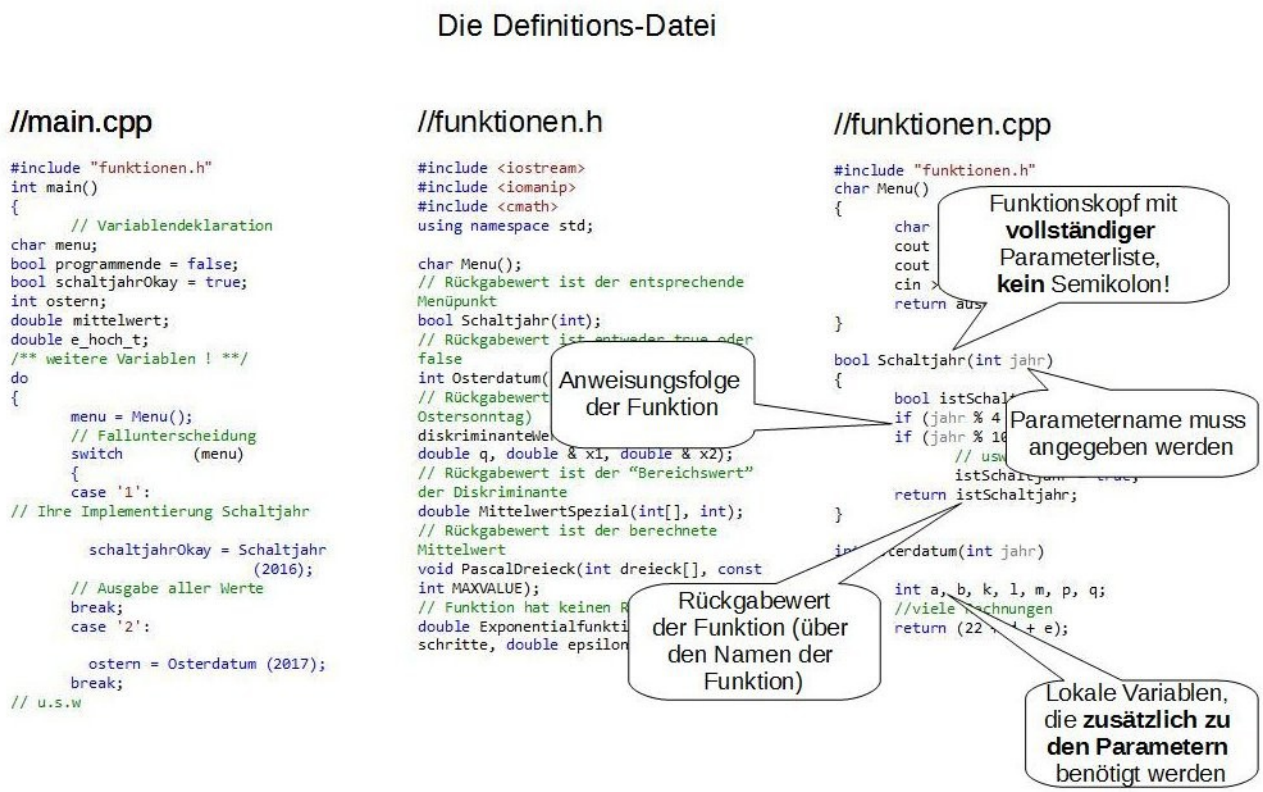
```
double MittelwertSpezial (int[],int);
```

Wer es gerne unübersichtlich und durcheinander mag, kann beide Varianten auch mischen.

Der Compiler ignoriert die Namen der Parameter allerdings, er braucht sie schlicht nicht. Die Header-Datei dient ja letztendlich nur dazu, dem Compiler eine Kurzfassung der verwendeten Funktionen zu liefern, anhand der er überprüfen kann, ob alle Aufrufe von Funktionen richtig erfolgen. Und dazu wird nur Anzahl, Typ und Reihenfolge der Parameter und des Rückgabewertes überprüft, Namen sind da nur hinderlich.

Bleibt noch die zweite Quelltext-Datei mit den eigentlichen Funktionen `funktionen.cpp`. In dieser Datei werden alle in der Header-Datei aufgeführten Funktionen definiert (daher auch der Name Definitions-Datei), also mit Leben und Programmanweisungen gefüllt.

Damit das auch klappt, müssen die Kopfzeilen der Funktionen mit den Prototypen exakt übereinstimmen. „Exakt" bedeutet hier, dass Anzahl, Typ und Reihenfolge der Parameter und des Rückgabewertes übereinstimmen müssen.



Hier müssen die Parameter jetzt Namen haben, denn unter diesem Namen sind die Parameter als lokale Variablen innerhalb der Funktion verfügbar und bekannt. Achtung: Die Namen der Parameter sollten so gewählt werden, dass man ihre Bedeutung daraus erkennen kann, dabei spielt es überhaupt keine Rolle, wie die Parameter heißen, die beim Aufruf der Funktion (in `main()`) angegeben sind!!!!! Es handelt sich hier um grundsätzlich unterschiedliche Variablen, die auch an ganz unterschiedlicher Stellen des Hauptspeichers abgelegt werden. (Natürlich dürfen die gleich heißen, bleiben aber trotzdem unterschiedliche Variablen!)[18]

Die Funktionsdefinition (der Funktionsrumpf, der Funktionskörper) soll ja etwas tun, also müssen hier auch irgendwelche Programmieranweisungen drin stehen. Solche Anweisungen werden in geschweiften Klammern zusammengefasst, davor darf also kein Semikolon stehen!

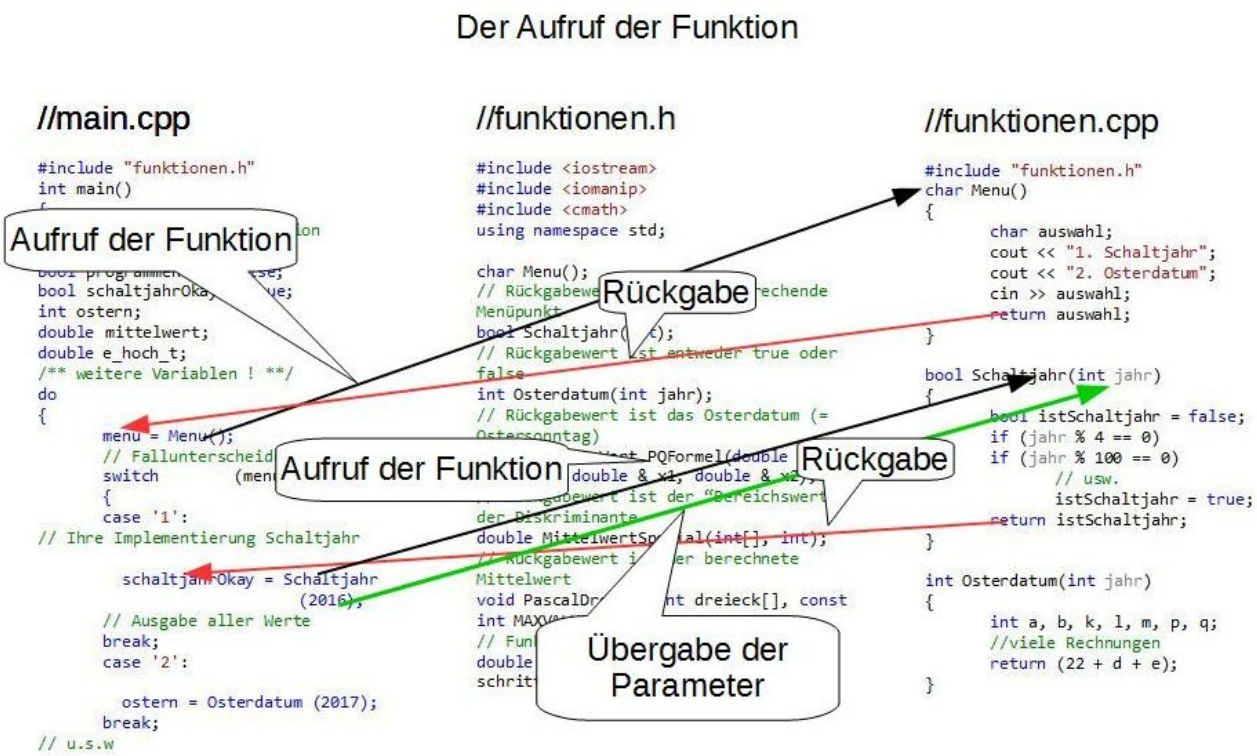
Benötigt eine Funktion weitere Variablen, um ihre Aufgabe erfüllen zu können, werden diese innerhalb der geschweiften Klammern definiert, nicht außerhalb der Funktionen. Diese lokalen Variablen leben also genauso lange, wie die Funktion arbeitet. Auch hier gilt wieder: Variablen, die im Hauptprogramm definiert wurden, sind in den Funktionen nicht bekannt, auch wenn sie den gleichen Namen tragen.

---

18 Programmieranfänger stellen sich hier oft selbst eine Falle. Sie vergeben Namen für die Parameter in der Header-Datei und kopieren dann diese Zeile in die zugehörige*.cpp-date und auch in die main-Datei, um Tippfehler zu vermeiden (eigentlich eine gute Idee). Damit heißen aber die Variablen überall gleich, das sorgt oft eher für Verwirrung und weniger für Übersicht

Kommen wir jetzt zum Zusammenspiel der Dateien, also zu dem Ablauf bei Aufruf einer Funktion aus dem Hauptprogramm (oder aus einer anderen Funktion heraus, auch `main()` ist ja eine ganz normale Funktion, sie hat lediglich die Sonderrolle, dass sie als erste vom Betriebssystem aufgerufen wird).

Wird in der `main()`-Funktion eine Funktion aufgerufen (schwarzer Pfeil), „springt" die Programmausführung an den Beginn dieser Funktion. Gleichzeitig werden die angegebenen Parameter auf den Stack kopiert, dabei gehen die Namen verloren, dort landen nur die Inhalte oder Werte (grüner Pfeil)



Jetzt wird die Programmausführung in der Funktion fortgesetzt, die erste Aktion dabei ist es, die übergebenen Parameter vom Stack zu holen und in die entsprechenden lokalen Variablen der Funktion zu kopieren. Dabei ist der Compiler ziemlich stur, der erste Wert vom Stack landet in der ersten Variablen (genauer: im ersten Parameter der Funktion), der zweite Wert im zweiten Parameter und so weiter. Spätestens jetzt sollte klar sein, warum es so wichtig ist, dass Anzahl, Typ und Reihenfolge der Parameter bei Aufruf einer Funktion und bei ihrer Definition übereinstimmen müssen. Und auch, warum in der Header-Datei die Angabe des Datentyps ausreicht, spätestens bei der Kopie auf den Stack sind die Namen überflüssig geworden und verschwinden.

Ist das alles erledigt, werden die zusätzlichen lokalen Variablen angelegt und dann beginnt die Abarbeitung der Anweisungen im Funktionsrumpf. Und irgendwann ist der Funktionsrumpf dann abgearbeitet, die letzte Anweisung ist immer eine `return`-Anweisung, mit der die Programmausführung wieder in das Hauptprogramm zurückspringt und dort weiterarbeitet.

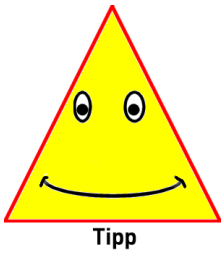
Hier müssen wir jetzt unterscheiden, ob die Funktion einen Rückgabewert hat oder nicht.Fangen wir mit dem einfacheren Fall ohne Rückgabewert an, d.h. der Funktionskopf beginnt mit `void funktionsname(…)`; Dann ist es wirklich einfach, die Programmausführung springt am Ende des Funktionsrumpfes zurück ins Hauptprogramm (genauer: in die aufrufende Funktion, auch das

Hauptprogramm main() ist ja schließlich nur eine ganz normale Funktion) und macht mit der nächsten Anweisung weiter, als wäre nichts geschehen[19].

Etwas mehr ist zu tun, wenn die Funktion einen Rückgabewert hat (also alles außer `void`). Das ist eher die Regel wie die Ausnahme. Dann wird vor dem Rücksprung zunächst dieser Rückgabewert (auch Returnwert genannt) auf dem Stack gelegt (da, wo vorher die Parameter gestanden haben), danach erfolgt der Rücksprung ins Hauptprogramm.

Das Hauptprogramm muss zuerst das Ergebnis der Funktion vom Stack holen und in der Variablen ablegen, die auf der linken Seite des Gleichheitszeichen steht (roter Pfeil, rechts vom Gleichheitszeichen steht ja der Funktionsaufruf). Ist das erledigt kann auch hier mit der nächsten Anweisung fortgefahren werden.

Eine Funktion mit Rückgabewert liefert also immer über ihren Namen einen Wert zurück, der mittels einer Zuweisung (oder auch einer Ausgabe oder ähnlichem) weiterverarbeitet werden kann. Man kann auch sagen, der Aufruf der Funktion wird während der Programmausführung durch das Ergebnis dieses Funktionsaufrufs ersetzt und kann auch genauso benutzt werden.



Die Verwendung des Wortes „*kann*“ deutet es schon an: man kann mit dem Funktionswert weiter arbeiten, man *muss* es aber nicht.

Niemand wird gezwungen, das Ergebnis einer Funktion in irgendeiner Weise auszugeben oder in eine Variable zu stecken. Allerdings stellt sich dann ganz schnell die Frage, warum die Funktion ein Ergebnis berechnet, welches dann niemand braucht. Und dann fragt nicht nur der Betreuer im Praktikum, warum das so gemacht wurde. Überlegen Sie sich die Antwort im Vorfeld, spontan dürfte das schwer werden.

8.6.4 Die Gretchenfrage[20]: Aufgabenteilung zwischen Hauptprogramm und Funktionen

Für alle Funktionen in einem Projekt- auch für die Funktion `main()`– gilt: jede Funktion soll ein Teilproblem lösen, das durch den Namen der Funktion nahezu vollständig beschrieben wird. Das Hauptprogramm übernimmt die gesamte logische Ablaufsteuerung und die Interaktion mit dem Benutzer, Alle anderen Aufgaben werden durch Funktionen erledigt. Der einzige Weg, um Informationen/Daten in die Funktion zu bekommen sind die Parameter, der einzige Weg zurück ist der Rückgabewert (und in besonderen Fällen die Verwendung von Referenzparametern).

Die meisten Funktionen „berechnen“ in irgendeiner Form ein Ergebnis, das müssen nicht zwangsläufig mathematische Ergebnisse sein. Auch das Suchen (und Finden) eines Wortes in einer Datei ist so ein Ergebnis. Hält man sich an die Regel aus dem vorherigen Absatz, kommuniziert die Funktion mit der Außenwelt nur über Parameter und Rückgabewert, sie darf also weder Eingaben noch Ausgaben machen. Diesen Part übernimmt grundsätzlich das Hauptprogramm, dass sich auch um falsche Eingaben und Fehlermeldungen kümmert. Das ist keine Willkür, sondern dadurch bleibt eine Funktion flexibel und universell einsetzbar, auch in wechselnder Umgebung.

19 Auch bei Funktionen ohne Rückgabewert (`void`-Funktionen) darf das Schlüsselwort `return` benutzt werden. Hinter `return` darf dann aber nichts angegeben werden! Es wird dann lediglich sofort zurück zum Aufrufer gewechselt, ggfs. wird die Funktion nicht bis zum Ende ausgeführt, sonder mehr oder weniger brutal abgebrochen.

20 Goethe: Faust I, https://de.wikipedia.org/wiki/Gretchenfrage

Keine Regel ohne Ausnahme, es darf natürlich auch Funktionen geben, die Ein- oder Ausgaben machen. Sie werden immer dann genutzt, wenn die Eingabe aufwändig ist oder kompliziert und das Hauptprogramm vor lauter Test und Kontrollen nicht mehr erkennbar ist. Funktionen für die Ausgabe kommen dann zum Tragen, wenn die Ausgabe komplex gestaltet werden muss und besondere Rahmenbedingungen notwendig sind (z.B. bei Spielfeldern). Diese Funktionen sind dann natürlich nur eingeschränkt wiederverwendbar.

Diese Aufteilung ergibt sich auch ganz von alleine, wenn man bei der Programmentwicklung nach dem Prinzip der sukzessiven Verfeinerung vorgeht, wie im Kapitel 3.3 ab Seite 14 beschrieben.

Ein weiterer Grund für die Aufweichung dieser Regeln liegen beim Dozenten der Veranstaltung: um die gesamte Bandbreite der Parameterübergabe in den Übungen zu zeigen, ist es notwendig, die Trennlinie zwischen Hauptprogramm und Funktion in jedem Einzelfall neu zu ziehen,.

8.6.5 extern-Deklaration

Bei der Definition von Daten(-typen) in einer Header-Datei muss man ein wenig aufpassen. Weil ja eine *Definition* nur genau einmal auftauchen darf (anders als eine *Deklaration*). Dabei hilft das Wörtchen `extern`. Es sagt dem Compiler, dass diese Definition nur ein Platzhalter ist, die eigentliche Definition an anderer Stelle steht, aber zum Übersetzen darf er die `extern`-Deklaration verwenden, die dann im Laufe des weiteren Übersetzungsvorgangs ausgetauscht wird.

Es muss also irgendwo eine Datei geben, in der dieselbe Definition **ohne** das Wörtchen `extern` steht, das darf sogar die gleiche Datei sein, in der zusätzlich dieselbe Definition als `extern` steht, solange es genau eine Stelle gibt, die ohne `extern` definiert wird.

Man könnte es auch etwas ungenau so formulieren: durch ein vorangestelltes `extern` wird für den Linker aus einer Variablen*definition* eine Variablen*deklaration*.

8.6.6 Der Präprozessor

Alle Anweisungen, die mit einem # beginnen, werden nicht durch den Compiler, sondern durch den vorgeschalteten Präprozessor abgearbeitet. Die Anweisung #include... benutzen wir ja schon seit unseren ersten Gehversuchen. Damit wird der Präprozessor angewiesen, die angegebene Datei zu suchen und an dieser Stelle einzufügen. Dabei ist es nicht verboten, dass die dort angegebene Datei selbst wieder #include enthält, die Abarbeitung erfolgt also rekursiv.

In den früheren Compilerversionen gab es z.B. noch keine echten Konstanten, um dennoch nicht eine Größe an beliebig vielen Stellen im Programm ändern zu müssen, wurde dafür eine Präprozessordirektive verwendet:

```
#define PUFFER 512
```

Dann hat der Präprozessor jedes Auftreten des Wortes PUFFER durch die Zahl 512 ersetzt, einfach durch Textaustausch, ohne inhaltliche Kontrolle. Heute schreibt man einfach:

```
const int PUFFER = 512;
```

und wird dann auch gnadenlos mit Fehlern bombardiert, wenn man eine Konstante auf der linken Seite einer Zuweisung einsetzt.

Eine heute noch sehr oft genutzte Verwendung ist das Steuern des Übersetzungsvorgangs durch den Präprozessor. Es gibt nämlich auch die Direktive #undef, die eine vorher gemachte Definition aufhebt. Damit kann man dann gezielt Programmteile in Abhängigkeit von der Definition eines Wertes übersetzen lassen oder auch nicht:

Noch leichter geht das, wenn man dazu auch noch #if, #elif, #else und #endif benutzt. Damit kann man dann wunderschön Bedingungen formulieren und seinen Übersetzungsvorgang steuern. In der täglichen Praxis an der Hochschule wird Ihnen das eher selten passieren, aber in größeren Projekten sind solche Methoden gang und gäbe.

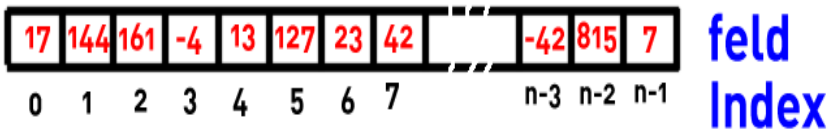
9 Zusammengesetzte Datentypen

Bisher haben wir uns mit den sogenannten elementaren Datentypen beschäftigt. Aber ähnlich wie man in der Chemie aus einzelnen Elementen völlig neue Moleküle zusammensetzen kann, kann man in der Informatik aus den elementaren Datentypen zusammengesetzte Datentypen bauen.

9.1 Array

Setzt man mehrere Daten vom gleichen Typ zusammen, erhält man ein Array. Der deutsche Name „Feld" ist derart unüblich, dass ich ihn auch hier nicht verwende (auch wenn ich sonst die deutschen Bezeichnungen bevorzuge).

Ein Array ist nichts anderes, wie eine Ansammlung einer Folge von Variablen des gleichen Typs unter einem gemeinsamen Namen. Der Zugriff auf die einzelne Variable darin wird über eine Positionsnummer,



den Index, vorgenommen. Die einzelnen Grundelemente werden also einfach durchnummeriert, ganz analog zu den Briefkästen im Studentenwohnheim, die mit der Appartementnummer versehen sind (über die zusätzlich vorhandenen Namen reden wir jetzt mal nicht, fragen Sie den Briefträger, nach welchem Kriterium er die Briefe einwirft).

Die Grafik zeigt ein Array mit den Namen *feld* mit n Elementen vom Typ Integer. Der Index ist in C++ nicht beliebig, sondern beginnt immer bei 0 für das erste Element und endet bei einem Array mit n Elementen bei (n-1). Dieser Fakt führt bei Programmieranfängern immer wieder für Verwirrung und zu Fehlern, weil der Index bei vielen Programmen entweder eins zu viel oder eins zu wenig beträgt. Das lässt sich aber im Normalfall schnell finden und klären.

Der Index wird in eckigen Klammern [ ] an den Namen der Variablen angehängt, das Array hat z.B. den Namen `feld` und soll 10 Elemente vom Typ `int` aufnehmen. Das wird dann so definiert: `int feld[10];` Der optische Unterschied ist also nur der Indexoperator bestehend aus den eckigen Klammern und dem Index dazwischen. Das erste Element im Array findet sich als `feld[0]` wieder, das letzte als `feld[9]`.

Als Index ist alles erlaubt, was man abzählen kann oder exakter, was einen definierten Vorgänger und Nachfolger hat. In C++ sind das alle ganzzahligen Datentypen wie `int`, `long`, `char` usw., nicht erlaubt sind `double`, `float` und Co.

Jedes Element des Arrays verhält sich genauso wie der zu Grunde liegende Datentyp und wird auch genauso verwendet und eingesetzt. Mit anderen Worten, überall da, wo (wie in diesem Beispiel) ein `int`-Wert stehen kann, darf auch ein einzelnes Element eines `int`-Arrays stehen:

```
feld[0] = 17;
feld[1] = 144;
feld[2] = feld[1] + feld[2];
feld[1] ++;
feld[i] = feld[k] - feld[0];
```

Der zusätzliche Nutzen entsteht erst dadurch, dass der Index natürlich auch eine Variable sein darf. Damit ist man dann in der Lage, den Speicherplatz eines Wertes (in gewissen Grenzen) selbst zu bestimmen. Und manche Aktionen wären ohne diese Zugriffsmethode überhaupt nicht möglich: versuchen Sie doch mal eine bunte Folge von Zahlen zu sortieren, wenn sie nicht auf jede dieser Zahlen einzeln zugreifen können. (Schreiben Sie die unsortierten Zahlen dazu einfach in einer langen Reihe auf einen Karton. Ohne Zerschneiden wird das mit dem Sortieren nichts werden.)

Wenn das mit den einzelnen Datenelementen so einfach funktioniert, kann man ja mal folgendes ausprobieren:

```
int feld1[10], feld2[10];
feld1[0] = 17;
feld1[1] = 144;
// usw. Das Feld komplett füllen
feld2 = feld1;  //feld1 in feld2 kopieren
```

Wir haben zwei Arrays `feld1` und `feld2`, die völlig identisch definiert sind. Dann sollte nach der letzten Anweisung in `feld2` das selbe drinstehen wie in `feld1`. Schön wär's. Der Compiler beschwert sich darüber, dass auf der linken Seite der Zuweisung ein „modifizierbarer Ausdruck“ stehen muss (manche Compiler erwarten einen L-Value, das ist aber auch nichts anderes) . Darunter versteht der Compiler eine Variable, die sich ändern darf, `feld2` ist das demnach nicht. (Der Compiler hat natürlich recht, wie immer. Der Name des Arrays ist ein Zeiger auf sein erstes Element und der darf nicht so einfach geändert werden, Details dazu finden sich im Kapitel 8.3 auf Seite 82 und im Kapitel 9.5).

Das Kopieren muss also anders gehen, und zwar auf die ganz schlichte Art: Element für Element. Und dabei hilft uns der beste Freund des Arrays, die `for`-Schleife. Die `for`-Schleife ist überhaupt ein sehr guter Freund (wenn nicht sogar mehr!) des Arrays, die beiden sind wie füreinander geschaffen. Unsere Kopieraktion läuft so am schnellsten:

```
const int MAXZAHL=10;  // Größe des Arrays
int feld1[MAXZAHL], feld2[MAXZAHL];

feld1[0] = 17;
feld1[1] = 144;  // usw. das Feld komplett füllen

for (int i = 0; i < MAXZAHL; i++) // Index von 0 bis MAXZAHL - 1
      feld2[i] = feld1[i];  //einzelnes Element kopieren
```

Die `for`-Schleife hilft auch, das Array per Tastatur zu füllen oder alle Elemente auf dem Bildschirm wieder auszugeben.

```
const int MAXZAHL=10;  // Größe des Arrays
int feld1[MAXZAHL];

for (int i = 0; i < MAXZAHL; i++) // Index von 0 bis MAXZAHL - 1
{
```

```
        cout << i+1 << ". Element eingeben: ";
        cin >> feld1[i];  //einzelnes Element von Tastatur holen
    }
    for (int i = 0; i < MAXZAHL; i++) // Index von 0 bis MAXZAHL - 1
    {
        cout << feld1[i] << "  ";  //einzelnes Element ausgeben
    }
    cout << endl;  // vor oder hinter die Klammer?
```

Die erste Schleife gibt den Text „1. Element eingeben:“ aus, dabei wird die Zahl aus der Schleifenvariable für die Ausgabe um 1 erhöht, der Index des ersten Elementes ist ja Null, der normal begabte Mensch zählt aber ab 1. Dahinter blinkt dann der Cursor und wartet auf die Eingabe des entsprechenden Elementes. Obwohl hier kein `endl` steht, landet die nächste Ausgabe in der nächsten Zeile! Daran ist die `Enter`-Taste Schuld, die ja zum Abschluss der Eingabe betätigt wurde und dabei auch den Cursor gleich mal in die nächste Zeile stellt. In den meisten Fällen ist das auch so gewollt...

Die zweite Schleife gibt die eingegebenen Werte wieder aus, diesmal ohne den dazu gehörigen Index. Hier gibt es jetzt auch ein `endl` und es ist verblüffend zu sehen, welchen Unterschied es macht, ob es vor oder (wie oben) hinter der geschweiften Klammer steht. Überlegen Sie erst und probieren Sie es dann aus.

Wenn man die (anfänglichen) Werte für das Array schon kennt, kann man diese bereits bei der Definition zuweisen:

```
int feld1[MAXZAHL] = { 7, 12, -23, 12*12, 5, 43, 2, -123, 10 };
```

Dieses Verfahren funktioniert allerdings nur bei der Definition, nicht im Laufe des Programms bei einer Zuweisung.

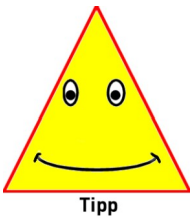
9.1.1 Mehrdimensionale Arrays

Der aufmerksame Leser erinnert sich bestimmt an den zweiten Satz im Abschnitt 9, der ganz allgemein aussagt, dass sich ein Array aus einer Anzahl von Elementen des gleichen Datentyps zusammensetzt. Kann man dann auch ein Array aus anderen Arrays zusammensetzen? Man kann! Und das ist sogar ein sehr häufiger Anwendungsfall, denken Sie mal an die Rechnung mit Matrizen in der Mathematik oder an eine einfache Tabelle. Aus dem eindimensionalen Array (man könnte hier auch von einem Vektor sprechen) wurde ein zweidimensionales Array (oder eben eine Matrix). In C++ bekommt die Variable dann einfach einen zweiten Index in eckigen Klammern, wer Lust hat und gerne den Überblick verliert, kann auch drei, vier, oder noch mehr Dimensionen schaffen. Mehr als vier Dimensionen brauchen nach meiner Erfahrung aber nur Physiker, die sind da vollkommen schmerzbefreit.

Bei zwei Dimensionen brauchen wir natürlich auch zwei `for`-Schleifen, bei drei Dimensionen drei, bei vier...

```
const int ZEILEN=10,SPALTEN=10;    // Größe des Arrays
    int feld[ZEILEN][SPALTEN];

    for (int z = 0; z < ZEILEN; z++) // z läuft über die Zeilen
    {
```

```
        for (int s = 0; s < SPALTEN; s++)  // s läuft über die Spalten
        {
              cout << "Element [" << z + 1 << "]["
                          << s + 1 << "] eingeben: ";
              cin >> feld[z][s];  //einzelnes Element von Tastatur holen
        }
  }
```



Bei zweidimensionalen Arrays sollte man die Laufindices nach ihrer Funktion benennen, also etwa ein z für die Zeile und ein s für die Spalte. Und das Einrücken und die Klammerung ist hier überlebenswichtig. Weniger wichtig ist, ob der erste Index die Spalte oder die Zeile darstellt, dass können Sie frei entscheiden, müssen es aber konsequent durchhalten. Technisch ist es wie im Listing gelöst, aber die interne Darstellung des Rechners ist nur in wenigen Fällen relevant.

9.1.2 C-Zeichenketten

Ein Sonderfall des Arrays entsteht, wenn als elementarer Datentyp `char` verwendet wird. Unbewusst haben wir mit unseren Textausgaben in den diversen Programmen immer schon lustig mit diesen *Zeichenketten* hantiert, ohne uns näher damit zu beschäftigen. Das holen wir jetzt nach.

In C++ gibt es zwei Abarten von Zeichenketten, die Klasse `string`, die Bestandteil von C++ ist, und die klassische C-Zeichenkette als Erbe der Vorgängersprache C, um die es hier jetzt geht.

```
char kette[80];
```

Diese Definition ist also nicht überraschend, sie stellt ein Array mit (in diesem Fall) bis zu 79 Plätzen für Zeichen dar. Die einzelnen Zeichen sind von 0 bis maximal 78 durchnummeriert. Stopp! 78? Was ist mit Element Nr. 79? Schreibfehler? Nein, hier schlägt der erste Unterschied zwischen einer Zeichenkette und einem „normalen" Array zu.

Zum einfacheren Umgang mit Zeichenketten fügt der Compiler ans Ende jeder Zeichenkette ganz von alleine eine \0 (also nicht die Ziffer Null, sondern den Zahlenwert Null!) als letztes Zeichen hinten an die Zeichenkette an. Diese \0 zeigt das Ende der Zeichenkette an und hilft bei der Verarbeitung. Denn so ist es möglich folgende Anweisungen zu schreiben:

```
cin >> kette;
cout << kette;
```

Mit den Werten eines „normalen" Arrays geht nämlich genau das nicht, wie wir auf Seite 99 gesehen haben! In diesem Fall kann man auf die `for`-Schleife für die Ein- und Ausgabe verzichten, die Zuweisung funktioniert aber nach wie vor nicht.

Hinweis: diese Form der Zeichenkette verwendet man ganz automatisch, wenn man in einem Programm ein paar Zeichen in doppelte Anführungszeichen einschließt, das ist eine klassische Konstante vom Typ `char[]`.

Für Zeichenketten gibt es in der Bibliothek `cstring` eine große Sammlung von fertigen Funktionen, die alle denkbaren Anwendungsfälle abdecken. Die Tabelle zeigt die wichtigsten:

`strcat (ziel, quelle)`	`quelle` wird an `ziel` angehängt
`strcpy (ziel, quelle)`	Kopiert `quelle` in `ziel` (ohne Überprüfung der Längen)
`strncpy (ziel, quelle, anzahl)`	kopiert `quelle` in `ziel` (mit Überprüfung der Größen)
`strlen (quelle)`	Ergibt die Länge der Zeichenkette
`strstr(quelle, such)`	Ergibt das erste Auftreten von `such` innerhalb von `quelle`, sonst 0
`atoi (kette)`	Wandelt `kette` in einen `int` um
`atol (kette)`	Wandelt `kette` in ein `long` um
`atof (kette)`	Wandelt `kette` in eine `double` um

9.2 String

C-Zeichenketten haben einen Nachteil, man muss zu Beginn des Programms genau festlegen, wie lang eine Zeichenkette werden darf und muss diesen maximalen Platz bei der Definition reservieren. Das ist zum einen Platzverschwendung, zum anderen aber unpraktisch. Man kann eine Zeichenkette im laufenden Programm nicht mal verlängern (über diese Grenze hinaus), wenn es notwendig wäre. Daher gibt es in C++ die Klasse `string`, die die Behandlung von Zeichenketten (ab hier spreche ich von *Strings* als Unterscheidung zu den klassischen C-Zeichenketten) noch viel einfacher macht. Und weil es sich bei den Strings um eine *Klasse* handelt, weicht die Schreibweise mancher Funktion ein wenig vom bisher gewohnten ab, daran gewöhnt man sich aber schnell. Spätestens im zweiten Semester geht einem diese objektorientierte Schreibweise in Fleisch und Blut über.

Um mit Strings arbeiten zu können, muss die Bibliothek `string` in das Projekt eingebunden werden. Danach kann man Variablen[21] vom Typ `string` anlegen:

```
#include <string>

// . . .

string name, vorname;
```

Es gibt hier keine Angabe der Länge, der erste Pluspunkt. Ein String passt sich nämlich der Größe des gespeicherten Textes dynamisch an. Damit hat die Platzverschwendung genauso ein Ende wie die Probleme beim Verketten (Aneinanderfügen) von Zeichenketten, ohne Längenbegrenzung kann man auch keine Länge überschreiten.

Man kann aber bei der Definition etwas angeben und damit einen Startwert ablegen. Und weil es ein Objekt einer Klasse gibt, kann man da aus unterschiedlichen Varianten wählen:

21 Streng genommen handelt es sich hier um ein **Objekt** der Klasse `string`, aber das soll uns hier nicht weiter belasten.

```
string ort("Darmstadt");     //objektorientiert
string ortsteil = "Eberstadt";     // klassische Zuweisung
string sterne(80, '*');            //Linie mit 80 Sternchen
```

Welche Variante man einsetzt, hängt vom persönlichen Geschmack ab. Man beachte aber die unterschiedliche Verwendung der Hochkomma: eine Folge von Zeichen wird in doppelte Hochkomma (üblicherweise als Anführungszeichen bezeichnet) eingeschlossen, ein einzelnes Zeichen in einfache Hochkomma.

Das Verketten von Strings geht dank Objektorientierung und Operatorüberladung[22] ganz einfach:

```
string adresse = "64297 " + ort + '-'  + ortsteil;
cout << adresse;  // -> 64297 Darmstadt-Eberstadt
```

Der Zugriff auf ein einzelnes Zeichen im String klappt ganz genauso wie bei den C-Zeichenketten:

```
ort[1] = 'u';    //klassisch
ort.at(2) = 'm';//objektorientiert
cout << ort;     // überlegen Sie selbst!
```

In seltenen Fällen (weil eine Funktion z.B. genau eine C-Zeichenkette erwartet) kommen Sie mit einem String nicht weiter, sondern brauchen eine Umwandlung in eine C-Zeichenkette. Dafür gibt es die Member-Funktion c_str(), in unserem Falle also etwa:

```
. . . = ort.c_str();
```

Auch für den String gibt es viele Funktionen, die einem das tägliche Leben sehr erleichtern. Da es sich aber um Objekte handelt, sind die Funktionen als Memberfunktionen realisiert, der Name der Funktion wird also durch einen einfachen Punkt verbunden, direkt an den Namen der Variablen (exakt: des Objektes) dran gehangen. In der folgenden Tabelle steht also der Bezeichner `Str` für einen konkreten String (also eine Variable vom Typ `string`).

`Str.clear()`	Löscht alle Zeichen des Strings
`Str.insert (p, string2)`	Fügt den String `string2` an der Position `p` in den `Str` ein
`Str.erase(p,n)`	Löscht `n` Zeichen ab Position `p`
`Str.replace(p, n, string2)`	Ersetzt `n` Zeichen ab der Position `p` durch den String `string2`
`Str.substr(p,n)`	Liefert den Teilstring von Str, der bei Position p beginnt und n Zeichen lang ist
`Str.length()` `Str.size()`	Liefern die Länge des Strings

Und dann sind da noch die Vergleiche, die man mit Strings ganz analog zu Zahlen anstellen kann, man kann sogar dieselben Operatoren verwenden, dank der Operatorüberladung von C++.

22 Die Ähnlichkeit zur Überladung von Funktionen ist nicht zufällig. Auch hier muss aus dem Zusammenhang und dem Umfeld klar sein, welche Bedeutung des Operators gerade gemeint ist

```
if (ortsteil < ort)
        cout << ort.length()
```

Ist Eberstadt kleiner als Darmstadt? Kommt drauf an, wen man fragt. In diesem Falle wird nämlich „Eberstadt“ mit „Darmstadt“ verglichen, mangels Informationen aus dem Einwohnermeldeamt nimmt der C++-Compiler einfach das Alphabet, und da ist bereits nach dem ersten Buchstaben klar, dass E größer ist wie D und damit ist dieser Vergleich `false`.

Auch alle anderen Vergleichsoperatoren `==, !=, >, <, >=` und `<=` arbeiten hier *lexikalisch*, also nach dem Alphabet. Sind dann Großbuchstaben auch größer als Kleinbuchstaben? Wenn Sie zum Nachschlagen in der ASCII-Tabelle zu faul sind, probieren Sie es aus.

9.2.1 Weitergehend: Umwandlung von Strings in Zahlen

Bei der Eingabe über die Tastatur erledigt die Funktion `cin` diesen Job (siehe Abschnitt 6.3). Kommen die Eingaben aber aus anderen Quellen (vor allem aus graphischen Benutzeroberflächen) muss man das als Programmierer selbst machen. Aber auch dafür gibt es eine Bibliothek, sie heißt `sstream` (für String-Stream). Ein Objekt dieser Klasse übernimmt quasi das, was bei direkter Eingabe über die Tastatur `cin` machen würde und wandelt einen String in eine Zahl um. Die Umwandlung läuft in zwei Stufen:

```
int zahl = 0 ;    //für das Ergebnis
string s = "12345"; // das soll nach Zahl
istringstream wandle_in_StreamObjekt(s);//string wird Stream-Objekt
wandle_in_StreamObjekt >> zahl;          //fertig
```

Der in Zeile 2 definierte String s wird dem Konstruktor eines `istringstream`-Objektes übergeben (mit dem sperrigen, aber hochinformativen Namen wandle_in_StreamObjekt), danach übernimmt der Operator `>>` die eigentliche Arbeit und speichert das umgewandelte Ergebnis in `zahl` ab. Fertig.

Für den umgekehrten Weg gibt es die Klasse `ostringstream`.

Wem dieser Weg zu steinig ist, der kann auf eine alte Funktion aus der C-Steinzeit zurückgreifen. Die Funktion atof(C-String) liefert zu einem C-String auch dessen numerische Entsprechung. „atof“ steht dabei für ASCII to Float, kann aber auch genauso gut für double-Werte genutzt werden. Der Nachteil: das Argument muss ein C-String, also eine klassische Zeichenkette sein, kein moderner String. Ein String string1 muss also erst in einen C-String umgewandelt werden, der dann mittels atof() in einen numerischen Wert umgewandelt werden kann:

```
string1 = "1.234567E+3";
num_wert = atof(string1.c_str());
```

Noch einfacher geht es mit einer Funktion direkt aus der string-Bibliothek (erst ab C++-Standard 11):

```
num_wert = stod(string1);
```

Ab sofort gibt es also keine Ausreden mehr, die Überprüfung von (numerischen) Eingaben zu unterlassen!

9.3 Strukturen (struct)

Nachdem wir jetzt Ansammlungen von Daten des gleichen Typs unter einem Dach ausführlich besprochen haben, kommen jetzt Daten aus verschiedenen Typen unter ein Dach. Bildlich kann man sich das wie eine Karteikarte vorstellen, auf der unterschiedliche Informationen abgelegt sind. In C++ nennt

man so etwas einen Variablenverbund oder `struct` (in manchen Büchern wird auch von einer *Struktur* gesprochen).

```
struct sStudent{
            string name;
            string vorname;
            unsigned int matrikelnummer;
            bool istBeurlaubt;
            int semester;
      };    //Semikolon hier nicht vergessen!
```

Unser `struct` trägt den Namen sStudent, das kleine s am Anfang soll verdeutlichen, dass es sich um ein `struct` handelt. Ein `struct` an sich ist eigentlich nur eine eigene Typdefinition, wenn man genau hinsieht erkennt man das auch an der Farbgebung (hier von VisualStudio), sStudent und string haben den selben Farbton (im Unterschied zu den elementaren Datentypen wie `int` oder `bool`). *Diese Farbgestaltung ist aber spezifisch für eine bestimmte Entwicklungsumgebung, außerdem kann jeder Anwender das so einstellen, wie er möchte. Es ist also nur ein dezenter Hinweis, der Unterschied wird auch hier nur in Listings sichtbar, nicht im Fließtext.*

Unser `struct` enthält fünf Datenfelder, die jedes für sich genommen nicht mehr unbekannt sein sollten. Wie üblich in C++ dürfen in einen `struct` alle Datentypen eingebaut werden, die man sich so vorstellen kann. Also auch ein `struct`. Oder ein Array. Und ganz analog zu den n-dimensionalen Arrays gilt auch hier: wer mehr als drei Schachtelungsebenen braucht, hat einen Denkfehler beim Design des Programms gemacht oder ist Physiker.

Der Compiler hat bis jetzt nur eine *Deklaration* vorgefunden (und auf seinen internen Notizzettel geschrieben), wir haben noch keine Variablen vom Typ sStudent erzeugt. Das holen wir jetzt nach:

```
sStudent studi;
sStudent zug[32];
```

Ich wiederhole mich gerne: man kann Datentypen aus beliebigen Datentypen zusammen setzen, ganz so wie man es braucht. Ein einzelner Student passt in die Variable `studi` hinein, ein ganzer Zug von 32 Studenten in das Array `zug[32]`.

Stellt sich die Frage, wie man einem Studenten ein Urlaubssemester verpasst oder seine Semesteranzahl ändert. Dafür gibt es den Punkt.Operator, der hier im Fließtext einfach untergeht, weil er so klein und unscheinbar ist. Aber wichtig:

```
studi.name = "Mueller";
studi.vorname = "Sascha";
// . . .
zug[4] = studi;         // Uppps! Oh!
zug[5].matrikelnummer = 765432;
```

Der Zugriff auf die Elemente einer `struct`-Variablen erfolgt also durch den Namen der Variablen, einen Punkt und dann den Namen des Elementes. Spannend ist die vorletzte Zeile, hier wird ein komplettes `struct` an ein anderes `struct` zugewiesen. Was bei Arrays und C-Zeichenketten eine Fehlermeldung gibt, funktioniert hier klaglos.

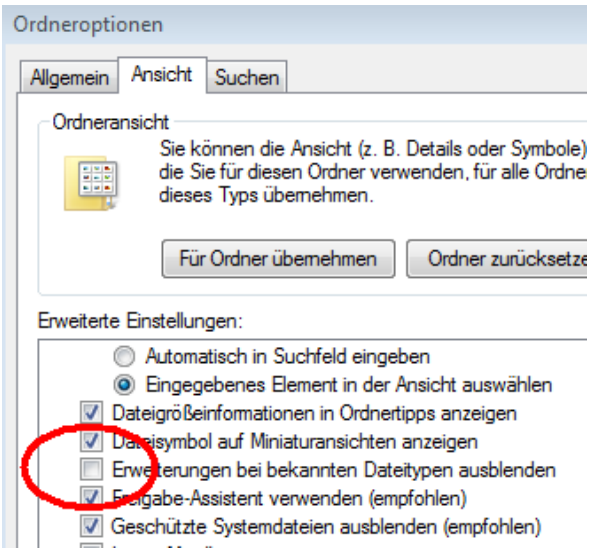
Aber: kein Licht ohne Schatten, obwohl ja `zug[4]` und `studi` identisch sind, kann man **nicht** mit dem Operator == auf Gleichheit prüfen.

9.4 Textdateien

Ein- und Ausgabe über Tastatur und Bildschirm beherrschen wir jetzt im Halbschlaf, aber trotzdem kommt der Zeitpunkt, an dem so richtig viel Daten in ein Programm hinein müssen. Solche Datenmengen fallen gerne bei Messreihen an, oft sogar außerhalb unseres Rechners. Die finden wir dann auf einem externen Datenträger in Form einer Datei wieder. In diesem Kapitel erfahren wir, wie man solche Dateien einliest (und verarbeitet) oder erzeugt.

Es gibt zwei Arten von Dateien, zum einen die Binärdateien, die aus einem Wust von Zeichen bestehen und für den Menschen unlesbar sind. Die ignorieren wir mal geflissentlich. Uns interessieren die Dateien, die die Daten in Textform ablegen und damit genauso strukturiert sind, wie unsere Quelltexte. Anders ausgedrückt: unsere Quelltext sind ein Beispiel für Textdateien. Messwerte bestehen meistens aus langen Zahlenkolonnen, die Mitgliederdatei eines Vereins vermutlich aus Zeilen, die alle relevanten Daten über das Mitglied enthalten. Allen gemeinsam ist, dass man diese Dateien als Mensch ohne weitere Hilfsmittel (vom Editor mal abgesehen) lesen und verstehen kann.

Die Verarbeitung von Dateien hängt zu einem gewissen Grade mit dem verwendeten Betriebssystem zusammen, wir betrachten hier aber nur die Funktionen, die auf allen gängigen Betriebssystemen gleich funktionieren. Eine Bedingung dafür ist, dass die zu verarbeitende Datei auf der lokalen Festplatte im selben Verzeichnis (Ordner) steht, in dem auch unsere selbst geschriebenen Quelltexte stehen[23]. Der Name der Datei ist frei wählbar, er sollte aber keine Umlaute, Leerzeichen und Sonderzeichen haben. Unter Windows ist besondere Vorsicht geboten, hier darf der Dateiname nur einen Punkt haben. Die Erweiterung (die drei Zeichen nach dem Punkt) unterschlägt Windows in der Grundeinstellung gerne, trotzdem ist diese Erweiterung Bestandteil des Namens. Abhilfe schafft es, entweder eine Erweiterung zu wählen, die Windows nicht kennt (z.B.xyx) oder in den Ordneroptionen die Option „Erweiterungen bei bekannten Datentypen ausblenden“ auszuschalten (das bietet dann sogar noch einen gewissen Schutz vor eingeschleusten Schadprogrammen).



Jede Textdatei kann mit der Entwicklungsumgebung geöffnet und bearbeitet werden (sie darf *nicht* Bestandteil des Projektes werden!), verwenden Sie dazu auf keinen Fall eine ausgewachsene Textverarbeitung wie Word, LibreOffice oder was auch immer. Danach ist ihre Textdatei auf keinen Fall mehr eine Textdatei. Auch der Windows-eigene Editor ist nur bedingt geeignet, weil er mit den unterschiedlichen Zeilenende-Markierungen der verschiedenen Betriebssysteme nicht immer klar kommt.

23 Vorsicht bei Computern, die einen angebissenen Apfel alsLogo tragen. Hier wird vom Betriebssystem gerne der Eindruck erweckt, dass eine Datei in einem bestimmten Ordner liegt. In Wahrheit liegt sie aber an völlig anderer Stelle, die für unser selbst geschriebenes Programm nicht erreichbar ist.

9.4.1 Öffnen, Schreiben, Schließen

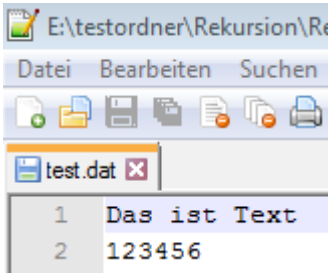
Dateien sind der Normalfall bei der Ein- und Ausgabe, `cin` und `cout` für Tastatur und Bildschirm sind zwei Sonderfälle, die sich nur dadurch unterscheiden, dass sie nach dem Einbinden von `iostream` einfach sofort funktionieren.

Die viel wichtigere Erkenntnis ist, dass alles, aber auch wirklich alles, was wir über `cin` und `cout` schon gelernt haben, ganz einfach weiter funktioniert. Das Konzept der `Stream`-Objekte unterscheidet nicht, ob der Eingabestrom von der Tastatur oder einer Textdatei stammt, genauso wenig wie es dem Ausgabestrom egal ist, ob er auf den Bildschirm oder in eine Textdatei schreibt.

Alles, was zusätzlich erforderlich ist, um mit Textdateien statt Tastatur und Bildschirm zu arbeiten, sind ein paar Zeilen Vorgeplänkel und eine Anweisung zum Abschluss:

```
string s = "Das ist Text";
int n = 123456;
fstream datei;    // Variable datei vom Typ fstream anlegen
datei.open("test.dat", ios::out);//Datei anlegen+zum Schreiben öffnen
datei << s;       //Text ausgeben
datei << endl;    //Neue Zeile beginnen
datei << n; //Zahl in Datei schreiben
datei.close(); // Datei schliessen
```

Zur besseren Verwirrung habe ich meiner Datei den Namen `datei` gegeben, das vermeidet Verwechselungen. In Zeile 1 wird ein Objekt vom Typ `fstream` angelegt und bekommt den sinnigen Namen `datei`. Flapsig dürfte man auch sagen, es ist eine Variable mit dem Namen `datei` vom Typ `fstream` angelegt worden. Dabei passiert aber sonst noch nicht so viel. Erst in Zeile 2 kommt Leben (also Inhalt) in das Objekt (die Variable), sie bekommt einen Namen (hier mal `test.dat`) und das Betriebssystem wird angewiesen, diese Datei zum Schreiben zu öffnen (`ios::out`). Existiert die Datei bereits, wird der Inhalt gelöscht, also die Datei geleert. Dabei wird keine Warnung oder Sicherheitsabfrage gebracht, das Betriebssystem kommt den Wünschen des Programms sofort und unmittelbar nach. Gab es die Datei bisher noch nicht, wird sie leer neu angelegt. Egal wie, nach dieser Zeile gibt es genau eine leere Datei mit dem Namen `test.dat` im gleichen Verzeichnis in de auch der Quelltextes steht.



Die nächsten drei Teile sind absolut selbsterklärend. Zeile 3 gibt eine Zeichenkette in die Datei aus, dann folgt ein Zeilenvorschub in Zeile 4 und in Zeile 5 folgt eine Zahl in die Datei.

Wenn Sie bei diesen drei Zeilen kein Déjà-vu haben blättern Sie sofort zurück zu Seite 44. Denken Sie sich statt des Wörtchens „`datei`“ einfach mal das Wörtchen „`cout`“ dort hin. Fällt der Groschen? (ach nee, das heißt ja jetzt 5 Cent). **Es gibt keinen Unterschied zwischen dem Schreiben in eine Datei und dem Schreiben auf den Bildschirm!!!** Das gilt auch für sämtliche Manipulatoren aus der Bibliothek `iomanip`.

Der guten Ordnung wegen machen wir die Datei in Zeile 6 dann noch zu, obwohl das der Compiler für uns gerne übernimmt und von sich aus an passender Stelle erledigt.

9.4.2 Lesen und Anfügen

Sie haben jetzt fünf Minuten Zeit, um sich zu überlegen, wie denn das gleiche Stück Programm aussehen müsste, um die eben geschriebene Datei wieder einzulesen. Sie müssen eigentlich nur alles, was ein Gegenteil hat, in sein Gegenteil umkehren:

```
string s;
int n;
datei.open("test.dat", ios::in); //Datei zum Lesen öffnen
datei >> s;      //Text in s ablegen
// datei >> endl; Nein, das nicht!!!
datei >> n; //Zahl aus Datei in speichern
datei.close(); // Datei schliessen
```

War doch nicht so schwer, oder? Wir müssen die Datei zum Lesen öffnen (`ios::in`), danach läuft es genau wie mit einem `cin` von der Tastatur. Damit wird auch klar, warum wir kein Zeilenendezeichen einlesen können, das funktioniert bei der Eingabe über Tastatur ja auch nicht.

Achtung: Beim Lesen aus einer Datei muss man peinlichst genau darauf achten, dass die gelesenen Daten auch zu den Variablen passen, in denen sie abgelegt werden sollen. Man kann zwar eine Ziffernfolge in einem String ablegen, aber keinen String in einer `int`-Variablen. Wenn man sich bei der Struktur einer Datei nicht zu 100% sicher ist, empfiehlt es sich, ein wenig Grips und Zeit in eine Einlese-Funktion zu stecken, die z.B. eine komplette Zeile aus der Datei in einen String einliest (das geht immer gut, solange die Zeile keine Leerzeichen enthält, dazu später mehr) und dann diesen String Stück für Stück mit den Funktionen aus Kapitel 9.2 auseinander nimmt. Ein direkt einsetzbares Beispiel finden Sie im Abschnitt 15.

Wenn wir die `open`-Anweisung ein klein wenig ändern, dass sie so aussieht

```
datei.open("test.dat", ios::app);
```

haben wir die eierlegende Dateimilchsau erschaffen, eine Datei, in die man schreiben kann, ohne dass sie vorher geleert wird. Diesen Modus nennt man Anhängen oder *Append.*

9.4.3 Wenn die Datei zickt

Bei der Arbeit mit Dateien kann es zu Problemen kommen, die außerhalb unserer selbst geschriebenen Programmzeilen liegen. Darauf muss ein Programm vorbereitet sein und entsprechend reagieren. So kann eine zum Lesen erwartete Datei schlicht nicht da sein, wo man sie sucht, der Dateiname stimmt nicht oder ähnliches. Existiert die Datei, kann sie kürzer wie erwartet sein, beim Schreiben kann die Festplatte voll sein oder die Schreibrechte im Ordner fehlen. Weitere Katastrophenszenarien bleiben der Fantasie des Lesers überlassen.

Für jedes dieser Probleme gibt es aber eine Lösung, bzw. eine Funktion, die uns da unterstützt. Fangen wir mit der Existenz der Datei an. Beim Schreiben ist das egal, da wird sie einfach angelegt (es sei denn, es fehlen die Schreibrechte im Verzeichnis). Beim Lesen sollte die Datei aber zumindest existieren. Das kann man mit der Memberfunktion `good()` überprüfen:

```
if ( ! datei.good()) cout << "Katastrophe!!!";  //Achtung, da ist ein !
```

Das identische Ergebnis erhält man auch als Wert des Objekts selbst:

```
if (datei)  //Achtung, da ist kein !
```

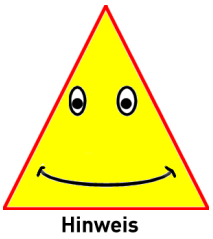
```
            // alles ok
      else
            // Katastrophe
```

Welche Variante man vorzieht, ist Geschmackssache, Hauptsache man überprüft die Existenz.

Die Funktion `good()` kann man übrigens nach jeder Dateioperation erneut abfragen und kann so erfahren, ob die letzte Operation gelungen ist. Aber vielleicht reicht es ja auch, wenn man sich darauf beschränkt, zu überprüfen, ob das Ende der Datei erreicht ist. Das sollte man auf jeden Fall tun, denn wer weiß schon, ob die versprochenen 20.000 Messwerte in der Datei stehen oder doch nur 19.999,

```
while (!datei.eof())
      {
            datei >> s;
            cout << s << endl;
      }
```

Diese Konstruktion stellt sicher, dass auf keinen Fall über das Ende der Datei hinaus gelesen wird, sollte also grundsätzlich angewendet werden.



Die Praktika und Übungen sind darauf ausgelegt, dass Sie den Umgang mit Textdateien erlernen, und nicht in erster Linie die Behandlung von möglichen Fehlern. Deswegen dürfen Sie davon ausgehen dass die zur Verfügung gestellten Textdateien „in Ordnung“ sind.

Das heißt aber nicht, dass Sie auf *jegliche* Fehlerbehandlung verzichten können, zumindest das ordnungsgemäße Öffnen muss genauso geprüft werden wie das Erreichen des Dateiendes.

9.5 Zeiger

Ohne Zeiger wäre C++ schlicht nicht machbar, auch wenn wir bisher (fast) ohne sie ausgekommen sind. Zeiger sind das Salz in der Suppe des Programmierens, es geht zwar lange Zeit auch ohne, aber vieles wird durch sie erst so richtig elegant.

Zeiger sind als schwierig und böse verschrien, dabei sind sie doch nur eine logische Weiterentwicklung des Index bei einem Array. Denn ein Index ist doch nichts weiter wie ein Zeiger auf ein einzelnes Element dieses Arrays.

Die Idee hinter den Zeigern ist es, nicht mehr mit den Daten im Speicher direkt zu arbeiten, sondern mit der Information, wo diese Daten liegen. Auf den ersten Blick ist das ja ein erhöhter Aufwand, aber nur dann, wenn es sich um sehr wenige Daten handelt. So geht es um Zehnerpotenzen schneller, wenn man beim Sortieren nicht die Daten durch den Speicher hin und her schaufelt, sondern einfach nur die Information über den Speicherplatz. Das wird umso effektiver, je größer die Datenblöcke werden. Es ist doch auch viel einfacher, statt der Bücher im Regal nur die Karteikarten mit der Signatur (also dem Standort im Regal) zu sortieren. Falls Sie das nicht glauben arbeiten Sie mal vier Wochen in der Zentralbibliothek und stellen die zurückgegebenen Bücher wieder an ihren Platz. Danach werden Sie Zeiger lieben.

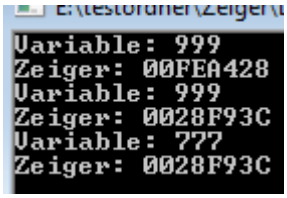
```
int i = 999;      // int-Variable
int *ip;          // Zeiger auf einen int-Wert


cout << "Variable: " << i << endl;
cout << "Zeiger: " << ip << endl;


ip = &i;    //ip wird mit der Adresse von i belegt
cout << "Variable: " << i << endl;
cout << "Zeiger: " << ip << endl;


*ip = 777;  // An der Zielposition von ip wird die 777 angelegt
cout << "Variable: " << i << endl;
cout << "Zeiger: " << ip << endl;
```

In diesem Programm definieren wir zunächst eine `int`-Variable i und initialisieren sie mit dem Wert 999. Nichts Neues, schon tausend Mal gemacht.

In Zeile 2 kommt das Neue: ein Zeiger auf einen `int`-Wert. Das `p` im Variablennamen deutet darauf hin (von engl. pointer), der Compiler erkennt das aber am Stern. In der Variablen `ip` steht aktuell noch kein vernünftiger Wert drin, da sie ja noch nicht initialisiert wurde. Zur Kontrolle geben wir die Inhalte beider Variablen mal aus (Achtung, je nach Compiler erzeugt Zeile 4 einen Fehler oder eine Warnung, weil die Variable `p` noch nicht initialisiert wurde).

Dem helfen wir jetzt ab, indem wir der Variablen ip einen Wert zuweisen. Als Zeiger auf einen `int`-Wert passen hier auch nur die Adressen (also der Speicherort) von `int`-Variablen hin. &i ist genau das, nämlich die Adresse der Variablen `i`. Im Ergebnis hat sich der Inhalt der Zeiger-Variablen `ip` geändert.

Zeile 8 ändert jetzt den Inhalt der Variablen `i`, nutzt dazu aber die Informationen aus der Variablen `ip`, also dem Zeiger.



Wir können so also Werte ändern, ohne die dazu angelegte Variable zu nutzen. Klingt auf den ersten Blick nicht so spannend, aber wenn wir die Variable dazu eigentlich nicht brauchen, kann man dann womöglich auf sie verzichten? Ja, man kann, aber erst später.

Kehren wir zunächst einmal zurück zu den Arrays. Das Grundlegende dazu haben wir ja bereits ab Seite 97 erfahren, Sollten Sie diesen Abschnitt bisher ausgespart haben, ist jetzt die Gelegenheit die Wissenslücken zu schließen.

9.5.1 Arrays und Zeiger

Im Kapitel 9.1 habe ich es schon einmal angedeutet, der Name eines Arrays ist ein Zeiger auf sein ersten Element, also die Start-Adresse des Arrays. Betrachten wir mal folgendes Programmstück:

```
int feld[10] = { 6, 7, 8, 9, 10, 1, 2, 3, 4, 5 };
int index = 0;
int *fp;


fp = feld;  //Startadresse in Zeigervariable
```

```
cout << *fp << "  " << fp << endl;
cout << feld[index] << "  " << &feld[index] << endl;

fp++;
index++;

cout << *fp << "  " << fp << endl;
cout << feld[index] << "  " << &feld[index] << endl;
```

In Zeile 1 definieren wir ein Array `feld` aus `int`-Werten und füllen es auch gleich auf. Danach definieren wir eine Variable `index`, die wir für den Index des Feldes verwenden möchten und setzen diese auf den Wert 0 (also das erste Element mit demInhalt 6). Zeile 3 deklariert einen Zeiger auf einen `int`-Wert, genauso wie im Listing zuvor.

Die erste Anweisung (Zeile 4) weist dem Zeiger die Startadresse des Feldes zu, damit zeigt diese Zeiger-Variable auf das erste Element des Arrays. Damit ist die Ausgabe in den Zeilen 5 und 6 keine Überraschung, beide geben den Wert 6 und die dazu gehörige Adresse aus

Interessant ist die nächste Zeile. Da wird die Zeigervariable um 1 erhöht. Und auch der Index wird um 1 erhöht. Das Ergebnis sieht zur Hälfte aber anders aus als erwartet, es wird zwar das nächste Element aus dem Array angezeigt, beim Erhöhen von Index auch logisch. Aber warum auch bei Verwendung des Zeigers? Der müsste dich streng genommen auf die nächste Speicherzelle zeigen, er zeigt aber auf eine viel weiter hinten liegende Adresse, vier Byte weiter. Diese vier Byte sind genau die Größe eines `int`-Wertes, der Zeiger hat also eigenmächtig die Anweisung des Programmierers missachtet und sich statt um 1 um 4 erhöht. Und es wird noch dreister, diese Erhöhung wird immer anhand des verwendeten Datentyps bestimmt, so dass der Zeiger immer auf das nächste Element und nicht auf die nächste Speicherzelle zeigt. Eigentlich eine geniale Idee des Compilers, oder?

9.5.2 Zeigerarithmetik

Wir können also mit dem Zeiger genau die gleichen Effekte erzielen wie mit dem Index. Zeiger um 1 erhöhen bedeutet nicht, auf die physisch nächste Adresse zu zeigen, sondern auf das nächste Element im Array. Dabei bestimmt der Compiler anhand des Datentyps automatisch, wie groß die einzelnen Elemente sind und passt den Offset entsprechend an. Probieren Sie es aus, indem das Programm von oben mit `double`, `char` und `long` als zugrunde liegenden Datentyp laufen lassen.

Apropos Datentyp `char`: ein Array aus `char` war doch der ideale Datentyp für die C-Zeichenketten, und in der Tat operieren die meisten Funktionen mit Zeichenketten im Hintergrund mit Zeigern. Die Funktion `strcpy(quelle, ziel)` kopiert mittels dieser Anweisungen

```
char *quelle;
char *ziel;
while (*quelle)
      *ziel++ = *quelle++;
```

```
*ziel = 0;
```

eine Zeichenkette von `quelle` nach `ziel` und fügt die charakteristische `Null` hinten an. Kürzer und effizienter geht es vermutlich nicht.

Konsequenterweise kann man mit dem Namen des Arrays auch rechnen: `*(feld + 2);` ist identisch mit `feld[2];` und in unserem Falle zu `*(fp + 2);`

(Die Klammern werden benötigt, weil der *-Operator eine höhere Priorität wie das +-Zeichen hat).

Wir halten fest: Ein Zeiger bzw. eine Zeiger-Variable bietet einen anderen, zusätzlichen Zugriff auf die Inhalte von Variablen. In der Regel sind Zugriffe per Zeiger schneller (das wirkt sich bei unseren Programmen aber noch nicht wirklich aus).

9.5.3 Dynamischer Speicher

Wir benutzen die Variablendeklaration im Rahmen unserer Zeigeroperationen eigentlich nur noch, um den Speicherplatz für den Wert zu organisieren, danach greifen wir nur noch per Zeiger auf diese Werte zu. Da wäre es doch nur konsequent, auf die Variable zu verzichten und sich den Speicher auf andere Weise zu besorgen.

Das würde uns auch vom Zwang, den Speicher bereits beim Starten des Programms anzufordern ein Stück befreien, denn dann könnten wir den Speicher ganz nach Bedarf besorgen und wenn wir ihn nicht mehr brauchen auch wieder an das Betriebssystem zurück zu geben.

Diesen Vorgang nennt man dynamische Speicherverwaltung, eine sehr effiziente und schnelle Methode zur Datenhaltung.

Um sich Speicher für einen Wert zu besorgen wird die Funktion new verwendet.

```
double *doublepointer = new double;
```

Hier passiert folgendes: auf der linken Seite der Zuweisung wird eine Zeigervariable `doublepointer` angelegt, die auf einen `double`-Wert zeigen kann. Auf der rechten Seite der Zuweisung wird mittels `new double` vom Betriebssystem genau die Menge Speicher angefordert, die für einen `double`-Wert erforderlich ist. Der Rückgabewert der Funktion new ist die Startadresse dieses Speicherbereichs. Und durch die Zuweisung landet diese Startadresse in der Zeiger-Variable `doublepointer`.

Statt über eine Variable haben wir den Speicher jetzt quasi *anonym* beim Betriebssystem bekommen, die einzige Verbindung zu diesem Speicher besteht ab jetzt nur noch in der Zeigervariablen. Dieser Vorteil ist gleichzeitig der größte Nachteil: verliert die Zeigervariable durch äußere Umstände (also Programmierfehler) den Kontakt zu ihrem Speicherbereich gibt es keine Chance mehr, an die dort abgelegten Daten zu kommen. Und weil das noch nicht schlimm genug ist, können wir dem Betriebssystem diesen Speicher auch nie zurück geben, wir wissen ja nicht mehr, wo er liegt. In Einzelfällen ist das nicht so schlimm, aber wenn sich das häuft, führen diese *Speicherlecks* zu akutem Speichermangel und irgendwann stürzt unser Programm oder sogar der ganze Rechner dann sang- und klanglos ab. **Also: Vorsicht beim Ändern von Zeiger-Variablen, einmal verlorener Speicher kann nie wieder gefunden werden.**

Wir sind aber ordentliche Programmierer und beobachten unsere Zeiger-Variablen sehr genau, daher sind wir in der Lage, nach Gebrauch den angeforderten Speicher auch wieder frei zu geben. Das erledigt die Funktion

```
delete doublepointer;
```

für uns.

9.5.4 Dynamische Arrays

Eigentlich ganz normal, was für elementare Datentypen funktioniert, sollte auch für zusammengesetzte Datentypen funktionieren. Tut es ja auch, wir zeigen das mal an einem Array:

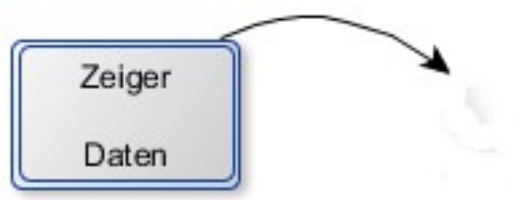
```
int *feld = new int[10];
*(feld) = 7;
feld[2] = 8;
cout << *(feld + 2) << endl;
delete[] feld;
```

Die diversen Schreibweisen für den Zugriff auf die Feldelemente kennen wir ja schon von Seite 110, das schockt keinen mehr. Wichtig ist die Änderung bei `delete[]`, ohne die eckigen Klammern gibt es leider keine Fehlermeldung, aber das Freigeben funktioniert nicht richtig (jeder Compiler macht hier etwas anderes, aber keiner macht das, was gemeint ist).

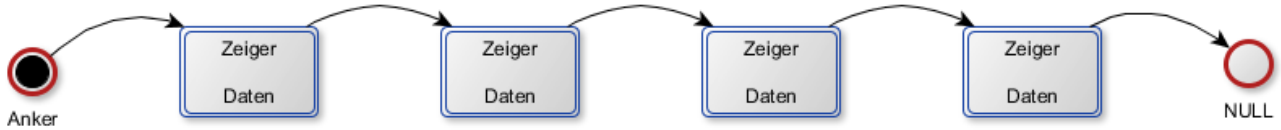
Auf diese Art und Weise sind wir endlich in der Lage, die Arrays in der genau benötigten Größe anzulegen und nicht einfach mal pauschal riesige Felder zu belegen,die dann zu 99% unbenutzt auf dem Stack liegen. Oder das reservierte Feld ist mal wieder zu klein, weil der Datenlogger doch mehr Messwerte gesammelt hat wie angenommen. [24]

9.5.5 Verkettete Listen

Jetzt treiben wir es auf die Spitze. Wir holen uns dynamisch Platz für ein `struct`, das neben den bisher schon bekannten Datenfeldern auch einen Zeiger auf eben dieses `struct` (also auf diesen Typ, nicht auf das konkrete `struct`) enthält. Dazu muss der Compiler einmal über seinen Schatten springen, denn hier haben wir wieder mal so ein Henne-Ei-Problem: entweder der Zeiger oder das `struct` ist bei seiner ersten Verwendung noch nicht bekannt. Das ist der einzig mir bekannte Fall, in dem der Compiler mal von seinen sonstigen Prinzipien abweicht und etwas Toleranz walten lässt.



Wir haben also ein `struct`, bestehend aus beliebigen Datenfeldern (hier Daten genannt) und einem Zeiger, der zunächst mal ins Nirgendwo zeigt, aber auf ein solches Element zeigen könnte. Daraus kann man jetzt mit etwas Phantasie eine Kette basteln:

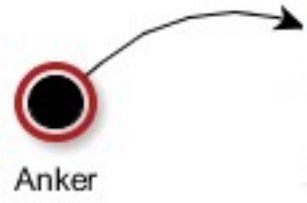


Los geht es mit einem *Anker*, das ist nichts anderes als ebenfalls ein Zeiger auf dieses `struct`, nur eben ohne Datenfelder dran. An diesem Anker hängt unsere ganze Kette, ohne den Anker kämen wir nicht mehr an die Datenfelder heran. Also: besondere Vorsicht beim Ändern dieser Zeigervariablen!

Das letzte Element zeigt definiert ins Leere, das ist aus alter Zeit die Adresse 0, seit C++11 gibt es dafür den `nullptr`.

24Und weil das eines der häufigsten Probleme darstellt, gibt es dafür eine eigene Klasse mit den Namen vector, die alle diese Probleme (und noch ein paar andere) löst.

Wie funktioniert das Ganze nun? Wir beginnen also ganz harmlos mit einem Zeiger auf dieses `struct`, das wird unser Anker. Der zeigt noch nirgendwo hin, kein Wunder, es ist ja auch noch nichts da, worauf er zeigen könnte. Dieser Zeiger ist die einzige Variable in unserer Liste, die einen Namen hat (von temporären Hilfszeigern abgesehen) und damit die einzig dauerhafte Verbindung zu unseren Daten.

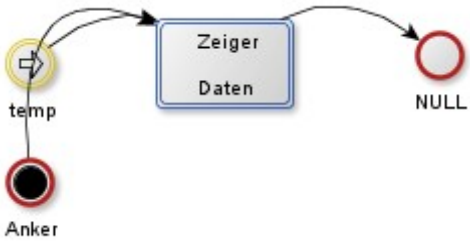


Deswegen brauchen wir als nächste ein Element für unsere Liste, das besorgen wir uns mit `new` und weisen dem Zeiger auf dieses neue Element einer temporären Variablen `temp` (einen Zeiger auf `struct`) zu, damit wir an das Element dran kommen.

Damit das neue Element nicht auch irgendwohin in die Pampas zeigt, wird sein Zeiger auf `NULL` (oder `nullptr`) gesetzt, das zeigt dann in Zukunft das Ende unserer Liste an.

Sinn und Zweck einer verketteten Liste ist es ja, Daten in einer bestimmten Form zu speichern. Im logischen Ablauf des Einfügens ist jetzt der Zeitpunkt, an dem die Daten irgendwie in den Datenfeldern abgelegt werden müssen. Das kann durch eine einfache Eingabe über die Tastatur aber z.B. auch durch das Auslesen aus einer Datei erfolgen. Aus Gründen der Übersichtlichkeit lassen wir aber hier jetzt alles, was mit den Daten im `struct` zu tun hat und beschränken uns auf die Abläufe in der Liste selbst.

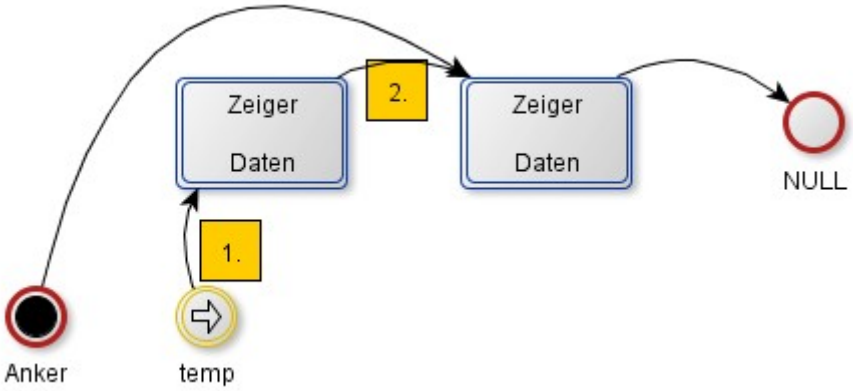
Das sieht ja schon aus, wie eine Liste mit einem Element, allerdings hängt sie noch nicht am Anker, sondern an einem temporären Zeiger. Das erledigen wir, indem wir dem Anker den Inhalt von `temp` zuweisen, und schon zeigen sogar gleich zwei Zeiger auf unser neues Element.



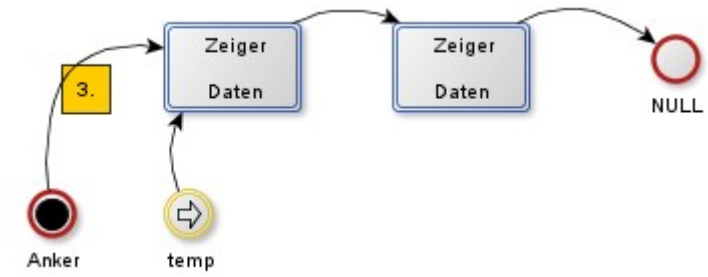
Damit wäre das erste Element richtig in der Liste angekommen.

Wie geht es weiter? Jedes weitere Element wird am Anfang der Liste eingefügt, also zwischen dem Anker und dem aktuell führenden Element. Natürlich könnte man auch am Ende einfügen oder irgendwo in der Mitte, das können Sie sich dann als Übung überlegen. (Mehr zum Thema Anfang und Ende einer Liste folgt im Abschnitt 9.5.6)

Um ein weiteres Element einzufügen, müssen wir uns natürlich erst ein Element besorgen. Das klappt am besten mittels `new`, wie sonst. Und wir hängen das auch wieder an unseren temporären Zeiger `temp` auf, damit es uns nicht durch die Lappen geht (1.). Im 2. Schritt setzen wir den Zeiger in dem neuen `struct` auf den Wert, den der Anker aktuell hat. Sowohl der Anker als auch das

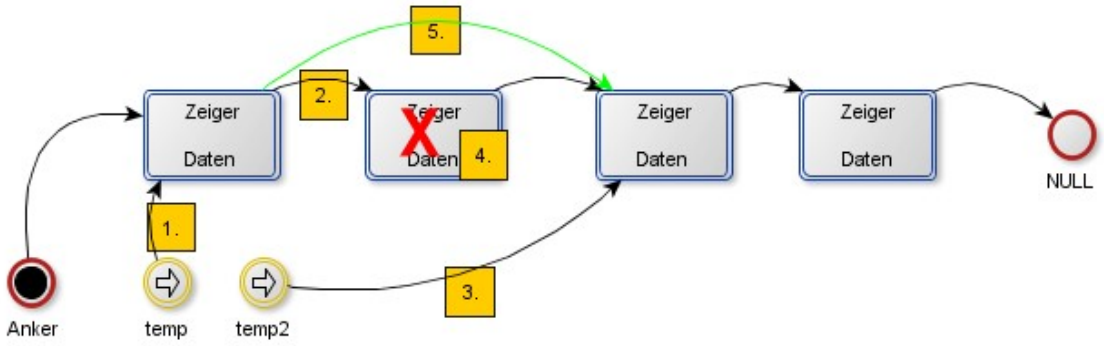


neue Element zeigen also beide auf das *„alte, erste“* Element. Deshalb muss in der letzten Operation (3.) noch der Anker auf das neue Element gesetzt werden, mit anderen Worten, der Anker erhält wieder den Wert von der temporären Variablen `temp`.



Dieses Spielchen wiederholt sich für jedes weitere Element der Liste. Wir haben die perfekte Liste kreiert! Aber damit sind wir ja noch nicht fertig, wir brauchen ja auch noch Verfahren, um Elemente aus der Liste zu entfernen (löschen). Dazu ist es hilfreich, wenn wir uns einmal ansehen, wie man am besten von einem Element der Liste zum nächsten kommt, sich also durch die Liste hangelt.

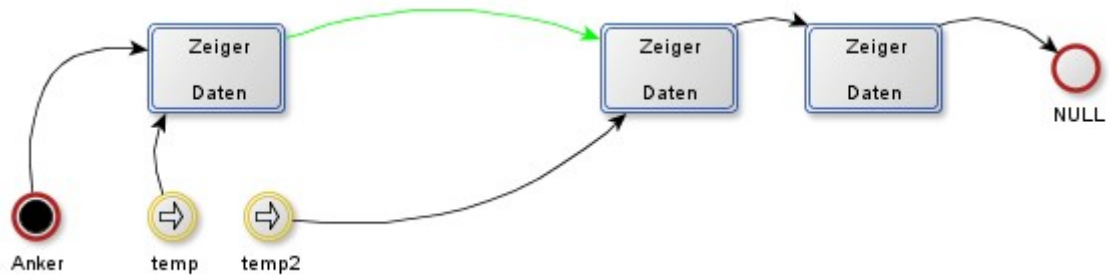
Man beginnt beim Anker, wo denn sonst. Und setzt einen Hilfszeiger (unser `temp` hat ja gerade nichts zu tun, also nehmen wir den) auf das erste Element (wegen der Verallgemeinerung betrachten wir hier den Fall, ein Element mitten aus der Liste zu entfernen, also nicht das erst oder das letzte, da ist es ein wenig einfacher. Ist also eine Fingerübung für Sie). Um an das nächste Element zu kommen, gibt es sogar einen extra Operator, den Pfeil-Operator `->` er besteht aus einem – und dem > (direkt aneinander gesetzt, also `->`. Mit `Anker->Zeiger` landen wir also auf dem Element, das als zweites in der Kette liegt[2.]. Jetzt prüfen wir, ob wir das zu löschende Element gefunden haben (z.B. durch den Vergleich eines der Datenfelder mit einem bestimmten Wert, aber das sparen wir uns genauso ein wie die Eingabe dieser Datenfelder).



Ist es das gesuchte (also zu löschende Element) gefunden, müssen wir uns zwei Zeiger merken: einen Zeiger, der auf das Element zeigt, welches dem zu löschenden[25] folgt[3.]. Und einen, der auf das Element davor zeigt[1.].

Zum Löschen geben wir zuerst mittels `delete` den Speicher frei, auf den das Vorgängerelement zeigt[2.]. Damit ist unser unerwünschtes Element schon mal aus dem Speicher verschwunden und belegt dort keinen Platz mehr. Im zweiten Schritt muss jetzt das Vorgängerelement so geändert werden, dass es auf den (ehemaligen) Nachfolger des gelöschten Elementes zeigt[5.]. Dessen Adresse haben wir ja im ersten Zeiger gespeichert. Und schon sind wir fertig!

25 In der Praxis lässt man den ersten Zeiger auf das Element hinter das zu löschende zeigen und schleppt den zweiten Zeiger immer ein Element hinterher. Dieser zeigt dann automatisch auf das zu löschende Element.



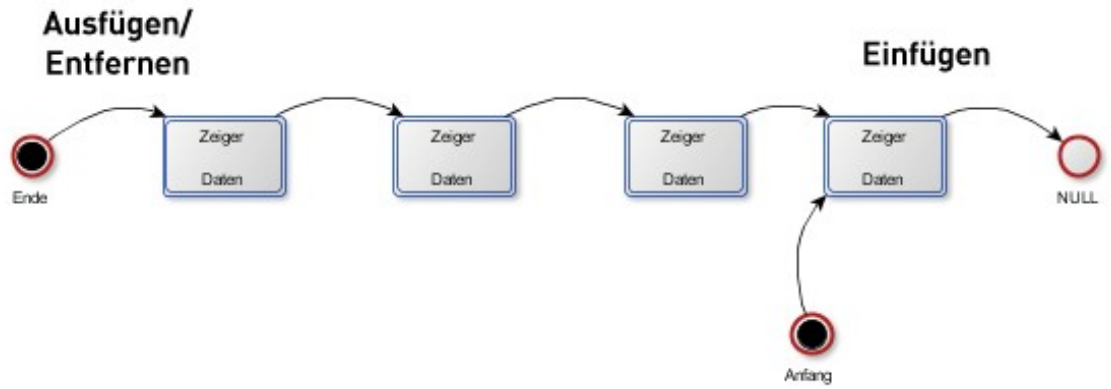
Nicht erwähnt wurde hier das Einfügen eines Elementes in der Mitte der Liste, das ist zum Beispiel hilfreich, um eine sortierte Liste zu erhalten. Dann wird das Einfügen an Anfang oder Ende der Liste zum Sonderfall. Die Lösung dafür bleibt dem geneigten Leser überlassen.

Analog gilt das auch für das endgültige Aufräumen einer Liste. Speicher, den man als Programmierer mittels `new` anfordert, muss man auch mit `delete` wieder freigeben, bevor man das Programm beendet. Auch diese Aufgabe stellt eine nette Fingerübung dar.

9.5.6 Die verschiedenen Typen einer Liste

Für den einfachen Umgang mit Listen ist etwas Denkarbeit im Vorfeld nötig. Es gibt nämlich zwei grundsätzlich unterschiedliche „Bauformen" für Listen. Und je nach Art dieser Liste kann man sich das Leben als Programmierer leichter oder auch komplizierter machen.

First-In First-Out (FIFO)
Das Problem kennen Sie vom Einkauf im Discounter. Vor der Kasse bildet sich eine Schlange, wer als Erster in der Schlange stand darf auch als Erster bezahlen und seine Einkäufe nach Hause tragen. In der Theorie der Informatik nennt man das eine Warteschlange. Dummerweise hat die Praxis damit ein Problem, denn wo ist bei einer Warteschlange der Anfang und wo ist das Ende? Verwenden Sie im Ernstfall einen Moment Zeit darauf, sich diese Frage zu stellen und für Ihr konkretes Problem auch zu lösen.



Im folgenden Beispiel endet die Warteschlange an der Kasse und sie beginnt mit dem Anstellen. Für die Liste bedeutet dies, dass neue Daten am Anfang (und nur dort) eingefügt werden und am Ende gelöscht werden. Optisch sieht das merkwürdig aus, weil der Anfang der Liste am rechten Bildrand liegt.

Auf den ersten Blick scheint das Einfügen am falschen Ende zu stehen, durch den zusätzlichen Zeiger Anfang wird es aber genauso leicht wie das Entfernen eines Listenelementes.

Last-In First-Out (LIFO)

Das ist der berühmte Stapel (oder Stack), der im Laufe dieses Skriptes ja schon öfter eine Rolle spielte. Hier wird das Element, welches als letztes auf dem Stapel abgelegt wurde, als erstes wieder herunter genommen. Das kann der Stapel mit dem Schmierpapier sein, der regelmäßig aufgefüllt wird. Das Blatt, das den Stapel einmal gegründet hat, muss sehr lange darauf warten, dass es einmal benutzt wird. Dagegen hat das oberste Blatt nur eine kurze Verweildauer auf dem Stapel. Im Supermarkt bilden die ungenutzten Einkaufswagen eine solche Liste, der Wagen, der als letzter abgestellt wurde, wird als erster wieder verwendet.

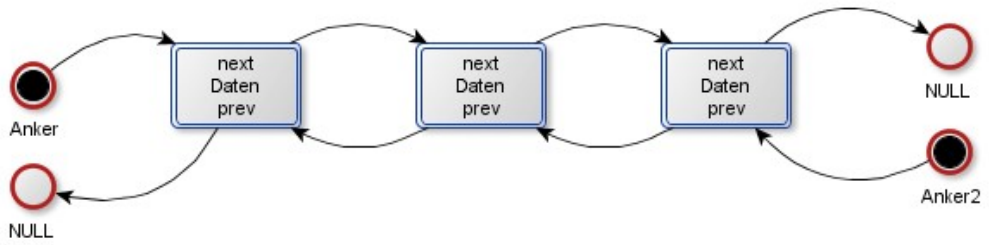
Hier liegt der Fall einfach, die Liste hat nur ein Ende, das auch gleichzeitig ihr Anfang ist. Da muss man nicht lange überlegen.

Mischformen
Natürlich kommt es immer wieder vor, dass diese beiden Grundformen abgewandelt auftreten. Wer vor der Kasse wartet wird doch gelegentlich von eiligen Kunden mal gefragt, ob er/sie nicht schnell mal dazwischen darf. Und aus dem großen Stapel Papier muss es manchmal eben das aus der Mitte sein, weil es eine bestimmte Farbe hat. Dann wählt man die Listenform, die die häufigsten Aktionen abdeckt.

9.5.7 Vorzugsrichtung und mehrfach verkettete Listen

Eine verkettete Liste hat eine Vorzugsrichtung, das ist die Richtung, in der die Elemente aneinander gereiht sind. In den Grafiken der vorangegangenen Abschnitte ist das immer von links nach rechts. Man kann auch sagen, es ist sehr viel einfacher von einem Element zu seinem rechten Nachfolger zu gelangen als zum links gelegenen Vorgänger. Eigentlich ist es sogar überhaupt nicht möglich, auf direktem Wege zum Vorgänger zu gelangen.

Das war aber bei den beiden Listentypen FIFO und LIFO auch nicht erforderlich. Wird diese Funktionalität benötigt, ergänzt man seine Listenelemente um einen weiteren Zeiger, der jeweils auf den Vorgänger zeigt Das macht das Ein- und Ausfügen/Löschen minimal aufwendiger, dafür aber das Durchlaufen in beide Richtungen einfacher.



Eine solche, doppelt verkettete Liste, hat daher zwei Anker, einen für die Vorwärtsverzeigerung (Verkettung über den Zeiger `next` der einzelnen Datenelemente) und einen Anker2 für die Rückwärtsverzeigerung (Verkettung über den Zeiger `prev` der einzelnen Datenelemente). Der jeweils letzte Zeiger zeigt auf `nullptr`.

9.5.8 Grundsätzliches Vorgehen bei Listen

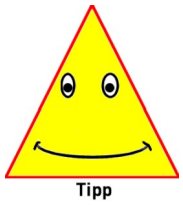
Unabhängig von der der Art der verwendeten Liste gibt es ein generelles Vorgehen bei der Verarbeitung von Listen. Was auch immer Sie mit einem Listenelement bzw. der Liste vorhaben, achten Sie auf folgende Szenarien:[26]

Erstes Element
Beim Einfügen bedarf das erste Element der Liste einer besonderen Beachtung. Bei einer leeren Liste zeigen (hoffentlich!) alle Zeiger auf `nullptr`, das macht das Einfügen zum Spezialfall. Dieser Spezialfall ist aber noch einen Hauch einfacher wie das Einfügen des x-ten Elements.

Letztes Element
Hier passiert das Gegenteil, beim Entfernen des letztes Elements muss man ebenso darauf achten, dass hinterher wieder alle Zeiger auf `nullptr` zeigen und der Speicher rechtzeitig frei gegeben wurde.

Leere Liste
Dieser Sonderfall wird beim Durchsuchen einer Liste gerne übersehen. Abhilfe schafft auch hier eine gute Skizze mit den Vorüberlegungen. Und manchmal hilft es auch, erst zu prüfen, dann zu handeln.

Liste durchlaufen
Das ist eigentlich trivial, wenn man die drei Szenarien im Hinterkopf behält. Man benötigt dafür nur einen Hilfszeiger, der in der Vorzugsrichtung der Liste von einem zum nächsten Element weiter gesetzt wird. So kommt man automatisch durch die gesamte Liste und kann beliebige Operationen (z.B. die Ausgabe der Datenfelder) mit den Listenelementen vornehmen.



Da eine Liste ja in der Regel eine unbekannte Anzahl von Elementen enthält und das Ende der Liste sehr plötzlich kommen kann, empfiehlt sich zum Durchlaufen eine kopfgesteuerte Schleife. Aufgrund der unbekannten Größe scheidet die `for`-Schleife aus. Bleibt einzig die `while`-Schleife übrig. Die passt aber bestens für diese Aufgabe, weil sie vor ihrem Block die Bedingung „Liste zu Ende?“ abprüfen kann und nur dann den Zugriff auf die Daten eines Listenelements erlaubt, wenn die Liste nicht bereits zu Ende ist.

9.5.9 Verkettete Liste zum Selberbasteln

Das Abarbeiten von verketteten Listen bereitet im Praktikum zunächst einmal Probleme, weil man das Verhalten so schlecht sichtbar machen kann. Und kleine Fehler fatale Folgen haben können. Da hilft eine verkettete Liste aus Pappe, Schnur, Magneten, doppelseitiges Klebeband und einer magnetischen Tafel (alternativ Pinnwandnadeln und eine Korkwand).

Als Listenelement dienen Pappkarten, an die ein Stück Schnur geknotet wird. Das freie Ende der Schnur befestigt man am Magneten (oder der Pinnwandnadel). Zusätzlich benötigt man noch ein oder zwei Zeiger (`Anker`, `temp`), also ein Stück Pappe mit doppelseitigem Klebeband auf der Rückseite, dafür ohne Datenfeld, aber auch mit Schnur und Magneten dran. Diese Zeiger halten auch ohne Magnet auf der Tafel, weil sie ja unter einem Namen ansprechbar sind.

26 Im folgenden wird im Text der Begriff `nullptr` genutzt. `NULL` hat nur in den Grafiken überlebt und sollte heute nicht mehr verwendet werden.

Jetzt kann man jede Operation auf ein Listenelement Schritt für Schritt nachvollziehen. Ein mit new erzeugtes Datenelement muss mit dem Magneten von einem Zeiger auf der Tafel gehalten werden, sonst sind die Daten verloren (das sieht man dann sofort!). Natürlich dürfen gleichzeitig mehrere Zeiger auf ein Element zeigen, aber niemals keiner.

Dann kann man sich sehr leicht und anschaulich klar machen, wie man Elemente aus der Liste entfernt, in die Liste aufnimmt oder wie man von einem zum nächsten Element kommt.

10 Weitere Sprachelemente und ausgewählte Bibliotheksfunktionen

10.1 Template-Funktionen

Wir haben ja bereits auf Seite 84 Funktionen überladen, also inhaltlich gleiche Abläufe unter demselben Namen, aber mit unterschiedlichen Parametern, geschrieben. Wenn man jetzt aber den immer gleichen Ablauf nur für unterschiedliche Daten**typen** benötigt, landet man bei den Template-Funktionen.

Ein Template ist eine Vorlage, aus der dann immer das aktuell benötigte Objekt automatisch erzeugt wird. Ähnlich funktionieren auch die Dokumentenvorlagen einer Textverarbeitung.

Die Bestimmung, welches von zwei Datenelementen das größere ist, läuft doch immer gleich ab, egal, welcher Datentyp zu Grunde liegt. Für Daten vom Typ `long` ergäbe sich folgende Funktion:

```
long max(long a, long b)
{
      if (a > b) return a; else return b;
}
```

Und für ein Zeichen:

```
char max(char a, char b)
{
      if (a > b) return a; else return b;
}
```

Der Unterschied ist jetzt nicht wirklich groß, das Ersetzen von `long` durch `char` keine große Kunst. Also kann man so etwas auch dem Compiler überlassen, der macht dabei wenigstens keine Tippfehler.

```
template <typename T>
T max(T a, T b)
{
      if (a > b) return a; else return b;
}
```

Statt dem Datentyp `long` oder `char` steht da jetzt einfach ein T. Und damit der Compiler weiß, dass dieses T stellvertretend für alle denkbaren Datentypen steht, schreiben wir noch eine Zeile davor, die ihm genau das erklärt. Fertig.

Übrigens: Dass für den Template-Typ immer der Buchstabe T verwendet wird, ist nur der Faulheit der Informatiker zu verdanken. Natürlich kann man statt dem T jeden beliebigen Bezeichner hinschreiben, das hat sich aber einfach nicht durchgesetzt. Nahezu alle Bücher und vor allem nahezu alle Programmierer schreiben nur ein T hin.

Darüber hinaus kann man auch bei Templates alles machen, was mit den bisher bekannten Datentypen als Parameter auch möglich ist, z.B. die Übergabe als Referenz:

```
template <typename T>
void tausche(T &a, T &b)
{
        T temp;
        temp = a;
        a = b;
        b = temp;
}
```

Erhöhen wir die Anforderungen noch ein wenig weiter: wir übergeben zwei Parameter mit unterschiedlichen Typen. Dann muss man keine Verrenkungen machen, sondern einfach zwei Typenamen definieren:

```
template <typename T1, typename T2>
```

Und das alles kann man wiederum auch untereinander kombinieren, z.B. bei Arrays:

```
template <typename T, int max>
```

Probieren Sie es einfach aus! Es geht vieles leichter und schneller von der Hand. Der Compiler selbst macht es vor, eine ganze Sammlung von Bibliotheksfunktionen (die wir in diesem Rahmen nicht einmal erwähnt haben) basiert auf Templates, daher auch der Name Standard Template Library (STL). Einen kleinen Einblick gibt das Kapitel *Über den STeLlerrand geblickt* auf Seite 182.

10.2 Namensräume

Die Entwickler von C++ denken ja manchmal weiter wie der Dozent in einer Vorlesung, vor allem denken sie daran, dass mit ihrem Compiler auch große und sehr große Projekte gestemmt werden, an denen viele Programmierer beteiligt sind. Da lassen sich dann Namenskonflikte einfach nur sehr schwer vermeiden, irgendwann benennen zwei Programmierer mal etwas gleich. Damit wäre das Projekt dann plötzlich nicht mehr funktionsfähig.

Das lässt sich einfach vermeiden, in dem man seine Funktionen und Variablen in einem eigenen Namensraum (`namespace`) zusammenfasst, dann muss jeder Programmierer nur noch seinen Namensraum eindeutig bezeichnen und schon hat man ein Problem weniger.

Innerhalb eines Namensraums ändert sich für den Programmierer nichts, er kann schalten und walten wie gewohnt. Spannend wird es erst, wenn er eine Funktion oder Variable aus einem anderen Namensraum verwenden will. Dann muss zur Unterscheidung der Namensraum, gefolgt von zwei Doppelpunkten vor den Namen der Funktion (bzw. Variablen) gestellt werden.

Denn auch der Compiler hält sich an seine eigenen Regeln und hat „seine“ Funktionen und Variablen in einen eigenen Namensraum gepackt, den Namensraum std. Als Anwender haben wir dann die oben beschriebenen Möglichkeiten, entweder schreiben wir vor jede aus std verwendete Funktion/Variable ein std:: oder wir wenden den Trick mit der Anweisung

```
using namespace std;
```

an, dann können wir uns das sparen.

Das klappt natürlich nur, wenn man sich auf die Verwendung eines Namensraums beschränkt, wenn auf zwei oder mehr Namensräume zugegriffen werden muss (was bei größeren Projekten zwangsläufig der Fall ist), kommt man nicht umhin, den Namensraum anzugeben.

Und so wird ein Namensraum definiert:

```
namespace myplace
{
	int max(int a, int b)
	{
		//...
	}
}
```

Ist doch nicht so schwer? Die Funktion max(a,b,) muss dann bei ihrer Verwendung als

```
myplace::max(6, 7);
```

angegeben werden.

10.3 try und catch

Egal, wie aufwändig Sie die Überprüfung von Fehlern im Programm gestalten, es wird immer einen Anwender geben, der einen Fehler produziert, der Ihnen nicht einmal im Traum eingefallen ist. Da wäre es doch geschickter, wenn man eine Gruppe von Anweisungen zusammen packen könnte und bei irgendeinem Fehler in dieser Gruppe gibt es eine mehr oder weniger allgemeine Fehlerbehandlung.

Gibt es natürlich, der kritische Block wird mit einem `try { }` geklammert, danach folgt ein Block `catch(...){ }`, der sich um die Fehlerbehandlung kümmert. Ein kritischer Block liegt oft bei der Behandlung und Verarbeitung von Dateien vor, da kann verdammt viel schief gehen. Also packt man das gesamte Dateihandling in einen try-Block. Geht alles gut, wird der catch-Block ignoriert, wenn nicht liefert er die notwendige Fehlerbehandlung (und wenn es nur die Ausgabe einer Fehlermeldung ist).

Diese Vorgehen ergibt sehr viel besser lesbare, aber auch besser wartbare Programme, weil der Code nicht durch umfangreiche Fehlerbehandlungen völlig undurchsichtig wird (die Sache mit dem Wald und den Bäumen).

Ach so: das hier ist eine sehr verkürzte und vereinfachte Darstellung der Abläufe und ist mehr als Hinweis zu verstehen, sich bei Bedarf aus anderen Quellen die weitergehenden Informationen zu besorgen.

10.4 Ausgewählte Bibliotheksfunktionen

Jeder Compiler liefert eine riesige Zahl von Bibliotheken mit, die auch irgendwo beschrieben sind. Für die gängigen Funktionen gibt es hier eine Übersicht.

10.4.1 Mathematische Standardfunktionen <math.h>

Um die mathematischen Funktionen nutzen zu können muss die Bibliothek <math.h> eingebunden werden. Alle Funktionen liefern ein Ergebnis vom Datentyp double zurück, bei Winkeln wird mit dem *Bogenmaß* gerechnet!

Trigonometrische Funktionen:

Funktionsdeklaration	Mathematische Bedeutung
`double acos (double)`	arcus cosinus
`double asin (double)`	arcus sinus
`double atan (double)`	arcus tangens
`double atan2 (double, double)`	arcus tangens (zwei Variablen)
`double cos (double)`	cosinus
`double cosh (double)`	cosinus hyperbolicus
`double sin (double)`	sinus
`double sinh (double)`	sinus hyperbolicus
`double tan (double)`	tangens
`double tanh(double)`	tangens hyperbolicus

Exponentialfunktionen:

Funktionsdeklaration	Mathematische Bedeutung
`double exp (double a)`	e^a (mit e= 2,718281828...)
`double pow (double a, double b)`	allgemeine Exponentialfunktion a^b
`double sqrt (double a)`	Quadratwurzel √ a
`double log (double a)`	Logarithmus von a zur Basis e (natürlicher Logarithmus *ln*)
`double log10(double a)`	Logarithmus von a zur Basis 10 (dekadischer Logarithmus)

Sonstige Funktionen:

Funktionsdeklaration	Mathematische Bedeutung
`double fabs (double a)`	Absolutwert von a |a|
`int abs (int)`	Absolutwert |a| (Bibliothek <cstdlib>
`double ceil (double)`	Liefert die nächsthöhere ganze Zahl

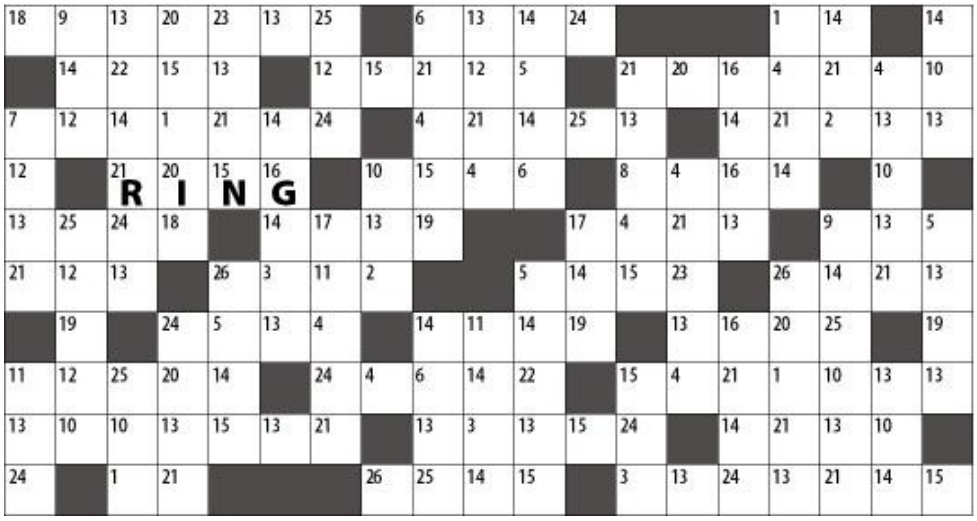
10.4.2 Einlese-Varianten

Um Eingaben von der Tastatur ohne abschließendes Return einlesen zu können (z.B. für die Steuerung von Spielen), gibt es die Funktion `_getch()` aus der Bibliothek `conio.h`

Funktionsdeklaration	Bedeutung
`int _kbhit()`	Liefert 0, falls keine Taste gedrückt wurde, sonst einen Wert !=0
`int _getch()`	Liefert den (Tastatur-)Code des Zeichens, nicht zwingend den ASCII-Code Sondertasten wie `F1`, `F2`, `Einfg`, `Entf` usw. liefern zwei Zeichen, zuerst eine 0 oder die 224, danach den eigentlichen Zeichencode
`cin.get(char*)`	Liest einzelne Zeichen aus dem Eingabestrom, gepuffert (keine Tastencodes) Bibliothek `<iostream>`
`cin.getline (char*s, int n)` `cin-getline (char *s, int n , char delim)`	Liest eine komplette Zeile aus dem Eingabestrom bis zur Länge `n` oder bis zum Begrenzungszeichen `delim` Bibliothek `<iostream>`

11 Fallstudie Zahlenrätsel

Vielleicht haben Sie in der Bahnhofsbuchhandlung auch schon mal ein Rätselheft in der Hand gehabt und dabei ein Kreuzworträtsel entdeckt, das keinerlei Erläuterungen zum Ausfüllen enthält, dafür aber in jedem freien Feld eine Zahl stehen hat. Die einzige Erklärung lautet „Gleiche Zahlen bedeuten gleiche Buchstaben", dazu gibt es noch eine durchnummerierte Reihe der verwendeten Zahlen, in der zu einigen wenigen Zahlen der richtige Buchstabe angegeben wird, damit das Rätsel nicht unlösbar[27] bleibt. Und wenn der Autor besonders nett war, hat er diese Buchstaben zusätzlich im Rätsel einmal eingetragen. (Bildquelle: Wikimedia, Zahlenrätsel.jpg von Michael Joachim Lucke).



Um ein solches Rätsel zu lösen versucht man jetzt, ausgehend von den bekannten Buchstaben, weitere Buchstaben zu ermitteln, Dabei hilft die Häufigkeit des Auftretens und ein wenig Grips. Und unser Programm. Das soll nämlich den aktuellen Denk-Zustand anzeigen, indem es das Rätselfeld blitzschnell ausfüllt und falsche Annahmen auch genauso schnell wieder aus unserem Blickfeld entfernt. Es ersetzt also nur das Rätselheft, Bleistift und Radiergummi, nicht die gedankliche Leistung. (Es steht aber jedem Leser frei, das Programm dahingehend zu erweitern).

Diese Aufgabe entspricht in ihrem Schwierigkeitsgrad etwa der 4. Übung im ersten Informatiksemester (wenn man von 6 Übungen im Semester ausgeht), damit Sie dieses Lösung auch nachvollziehen können, wenn die Vorlesung erst am Anfang steht. Das Herangehen an die Lösung können Sie als Blaupause für ihre eigenen Übungen verwenden, die gemachten Annahmen und Vereinfachungen liegen im Rahmen dessen, was auch in einer Übung erlaubt und gefordert ist.

11.1 Aufgabenstellung

Gesucht ist ein Programm, das bei der Lösung eines Zahlenrätsels unterstützt. Es soll den aktuellen Zustand des Rätsels (also alle bekannten Buchstabe-Zahl-Kombinationen) anzeigen, in freien Feldern wird die Zahl angezeigt. „Schwarze" Felder werden durch ein selbst zu wählendes Sonderzeichen dargestellt.

Welche Zahl in welches Feld gehört wird durch eine Textdatei festgelegt, die einfach zeilenweise die entsprechenden Zahlen enthält. Schwarze Felder werden durch eine Null symbolisiert.

Das Programm zeigt als zentrales Element das Zahlenrätsel in seinem aktuellen Zustand an, wenn zu einer Zahl der Buchstabe bekannt ist, wird der Buchstabe ausgegeben, ansonsten die Zahl.

27 Der Lösungsvektor: 1=D, 2=M, 3=V, 4=O, 5=H, 6=B, 7=Q, 8=Y, 9=W, 10=S, 11=J, 12=U, 13=E, 14=A, 15=N, 16=G, 17=P, 18=Z, 19=X, 20=I, 21=R, 22=K, 23=F, 24=T, 25=L, 26=C

Darunter steht der aktuelle Lösungsvektor (also die aktuelle Zuordnung von Zahlen und Buchstaben). Als Erweiterung kann zu jeder Zahl die Häufigkeit des Vorkommens im Rätsel ausgegeben werden (das haben wir bei unserer Lösung schon eingebaut).

Der Benutzer hat danach die Möglichkeit, eine Zahl und den seiner Meinung nach dazu passenden Buchstaben einzugeben. Diese Änderung wird dann im Rätsel vorgenommen und der Durchlauf beginnt von neuem, bis allen Zahlen ein Buchstabe zugeordnet wurde und kein Buchstabe doppelt vergeben wurde.

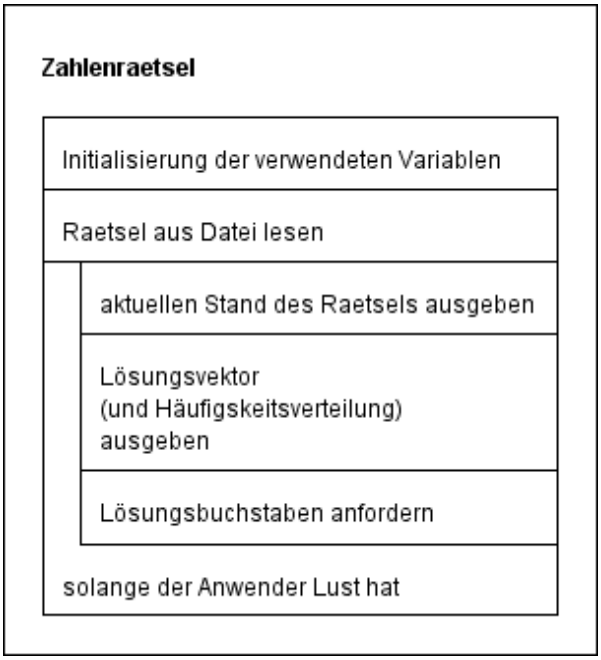
Damit ist das Rätsel gelöst.

Erweiterungen: Anstelle der Eingabe von Zahl und Buchstaben kann ein Menü treten, das zusätzlich die Möglichkeit bietet, den aktuellen Lösungsvektor in einer Datei abzuspeichern bzw.. wieder aus einer Datei zu lesen. In der Praxis hat sich auch eine Funktion zum Vertauschen zweier Buchstaben als hilfreich erwiesen.

11.2 Stufe 1: Grobablaufplan

Der grobe Ablauf ist aus der Aufgabenstellung gut zu entnehmen:

- Initialisierungen
- Ausgabe des Rätselfeldes
- Ausgabe des Lösungsvektors
- Verarbeitung der Eingabe



11.3 Stufe 2: Datenstrukturen und Designfragen

Das Rätselfeld basiert in jedem Fall auf einem zweidimensionalen Array. Als Basisdatentyp kommt entweder ein `int` in Frage (nämlich dann, wenn man nur die Zahl abspeichert, die im Rätsel an der entsprechenden Position steht) oder ein `struct`, das auch den dazugehörigen Buchstaben enthält (weitere Datenfelder sind denkbar, wir haben die erste Variante gewählt).

Um das Rätsel auf einem Bildschirm mit 80 Spalten vernünftig darstellen zu können, werden pro Kästchen drei Spalten benötigt (zweistellige Zahl plus ein Leerzeichen), also wird es ab einem Rätsel mit mehr als 26 Spalten unübersichtlich. Zufällig hat auch der Lösungsvektor diese Länge, so dass man die horizontale Größe hier entsprechend beschränken sollte.

In vertikaler Richtung wäre ja viel mehr Platz, weil man den Bildschirm ja scrollen kann. Aber auch hier sollte man bei 24 Zeilen enden, damit man eben nicht mehr scrollen muss (Zum Anzeigen des Lösungsvektors muss man dann aber eine Taste drücken).

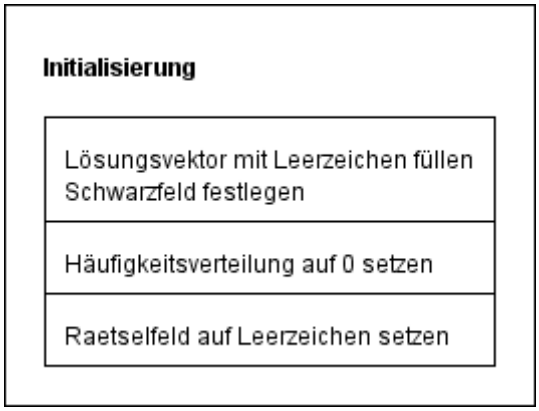
Für den Lösungsvektor braucht man (wie der Name ja schon sagt) ein eindimensionales Array vom Datentyp `char`. Und für die Häufigkeitsverteilung eines vom Datentyp `int`. Diese haben beide eine Länge von 26 oder 27 Elementen, weil es nur 26 verschiedene Buchstaben gibt (wenn man die schwarzen Felder auch wie Buchstaben behandelt sind es maximal 27).

Der gesamte Ablauf wird in einer Schleife (fußgesteuert) wiederholt, bis alle Buchstaben gefunden wurden und der Lösungsvektor vollständig ist. Da dies nicht so ganz einfach festzustellen ist, machen wir da eine Vereinfachung.

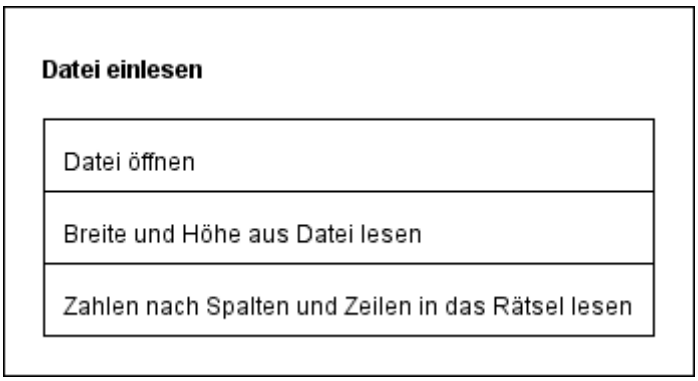
11.4 Stufe 3: Funktionen

Die einzelnen Funktionen, die das Grundgerüst bilden, tauchten bereits in den vorherigen Stufen auf:

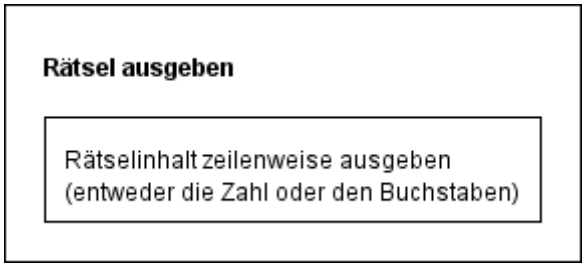
- Initialisierung: hier werden das Rätselfeld und die diversen Vektoren initialisiert, d.h. die Zahlen auf 0 gesetzt und die Buchstaben auf das Leerzeichen



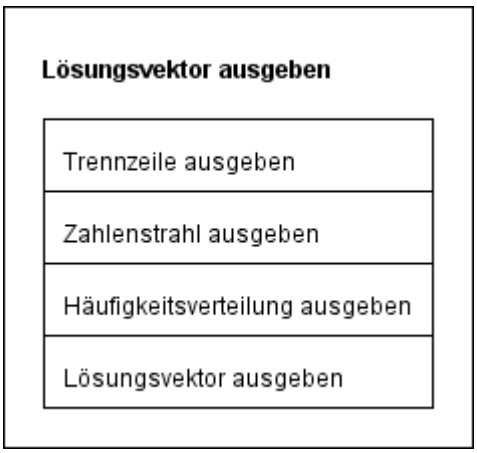
- Einlesen: liest die Zahlen des Rätselfeldes aus der Datei aus und füllt das Rätselfeld, Breite und Höhe sowie die Zahl der unterschiedlichen Buchstaben werden ermittelt



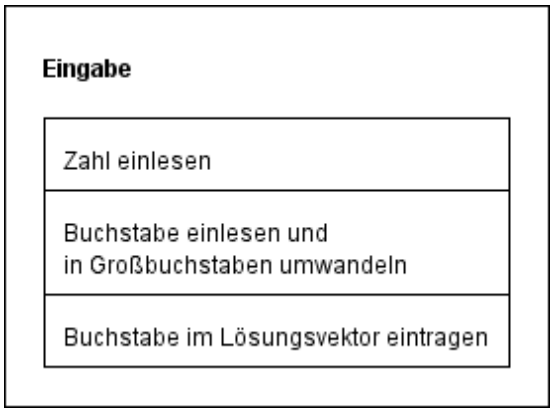
- Anzeige Rätselfeld: zeigt das Rätselfeld in seinem aktuellen Zustand an



- Anzeige Lösungsvektor: gibt den aktuellen Lösungsvektor aus



- Eingabe: liest eine Zahl und einen Buchstaben vom Benutzer ein, prüft auf Fehleingaben (falsche Zahl, Buchstabe schon vorhanden...)



- Update: aktualisiert das Rätselfeld, falls nötig (hängt vom gewählten Datenmodell ab, ist in unserem Fall nicht erforderlich)

11.5 Stufe 4: Detaillösung

Auf den ersten Blick sieht der ganze Ablauf doch sehr linear aus. Wie so oft steckt der Teufel aber im Detail. Wir haben zwar jetzt für alle Funktionen und das Hauptprogramm eine Idee, aber wissen immer noch nicht, wie die Funktionen ganz konkret ablaufen. Und über die Parameter, die eine Funktion benötigt und/oder ihre Rückgabewerte haben wir uns noch keinerlei Gedanken gemacht. Das passiert jetzt.

11.5.1 Benötigte Datentypen und Datenstrukturen

Um die Funktionen im Detail zu entwickeln, müssen wir wissen, mit welchen Datentypen und -strukturen wir es zu tun haben. Aus der Aufgabenstellung und unseren Vorüberlegungen können wir bereits ein paar Randbedingungen entnehmen, die durch die Größe des Bildschirms und den Umfang des deutschen Alphabets vorgegeben sind.

Alle diese Datentypen und Variablen sind global und werden damit außerhalb jeglicher Funktionen (also auch außerhalb der `main`()-Funktion) definiert[28].

Das deutsche Alphabet hat bekanntlich 26 Buchstaben (ohne Umlaute), dazu kommt ein Sonderzeichen für das Schwarzfeld. Macht 27. Unser Lösungsvektor benötigt deshalb eine maximale Länge von 27 Elementen. Auch der Vektor der Häufigkeitsverteilung hat maximal 27 Elemente (die Häufigkeit des Schwarzfeldes ist zwar nicht so interessant, wird aber wie ein normaler Buchstabe behandelt). Damit haben wir die erste Konstante gefunden:

```
const int MAXLETTERS = 27;  //Max. Anzahl unterschiedlicher Buchstaben
```

Wegen der begrenzten Größe des Bildschirms haben wir uns bereits zu Beginn auf eine maximale Breite von 25 Feldern und eine maximale Höhe von ebenfalls 25 Zeilen festgelegt. Auch das gießen wir in zwei Konstanten (die haben zufällig den gleichen Wert, aber unterschiedliche Bedeutung, daher zwei Konstanten):

```
const int MAXX = 25;          //max. Felder pro Zeile
const int MAXY = 25;          //max. Anzahl der Zeilen
```

Für das Rätselfeld gibt es mehrere Ansätze, wir entscheiden uns hier für die Abbildung durch ein zweidimensionales Array. In den einzelnen Elementen wird die Zahl des gesuchten Buchstaben abgelegt, nicht der Buchstabe selbst. Damit müssen wir dieses Feld niemals ändern, weil es nur als Bestückungsvorschrift eingesetzt wird. Die Verknüpfung zwischen Zahl und Buchstaben erledigen wir über den Lösungsvektor, daher betrachten wir diese beiden Dinge hier gemeinsam.

Wir benötigen also ein Rätselfeld, dass in der Lage ist, MAXX * MAXY Elemente vom Typ Integer aufzunehmen. Diese Festlegung sagt nichts über die tatsächlich verwendete Größe aus, solange sie unterhalb dieser Grenzen liegt.

```
int raetsel [MAXX] [MAXY];          // Das Raetselfeld
```

Durch die Benutzung der Konstanten wird auch sofort klar, dass der erste Index die Spalte (von links nach rechts) bedeutet und der zweite Index die Zeile (von oben nach unten).

Passend dazu benötigen wir einen Lösungsvektor, der zu einer Zahl einen Buchstaben liefert. Dieser muss also vom Typ Character sein:

```
char loesungsvector [MAXLETTERS];
```

28 Der aufmerksame Leser wird sich jetzt wundern. Im Kapitel 8 über Strukturierte Programmierung wurde vor der Benutzung globaler Variablen gewarnt. Hier werden sie jetzt trotzdem verwendet, um die Parameterliste nicht zu sehr aufzublähen. In den Übungen sollten Sie das aber nicht nachmachen.

Die Länge des Lösungsvektor hängt von der verwendeten Sprache ab, in Griechenland wäre er etwas kürzer, in China ein klein wenig länger. Auch hier gilt: das ist die maximale Länge, im Rätsel können durchaus weniger als 26 verschiedene Buchstaben vorkommen, aber auf keinen Fall mehr.

Für die Speicherung der Häufigkeitsverteilung benötigen wir einen gleich großen Vektor, der die Anzahl des Auftretens jedes einzelnen Buchstabens enthält. Der hat die selbe Länge wie der Lösungsvektor, aber den Datentyp Integer, weil er ja Zahlen aufnehmen soll:

```
int haeufigkeit [MAXLETTERS];
```

Damit sind wir fast durch. Es fehlen uns einzig noch zwei Variablen, die die tatsächliche Größe des Rätselfeldes angeben, die ja kleiner oder gleich der Konstanten MAXX und MAXY sein müssen:

```
int Breite, Hoehe;     // tatsaechliche Groesse des Raetsel
```

Zu diesem Zeitpunkt müsste das alles sein, was man für die Lösung der Aufgabe braucht. Das schließt aber spätere Ergänzungen nicht aus.

Und mit diesen Informationen können wir jetzt die erste Funktion angehen, die unsere frisch kreierten Daten mit sinnvollen Startwerten füllt.

11.5.2 Initialisieren

Diese Funktion soll alle Arrays und Vektoren mit sinnvollen Startwerten füllen. Das sind der **Lösungsvektor**, die **Häufigkeitsverteilung** und das **Rätselfeld** an sich. Der Lösungsvektor wird für alle Elemente auf das Zeichen 32 (Leerzeichen) gesetzt, die Häufigkeit für jede Zahl auf Null. Die Rätselfelder sollen als Startwert ein Schwarzfeld erhalten (dafür verwenden wir das @-Zeichen mit dem ASCII-Wert 64).

Um einen Vektor zu initialisieren, d.h. alle Elemente einmal zu durchlaufen, benötigt man sinnvollerweise eine `for`-Schleife. Macht hier also je eine Schleife für den Lösungsvektor und die Häufigkeitsverteilung. Es klappt aber auch mit einer einzigen Schleife!

Optimierungen sind eine feine Sache, wenn das Programm bereits funktioniert und man nur noch an Details Hand anlegen muss. In der Entwurfsphase sollte man auf derartige Feinheiten noch keinen Wert legen, das lenkt nur vom eigentlichen Problem ab.

Für das Rätselfeld brauchen wir zwei Schleifen, die ineinander geschachtelt sind. Die äußere Schleife läuft dabei über die benutzen **Zeilen**, die innere Schleife über die tatsächlich belegten **Spalten**.

In der Musterlösung wurden zusätzlich bereits die bekannten Zahlen-Buchstaben-Kombinationen eingetragen, das wird bei einem „normalen“ Programm aber sicher entfallen müssen, weil es ja nur für dieses Beispiel gilt. Es hilft aber bei der Fehlersuche und spart ein paar Eingaben.

Neben der Funktionalität benötigt jede Funktion auch noch Parameter. In den vorstehenden Absätzen tauchen alle benötigten Parameter auf, zur Vereinfachung wurden sie **fett** markiert. Hier noch einmal auf einen Blick: Lösungsvektor, Häufigkeitsverteilung, Rätselfeld, belegte Breite, belegte Höhe des Rätselfelds. Dabei werden außer der Breite und Höhe alle Daten in der Funktion geändert (wir wollen sie ja gerade initialisieren), diese Parameter müssen also als Referenzparameter übergeben werden. Wir haben also auch noch Glück, dass die Vektoren und das Rätselfeld als Array abgespeichert sind, die ja grundsätzlich als Referenz übergeben werden. Mit anderen Worten: wir müssen bei den Parametern nichts Besonderes beachten.

Dieses Struktogramm ist unmittelbar in C++ umsetzbar, dabei wurde ausgiebig von der Möglichkeit Gebrauch gemacht, die Parameter anders zu benennen als im Hauptprogramm. Um den Quellcode kurz und übersichtlich zu halten wurde aus dem Lösungsvektor l, aus der Häufigkeit(sverteilung) h und das Rätselfeld schrumpft auf ein r.

```
void init(int r[MAXX][MAXY], int breite, int hoehe, char l[], int h[])
{
	for (int i = 1; i < MAXLETTERS; i++)
		l[i] = 32;	//Lösungsvektor auf Leerzeichen setzen
	l[0] = 64;		// "Schwarzes" Feld
	l[21] = 'R';	// nur für Testzwecke
	l[20] = 'I';
	l[15] = 'N';
	l[16] = 'G';
	for (int i = 1; i < MAXLETTERS; i++)
		h[i] = 0;
	for (int z = 0; z < hoehe; z++)
	for (int s = 0; s < breite; s++)
	{
		r[z][s] = 0;  //Rätsel auf Null setzen
	}
}
```

11.5.3 Rätsel ausgeben

Das Rätselfeld soll ja der gedruckten Vorlage möglichst ähnlich sehen. Leider können wir die Schriftgröße nicht beeinflusse und müssen daher improvisieren. Auch auf die Striche zwischen den einzelnen Feldern verzichten wir, um die Sache übersichtlich zu halten.

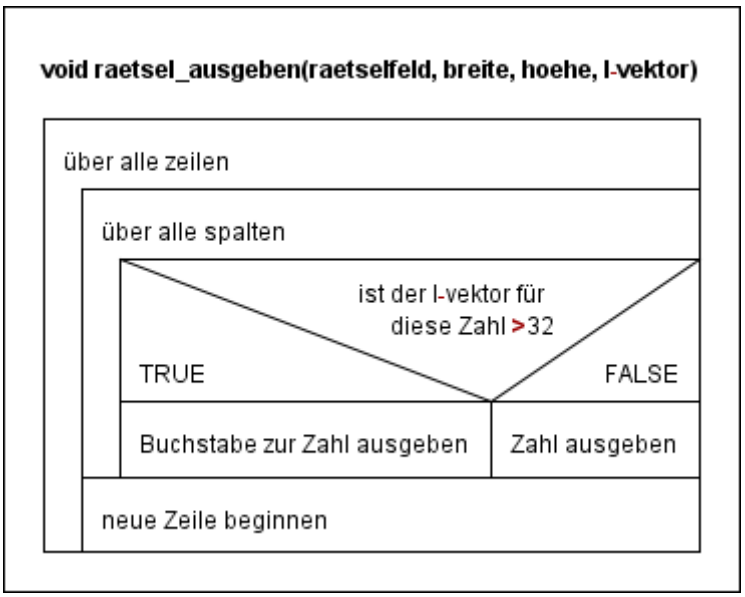
Für ein Feldelement beschränken wir uns auf drei Zeichen in der Breite und ein Zeichen in der Höhe. Die Überlegungen dazu von oben gelten weiterhin.

Das Besondere an der Ausgabe ist, dass normalerweise die Zahl des Feldes ausgegeben wird. Gibt es zu dieser Zahl bereits einen Buchstaben, wird stattdessen der Buchstabe ausgegeben. Woran erkennt man aber, ob es zu einer Zahl schon einen Buchstaben gibt?

Diese Information beziehen wir aus dem Lösungsvektor und nutzen unsere Kenntnisse über die Codierung von Zeichen. Wir haben den Vektor ja mit Leerzeichen (32) initialisiert, wenn also der Lösungsvektor nach wie vor für diese Zahl den Wert 32 enthält, muss die Zahl ausgegeben werden, sonst der Buchstabe.

Die Ausgabe selbst erfolgt analog zur Initialisierung wieder mit zwei geschachtelten Schleifen.

Damit die Funktion arbeiten kann, braucht sie Zugriff auf das Rätselfeld, dessen aktuelle Breite und Höhe und den Lösungsvektor. Diese Daten müssen also per Parameter übergeben werden. Da in der Funktion alle diese Werte nur lesend benutzt werden kann die Übergabe grundsätzlich als Wertparameter erfolgen (unbeschadet der Tatsache, dass Arrays ja grundsätzlich als Referenzparameter übergeben werden)



Der eigentliche Trick bei der Programmierung ist die Bedingung in der Verzweigung. Das Rätselfeld an der aktuellen Position enthält ja grundsätzlich nur die Zahl. Diese Zahl ist der Index für den Lösungsvektor und liefert so den Buchstaben zu dieser Zahl. Die eckigen Klammern sind also an dieser Stelle mit aller Vorsicht und nach gründlicher Überlegung zu setzen!

```
void raetsel_ausgeben(int r[MAXX][MAXY], int breite, int hoehe, char l[])
{
	for (int z = 0; z < hoehe; z++)
	{
		for (int s = 0; s < breite; s++)
		{
			if (l[r[z][s]] > 32 )
			cout << setw(3) << l[r[z][s]];
			else cout << setw(3) << r[z][s];
		}
		cout << endl;
	}
}
```

11.5.4 Lösungsvektor (und Häufigkeitsverteilung) ausgeben

Wir bleiben bei der Ausgabe, nach dem Rätselfeld soll ja auf dem Bildschirm der aktuelle Stand des Lösungsvektors (und in der erweiterten Version auch die Häufigkeitsverteilung) ausgegeben werden.

Aus Platzgründen und auch wegen der Übersicht geben wir den Lösungsvektor horizontal aus und nutzen auch hier die Breite von drei Zeichen für jedes Element aus. Zur Abgrenzung vom Rätselfeld gibt es zunächst eine Trennzeile, dann in je einer Zeile die Häufigkeitsverteilung, den Index und den dazugehörigen Buchstaben (soweit bekannt). Da jede dieser Zeilen eine Wiederholung benötigt, finden wir vier Schleifen vor, die sich alle sehr, sehr ähnlich sehen.

Als Parameter brauch wir hier nur den Lösungsvektor und die Häufigkeitsverteilung, das Rätselfeld ist außen vor.

Dieses Struktogramm bietet jetzt keine Überraschungen, auf die Ausgabe des Zeilenumbruchs nach jeder Schleife wurde verzichtet.

```
void anzeige_loesungsvector(char l[], int h[])
{
	for (int i = 1; i < MAXLETTERS; i++) {
		cout << setw(3) << "-+-";
	}
	cout << endl;
	for (int i = 1; i < MAXLETTERS; i++){
		cout << setw(3) << h[i];
	}
	cout << endl;
	for (int i = 1; i < MAXLETTERS; i++){
		cout << setw(3) << i;
	}
	cout << endl;
	for (int i = 1; i < MAXLETTERS; i++){
		cout << setw(3) << l[i];
	}
```

```
        cout << endl;
}
```

11.5.5 Datei einlesen

Bisher können wir aber noch nichts ausgeben, wir müssen ja unser Rätselfeld irgendwann einmal mit etwas Sinnvollen füllen. Dazu wollen wir die Zahlen, die das Rätsel bilden, aus einer Textdatei einlesen. An dieser Stelle machen wir es uns etwas einfacher, weil wir davon ausgehen, dass unsere Datei perfekt und fehlerfrei erstellt wurde. Eine komplette Behandlung aller denkbaren Fehler würde den Rahmen sprengen, Sie dürfen sich da aber gerne austoben.

Um verschieden große Rätsel verarbeiten zu können, müssen wir zunächst wissen, wie breit und hoch unser Rätselfeld ist. Die Breite und Höhe bilden deshalb die erste und zweite Zeile unserer Datei. Daran schließen sich Zeile für Zeile die einzelnen Rätselfelder an, entweder steht dort die ausgedruckte Zahl oder eine Null für ein Schwarzfeld. Für unser Beispiel von Seite 124 sieht die Datei also so aus:

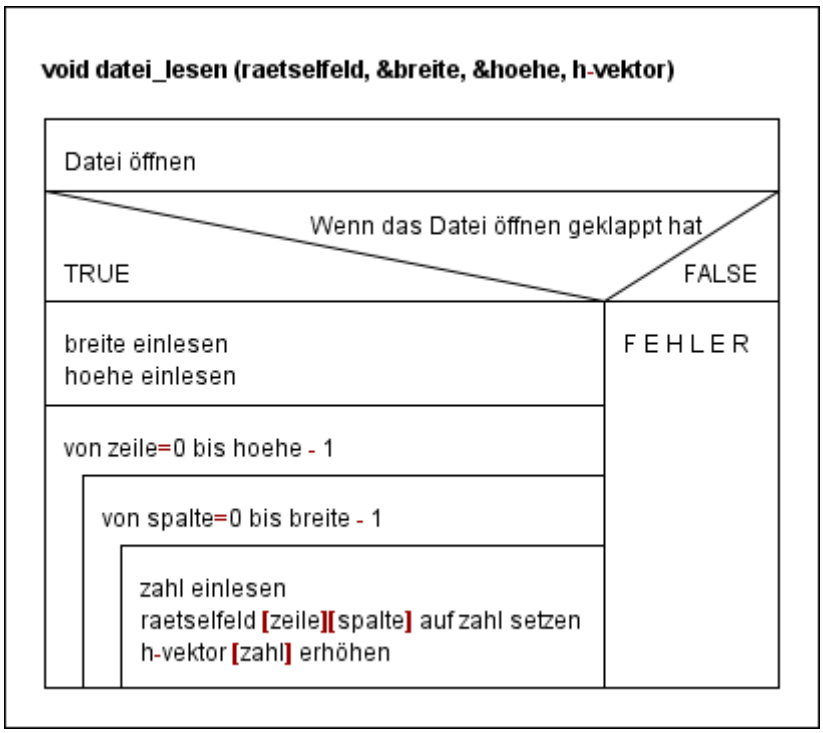
```
19
10
18 09 13 20 23 13 25 00 06 13 14 24 00 00 00 01 14 00 14
00 14 22 15 13 00 12 15 21 12 05 00 21 20 16 04 21 04 10
07 12 14 01 21 14 24 00 04 21 14 25 13 00 14 21 02 13 13
12 00 21 20 15 16 00 10 15 04 06 00 08 04 16 14 00 10 00
13 25 24 18 00 14 17 13 19 00 00 17 04 21 13 00 09 13 05
21 12 13 00 26 03 11 02 00 00 05 14 15 23 00 26 14 21 13
00 19 00 24 05 13 04 00 14 11 14 19 00 13 16 20 25 00 19
11 12 15 20 14 00 24 04 06 14 22 00 15 04 21 01 10 13 13
13 10 10 13 15 13 21 00 13 03 13 15 24 00 14 21 13 10 00
24 00 01 21 00 00 00 26 15 14 15 00 03 13 24 13 21 14 15
```

Unser Rätsel ist 19 Felder breit (horizontal) und 10 Zeilen hoch (vertikal). Um die Struktur besser zu erkennen, wurden die einstelligen Zahlen durch eine führende Null ergänzt, das hat aber keinen Einfluss auf das Programm.

Wir beginnen also damit, die Datei zu öffnen, den Namen haben wir im Programm fest vorgegeben, das kann und darf man aber in der eigenen Version auch ändern. Zur (minimalen) Sicherheit sollte man überprüfen, ob diese Operation geklappt hat und nur dann weitermachen, sonst ergeben alle weiteren Aktionen relativ wenig Sinn.

Nach dem Einlesen von Breite und Höhe kann das eigentliche Rätsel mittels zweier geschachtelter Schleifen (da war doch schon mal was…) gelesen und im Rätselfeld eingetragen werden.

Da jedes Element des Rätselfeldes einzeln eingelesen wird, ist das der ideale Zeitpunkt, auch die Häufigkeit jeder eingelesenen Zahl zu ermitteln. Alternativ lässt sich das aber auch nachträglich aus dem Rätselfeld ermitteln.



Diese Funktion braucht natürlich das Rästelfeld und die Häufigkeitsverteilung, um sie zu füllen. Diese beiden Arrays werden aber sowieso als Referenzparameter übergeben und stellen damit kein Problem dar.

Zusätzlich stellt diese Funktion aber auch die tatsächliche benutzte Breite und Höhe des Rätsels fest. Damit diese Informationen an den Aufrufer (also in unserem Fall die `main`()-Funktion) gelangen, müssen wir hier aktiv eingreifen und die Parameter als Referenzparameter setzen. Das erledigt das unscheinbare & vor dem Parameternamen. Würde es sich nur um eine Größe handeln, könnte man diese über den Funktionsnamen mit Hilfe der `return`-Anweisung zurückgeben, das scheitert hier aber, weil sowohl Höhe wie Breite ermittelt werden und zurück gegeben werden müssen.

```
void datei_lesen(int r[MAXX][MAXY], int &breite, int &hoehe, int h[MAXLETTERS])
{
	fstream dat;
	int temp;
	dat.open ("raetsel.dat", ios::in);
	if (dat)
	{// alles ok
		dat >> breite;
		dat >> hoehe;
		for (int z = 0; z < hoehe; z++)    {
			for (int s = 0; s < breite; s++)   {
				dat >> temp;
				r[z][s] = temp;
				h[temp]++;
			}
		}
	}
	else
		cout << "Das stimmt was nicht!";   // Katastrophe
}
```

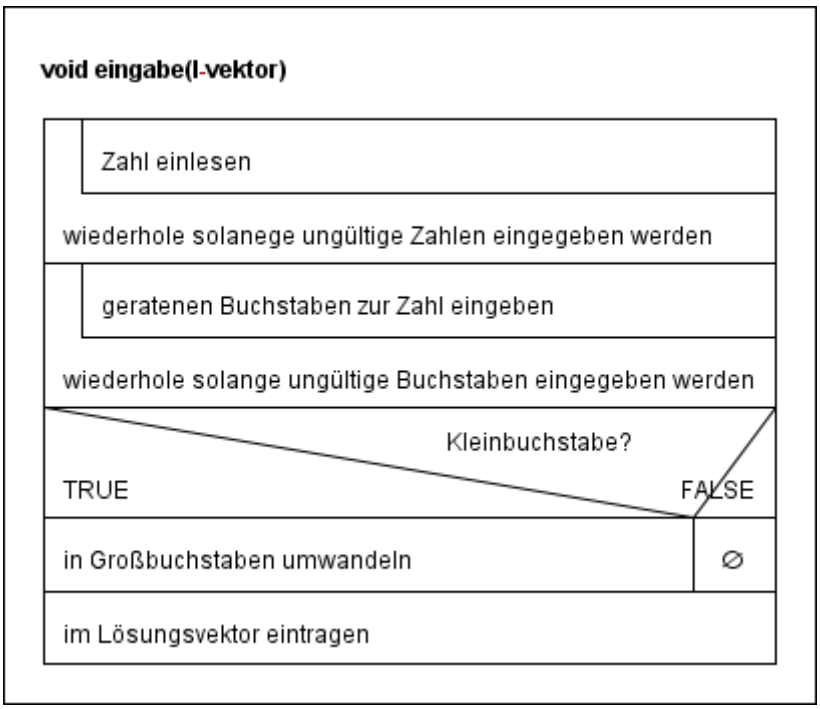
11.5.6 Buchstaben eingeben

Fehlt noch die Interaktion des Benutzers, er soll ja schließlich die Chance haben, zu den Zahlen mehr oder weniger sinnvoll Buchstaben zu erraten und dann auch eingeben können.

Dank unserer Vorüberlegung bei den Datenstrukturen reicht es nach solchen Eingaben, den entsprechenden Eintrag im Lösungsvektor zu ändern, das Rätselfeld selbst bleibt unangetastet. Das vereinfacht die Eingabe an sich und spart uns das mühsame Aktualisieren des Rätselfeldes. Und nebenbei brauchen wir nur den Lösungsvektor als Parameter.

In diese Funktion kann man jetzt beliebig viel Grips und Energie einfließen lassen, indem man alle denkbaren und undenkbaren Ideen des Anwenders abfängt. Wir beschränken uns an dieser Stele darauf, dass wir überprüfen, ob die eingegebene Zahl und der dazu geratene Buchstabe im gültigen Bereich ist. Als kleine Erleichterung (und aus optischen Gründen) wandeln wir eingegebene Kleinbuchstaben in den passenden Großbuchstaben um, damit die Ausgabe einheitlich aussieht.

Denkbar sind Erweiterungen, die auf doppelte Buchstaben im Lösungsvektor prüfen oder Lücken im Lösungsvektor erkennen.



Die Eingaben erfolgen jeweils mit einer fußgesteuerten Schleife, man muss ja zunächst einmal etwas eingeben, das man danach überprüfen kann.

Für die Umwandlung von Klein- in Großbuchstaben nutzen wir schamlos den ASCII-Code und die Tatsache, dass der Datentyp `char` ja nur ein getarnter `int`-Typ ist, aus.

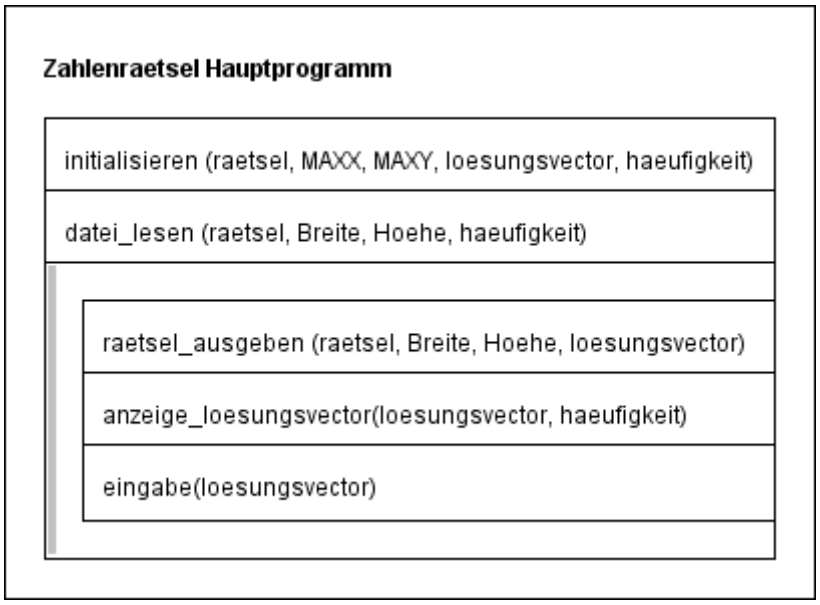
```
void eingabe(char l[])
{
	int zahl;
	char ch;
	do{
		cout << "Fuer welche Zahl haben Sie einen Buchstaben? :";
		cin >> zahl;
	} while (zahl > MAXLETTERS - 1 || zahl < 0);
	do{
		cout << "Soll welcher Buchstaben werden: ";
		cin >> ch;
		if (ch > 96) ch -= 32;
	} while (ch < 65 || ch > 96);
	l[zahl] = ch;
}
```

Der Programmierer hat hier übrigens von seinem Recht auf eigene Ideen Gebrauch gemacht und die Umwandlung in Großbuchstaben in der Eingabeschleife mit erledigt. Das macht die Lösung aber nicht falsch.

11.5.7 Hauptprogramm

Bleibt noch das Hauptprogramm, das den gesamten Ablauf steuert und die einzelnen Funktionen in einer sinnvollen Reihenfolge aufruft.

In der Praxis wird man das Hauptprogramm stückweise mit jeder Funktion erweitern, damit man diese Funktion auch testen kann. Hier im Buch ist das aber ungeschickt, deswegen kommt das Beste zum Schuss.



Hier haben wir jetzt ein wenig sehr vereinfacht. Nach dem Initialisieren und dem Einlesen der Datei findet sich die eigentlich verbotene Endlosschleife. In dieser Schleife wird die Ausgabe des Rätselfeldes und des Lösungsvektors (inklusive Häufigkeitsverteilung) sowie die Eingabe eines Zahlen-Buchstaben-Paares unendlich oft wiederholt.

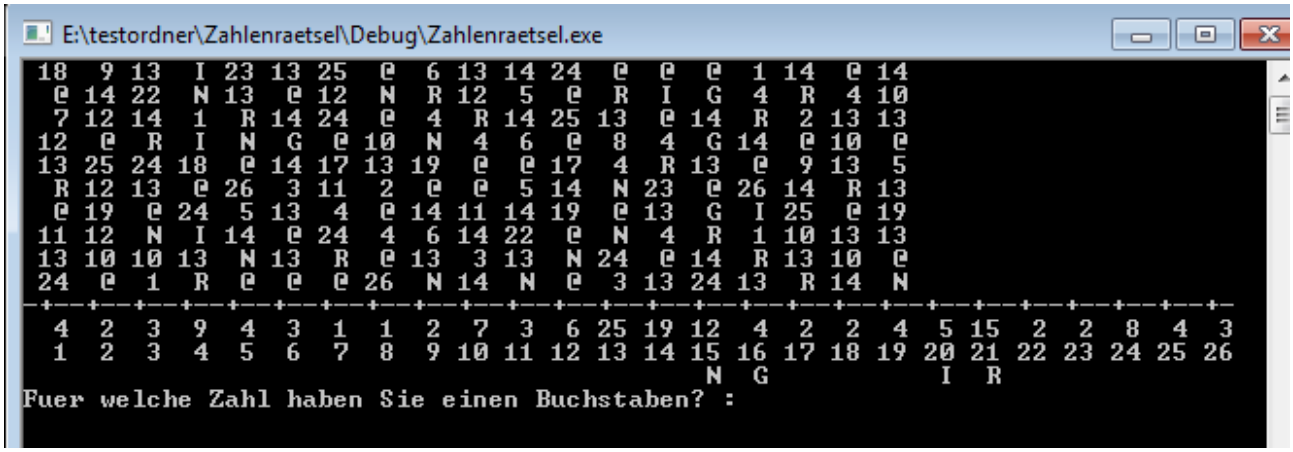
Es ist nämlich gar nicht so einfach, hier ein vernünftiges Abbruchkriterium zu finden. Man könnte die Anzahl der Rateversuche begrenzen, oder man hört auf, wenn der Lösungsvektor „voll" ist (wie stellt man das am besten fest?) oder man fragt „Weitermachen J/N?". Das dürfen Sie selbst ergänzen.

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

const int MAXX = 25, MAXY = 25;  // Maximale Breite/Höhe des Rätselfeldes
const int MAXLETTERS = 27;
int raetsel[MAXX][MAXY];
char loesungsvector[MAXLETTERS];
int haeufigkeit[MAXLETTERS];
int Breite, Hoehe;  // tasächliche Größe aus der Datei
```

```
int main()
{
	init(raetsel, MAXX, MAXY, loesungsvector, haeufigkeit);
	datei_lesen(raetsel, Breite, Hoehe, haeufigkeit);
	do {
		raetsel_ausgeben(raetsel, Breite, Hoehe, loesungsvector);
		anzeige_loesungsvector(loesungsvector, haeufigkeit);
		eingabe(loesungsvector);
	} while (true);
}
```

11.5.8 Fazit

Damit ist unsere Aufgabe erfüllt. Probieren Sie es aus, die Listings können Sie abtippen (das übt!) oder per Copy & Paste in ihre Entwicklungsumgebung einfügen. Dabei müssen Sie ein wenig auf die richtige Reihenfolge achten, die Konstanten sollten natürlich ganz am Anfang, vor der ersten Funktion, deklariert werden.



An einigen Stellen haben wir ein wenig vereinfacht, um die eigentliche Lösung nicht zu sehr zu verwässern. Vor allem bei den Überprüfung der Eingaben und bei der Dateibehandlung haben wir es uns einfach gemacht.

Unterm Strich bleibt eine Aufgabe und ihre Lösung, die im Praktikum ohne Probleme akzeptiert wird, wenn Sie die Erläuterungen, die hier überall eingestreut wurden auch selbst geben können.

Ach so: oben steht die erste Ausgabe unseres Programms nach dem Start.Um die vollständige Ausgabe mit den richtigen Buchstaben zu sehen, müssen Sie das Programm allerdings selbst schreiben.

Die folgenden Übungen können Sie alle nach diesem Verfahren angehen und werden damit ziemlich schnell zu guten Ergebnissen kommen. Und dann können Sie mit ruhigem Gewissen in die Klausur gehen, das ist dann keine ernsthafte Hürde mehr.

11.6 Weitere Übungen

Um diese Übungen hier vernünftig lösen zu können, benötigen Sie das Wissen aus unterschiedlichen Abschnitten aus diesem Buch. Unter jeder Aufgabe finden Sie Stichworte, die Ihnen bei der Lösung helfen könnten. Die Übungen sind unterschiedlich schwer lösbar und mal mehr und mal weniger aufwändig. Sie können aber jederzeit Vereinfachungen vornehmen, Sie müssen die Lösung ja nicht im Praktikum vorzeigen.

Einige Übungsaufgaben verlangen nur nach dem Schreiben einer Funktion. Es bleibt aber nicht aus, dass Sie zum Testen dieser Funktionen ein passendes Hauptprogramm brauchen.

Histogramm

Schreiben Sie ein Programm, dass das Würfeln mit zwei Würfeln simuliert und die Verteilung der Augenzahl „grafisch" darstellt. Grafisch bedeutet hier, dass in Relation zur Häufigkeit ein horizontaler Balken aus * oder # ausgegeben wird. Um einen Effekt zu sehen, müssen Sie schon ein paar Tausend Mal würfeln. *(Stichworte: Zufallszahlen, Array)*

Funktionsplotter

Mit diesem Programm sollen (beliebige) Funktionen grafisch auf dem Bildschirm ausgegeben werden. Auch hier bedeutet graphisch, dass die Ausgabe durch * und andere Zeichen dargestellt wird (+|- #...)

Der Anwender darf das Intervall (x-Achse) bestimmen, das Programm ermittelt dann die optimale Darstellung (der verfügbare Bereich soll ausgeschöpft werden). Die Funktion ist im Programm fest vorgegeben (z.B. die Sinus-Funktion, natürlich können Sie das Programm um ein Menü erweitern, das die Auswahl einer Funktion ermöglicht oder Sie lassen Parameter für die Funktion zu). (*Stichworte: zweidimensionales Array, Dreisatz, Schleifen)*

Buchstaben chiffrieren

Gesucht ist ein Programm, das einen eingegebenen Text „verschlüsselt". Die Verschlüsselung erfolgt durch eine einfache Vorschrift: ersetze jeden Buchstaben durch seinen n-ten Nachfolger im Alphabet. Für n=5 würde als aus einem A ein F, aus B ein G, aus C ein H usw. bis zum Z, das zum E wird. Der Wert für n und der zu verschlüsselnde Text sind einzugeben (*Stichworte: ASCII, Modulo, Zeichenkette, Eingaben mit Leerzeichen*)

Komplexe Zahlen

Entwickeln Sie ein Programm, das mit komplexen Zahlen die vier Grundrechenarten durchführen kann. Die Ein- und Ausgabe der komplexen Zahlen soll in der Schreibweise a+ib erfolgen, die Rechenoperationen sollen über ein Menü ausgewählt werden.

Das Programm soll „Kettenrechnungen" ermöglichen, d.h. das Ergebnis einer Berechnung steht sofort als erster Operand für die nächste Operation zur Verfügung (es sei denn, der Anwender wählt im Menü ein anderes Vorgehen). Die Regel „Punktrechnung vor Strichrechnung" gelten bei Kettenrechnungen nicht! (*Stichwort struct*)

Mastermind

Schreiben Sie ein Programm, das das heute kaum noch bekannte Spiel Mastermind aus den 70er Jahren des vorigen Jahrhunderts nachbildet

(Spiel des Jahres 1973). Der Computer „denkt“ sich eine vierstellige Zahl aus den Ziffern 1 bis 6 aus. Diese Zahl wird vom Anwender erraten. Der Computer gibt an, wie viele der vier Ziffern an der richtigen Stelle stehen und wie viele Ziffern in der erdachten Zahl vorkommen, aber nicht an der richtigen Position stehen. Das Spiel endet, wenn alle Ziffern korrekt erraten wurden.

Statt mit Zahlen können Sie das Ganze -wie im Original- auch mit Farben lösen. (*Stichworte: Modulo, Zufallszahlen, Array, Schleifen, Datentyp Bool..* Bildquelle: Wikimedia.org/Mastermind.jpg von ZeroOne)

Palindrom

Schreiben Sie ein Programm (oder eine Funktion), das/die ein eingegebenes Wort (string) darauf überprüft, ob es sich um ein Palindrom handelt. Ein Palindrom ist ein Wort, dessen Buchstaben von vorne und hinten gelesen zum gleichen Ergebnis führen (z.B. Mutakirorikatum oder Reliefpfeiler). Erweitern Sie den Code so, dass Leerzeichen vor dem Test entfernt werden. *(Stichworte: string, Array, char, for-Schleife, ASCII-Wert)*

Tageszeit

Schreiben Sie eine Funktion, die in Abhängigkeit der aktuellen Uhrzeit (des Rechners) „Guten Morgen“, „Guten Tag“ oder „Guten Abend“ wünscht. Um die aktuelle Uhrzeit zu ermitteln können Sie das Programmhäppchen im Kapitel 15 verwenden. Um welche Zeit der Tag bzw. der Abend beginnt bestimmen Sie einfach nach ihrem persönlichen Lebensrhythmus. *(Stichwort: string-Verarbeitung)*

Funktion x^n

Schreiben Sie eine Funktion, die zu einer Gleitkommazahl die ganzzahligen Potenzen berechnet. Nennen Sie die Funktion `double pow (int n)` und überladen Sie so die eingebaute `pow`-Funktion. Zur Mathematik dahinter: x^0 ist immer 1, für alle x. Für negative Exponenten ist das Ergebnis als $(\frac{1}{x})^n$ definiert. Und für x=0 ist das Ergebnis auch 0, solange der Exponent positiv ist. Für negative Exponenten ist die Funktion nicht definiert, da ist Phantasie bei der Fehlerbehandlung angesagt.

Quadratische Gleichung

Erstellen Sie eine Funktion, die die quadratische Gleichung $ax^2+bx+c=0$ löst. Es gilt:

$$x_{1,2}=\frac{(-b\pm\sqrt{b^2-4ac})}{2ac}$$

Wenn Ihnen das zu einfach war: lassen Sie auch komplexe Lösungen zu (wenn also $(b^2-4ac)<0$ ist).

Selection Sort

Dies ist ein anderes Verfahren zum Sortieren von Daten: zunächst wird das Array (der Vektor) nach dem kleinsten Element durchsucht und dieses dann mit dem ersten Element vertauscht. Das Verfahren wiederholt sich dann für den jeweiligen Restvektor.

Schreiben Sie eine Funktion, die ein übergebenes Array (Datentyp frei wählbar) sortiert. Für Fortgeschrittene: verwenden Sie eine Template-Funktion

Mischen

Schreiben Sie eine Funktion, die zwei (sortierte) Vektoren (Datentyp `int` oder `long`) so miteinander vermischt, dass wieder ein sortierter Vektor mit den Elementen aus den beiden Ausgangsvektoren entsteht. Zusatzaufgabe: Der Speicherplatz für den Ergebnis-Vektor ist dynamisch zu beschaffen.

Stack als Listen

Schreiben Sie alle nötigen Funktionen, um einen Stack (der Einfachheit halber für `int`-Werte) mittels einer verketteten List zu realisieren. Stack bedeutet, dass das zuletzt eingegebene Element als erstes wieder ausgegeben wird und vom Stack entfernt wird. Das bedeutet für die Liste, dass immer am Kopf der Liste (Anker) operiert wird.

Sehen Sie neben einer Funktion zum Einfügen und Entfernen des obersten Elementes auch eine Funktion vor, die alle Werte des Stacks ausgibt, ohne sie vom Stack zu entfernen (browse).

Wortfilter

Schreiben Sie ein Programm, dass eingegebene Worte direkt wieder ausgibt, mit einer Ausnahme: steht das Wort auf einer (im Programm fest vorgegebenen) Liste, wird es durch ein „Beep-Beep“ ersetzt. Beginnen Sie mit nur einem Wort auf der Sperrliste und erweitern Sie dann auf mehrere.

Perfekte Zahlen

Eine perfekte Zahl (manchmal spricht man auch von einer vollkommenen Zahl) hat man vor sich, wenn die Summe der echten Teiler dieser Zahl gleich der Zahl selbst ist. Echte Teiler sind alle Teiler, die kleiner als die Zahl selbst sind, inklusive dem trivialen Teiler 1. So ist 6 eine perfekte Zahl, weil sie die Summe aus den drei echten Teilern 1 + 2 + 3 ist. Auch die 28 ist demnach eine perfekte Zahl (1+2+4+7+14=28).

Schreiben Sie eine Funktion, die überprüft, ob eine übergebene Zahl eine perfekte Zahl ist (Rückgabewert vom Typ `bool`). Schreiben Sie dann ein passendes Hauptprogramm, dass die perfekten Zahlen in einem gewissen Intervall ausgibt. (Achtung: die Anzahl der perfekten Zahlen ist überschaubar, selbst der Datentyp `long` kommt da schnell an seine Grenzen. Die ersten 12 Zahlen finden Sie zur Kontrolle unter https://de.wikipedia.org/wiki/Vollkommene_Zahl)

Wortendung

Schreiben Sie eine Funktion `endet_mit(str, endung)`, die feststellt (`bool`), ob der übergebene String *str* mit *endung* aufhört. Dabei sind sowohl Zeichenketten als auch ein einzelner Buchstabe als *endung* erlaubt.

Whitespace-Killer

Schreiben Sie eine Funktion, die aus einem übergebenen String alle Whitespace-Zeichen entfernt und den String entsprechend kürzt. Whitespace-Zeichen sind Leerzeichen, Tabulatoren, Zeilenende und Zeilenvorschübe (32, 8, 9, 10 13). Sie könne auch pauschal alle Zeichen mit einem ASCII-Wert < 32 aus dem String eliminieren.

Tausenderstellen

Große Zahlen neigen dazu, schnell unübersichtlich zu werden (das haben Sie bei den perfekten Zahlen ja schon gesehen). Deswegen hatten findige Köpfe die Idee, nach jeweils drei Stellen einen Dezimalpunkt als Tausendertrennzeichen einzuführen.

Schreiben Sie eine Funktion,die eine übergebene Zahl (Datentyp `long long` oder `long`) so in eine Zeichenkette umwandelt, dass diese Tausendertrennzeichen eingefügt wurden.

Für Fortgeschrittene: bei Gleitkommazahlen ist das Problem deutlich komplexer, die Trennzeichen werden hier nur in der Mantisse (vor dem Komma) eingebaut.

Zahlenbasis umwandeln

Zahlen lassen sich ja in jedem beliebigen Zahlensystem darstellen, wir haben hier schon dezimal (Basis 10), hexadezimal (Basis 16) und oktal (Basis 8) gesehen. Aber auch jede andere Basis ist grundsätzlich möglich.

Schreiben Sie ein Programm, das eine eingegebene Dezimalzahl in eine beliebige andere Zahlenbasis umwandelt. „Beliebig" beschränken wir auf maximal 36, weil dies mit den Ziffern 0 bis 9 und den Buchstaben A bis Z (für die Werte 10 bis 36) noch relativ einfach dargestellt werden kann.

Der Algorithmus dahinter geht so: die Zahl wird wiederholt durch die neue Basis dividiert (ganzzahlig!), der Rest dieser Division „notiert". Wenn von der Zahl nichts mehr übrig ist (also die Division eine 0 ergibt), ist das Verfahren zu Ende. Die „notierten" Reste ergeben *rückwärts gelesen* die Zahl im neuen Zahlensystem.

Erweiterung: Man darf auch die Eingabe in einer beliebigen Basis machen. Diese wird dann zuerst in die Basis 10 umgewandelt.

Sieb des Eratosthenes

Der griechische Mathematiker und Philosoph Eratosthenes hat bereits sehr früh eine Methode gefunden, um die Primzahlen ganz ohne Division zu ermitteln. Er hat einfach alle Zahlen des fraglichen Intervalls (beginnend mit 2) aufgeschrieben und dann systematisch die Vielfachen aller Zahlen entfernt. Im ersten Durchgang werden so alle Vielfachen von 2 (also alle geraden Zahlen) aussortiert. Danach wird das Verfahren mit der nächsten Zahl, die stehen geblieben ist (das sollte die 3 sein) analog verfahren. Jetzt sind zusätzlich alle Vielfachen von 3 davon betroffen und werden entfernt. Dieses Verfahren wird so lange wiederholt, bis man keine Vielfachen mehr zum Streichen hat.

Am Ende hat man eine Liste mit allen Primzahlen im gesuchten Bereich.

Schreiben Sie ein Programm, dass das Sieb des Eratosthenes nachbildet und die Primzahlen im Bereich 2 bis 1000 (10.000, 100.000, 1.000.000) ermittelt.

Magisches Quadrat

Ein magisches Quadrat ist eine quadratische Anordnung von ganzen Zahlen, deren Summe horizontal, vertikal und diagonal immer gleich ist. Berühmt ist das 4*4 Quadrat von Albrecht Dürer:

16	3	2	13
5	10	11	8
9	6	7	12
4	15	14	1

Hier beträgt die Summe immer 34.

Ihre Aufgabe ist es, eine Funktion zu schreiben, die feststellt, ob ein übergebenes, zweidimensionales Array ein magisches Quadrat darstellt. Wenn es sich nicht um ein Quadrat handelt, können Sie einfach sofort ein `false` zurück geben.

Und wenn Sie es unbedingt versuchen wollen: schreiben Sie eine Funktion, die zu einer gegebenen Summe und der Größe des Quadrates das passende magische Quadrat berechnet. Ein mögliches Vorgehen beschreibt der Artikel in der Wikipedia: https://de.wikipedia.org/wiki/Magisches_Quadrat.

Primfaktorenzerlegung

Jede natürliche Zahl > 2 kann als Produkt einer Folge von Primzahlen dargestellt werden. Schreiben Sie ein Programm, dass diese Aufgabe erfüllt. Die Zahl 60 z.B. kann als 2*2*3*5 dargestellt werden, d.h. eine Primzahl kann durchaus mehrfach vorkommen.

Simulation einer Kettenreaktion

Disney hat bereits 1957 in einem Film den Ablauf einer atomaren Kettenreaktion optisch ansprechend dargestellt. Den Film gibt es auf YouTube: https://youtu.be/QRzl1wHc43I?t=35m22s

Das Experiment besteht aus 625 Mausefallen, die in einem Raster von 25 Zeilen und 25 Spalten auf dem Boden einer großen Kiste aufgestellt sind, jede Mausefalle wird zu Beginn mit zwei Tischtennisbällen geladen und gespannt. Wird die Mausefalle von einem Tischtennisball getroffen löst sie aus und schießt ihre beiden Bälle ebenfalls los. Nach wenigen Augenblicken sind alle Mausefallen ausgelöst und alle Tischtennisbälle verschossen (Lawineneffekt). Wir wollen uns hier auf die weniger spektakuläre Simulation mittels eines Programms beschränken und machen für die Simulation folgende Annahmen:

- Jeder abgeschossene Tischtennisball landet auf einer Mausefalle, die per Zufall (zufällige Spalte und zufällige Reihe) ausgewählt wird. Ist diese Falle noch geladen, löst sie aus. Ist die Falle schon ausgelöst worden, passiert nichts, der Ball hat keinerlei Wirkung.
- Jeder Ball, der eine Mausefalle getroffen hat, ist danach aus dem Rennen und hat keinerlei weitere Folgen. (also kein Abprallen, zurück springen oder ähnliches)
- Bälle, die von einer Mausefalle abgeschossen werden, können an den Wänden der Kiste ein- bis viermal reflektiert werden.

Diese Aufgabe ist sehr anspruchsvoll! Überlegen Sie sehr genau und ausführlich, wie Sie den aktuellen Zustand der Kiste und der Tischtennisbälle darstellen.

Starten Sie die Simulation mit einem (zusätzlichen) Ball, der von außen eingeworfen wird und eine zufällig ausgewählte Mausefalle trifft. Das löst zwei Bälle aus, für jeden dieser Bälle muss dann berechnet werden, wie er sich verhält (entweder prallt er gegen eine Wand und wird reflektiert (maximal viermal!) oder er trifft direkt auf eine Mausefalle). Klingt stark nach Rekursion.

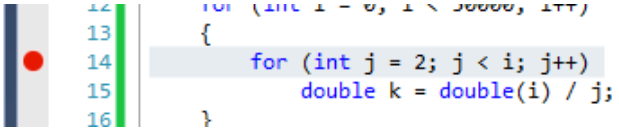
Die Simulation endet, wenn alle Bälle aus dem Rennen sind. Dann soll das Programm ausgeben, wie viele Schritte (Iterationen) gebraucht wurden und wie viele der Mausefallen ausgelöst wurden. Sie können auch die maximale Anzahl von „fliegenden" Bällen ermitteln.

Die Wahrscheinlichkeiten für Reflexion oder Treffen einer Mausefalle sollte als Konstante definiert werden, um die Simulation regeln zu können.

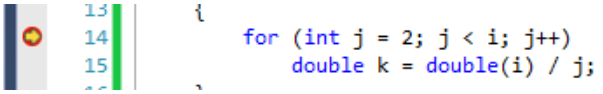
11.7 Dem Fehler auf der Spur (Der Debugger)

Manchmal ist es gar nicht so einfach, einen Fehler einzugrenzen oder überhaupt zu finden, das Programm tut etwas, aber das Ergebnis sieht völlig anders aus, als die Erwartungen. Dann hilft nur der Einsatz des Debuggers (hier jetzt am Beispiel von VisualStudio)

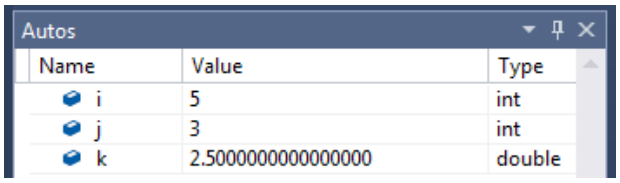
Mit dem Debugger kann man vor allem eins: Programme kontrolliert ablaufen lassen. Dabei kann man Haltepunkte (Breakpoints) setzen und sich den Inhalt von Variablen anzeigen lassen. In einer IDE klickt man dazu einfach nur ganz links vor eine Zeile im Quelltext, ein dicker roter Punkt zeigt den Haltepunkt an:



Dieser rote Punkt hält das Programm beim Laufen **vor** dieser Zeile an. Der Ausführen-Pfeil (gelb) steht *vor* Zeile 14, diese wurde also noch nicht ausgeführt.



Bei laufendem Debugger erhält man ein zusätzliches Fenster mit dem Titel „Autos“, in dem die Inhalte aller Variablen, die gerade aktiv sind, dargestellt werden.

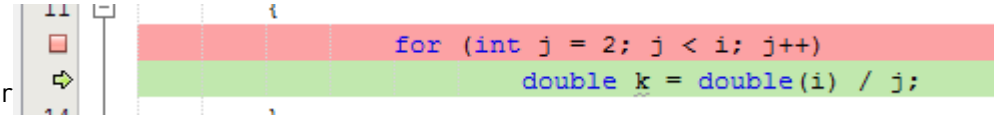


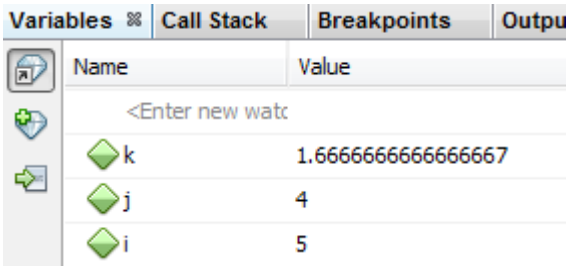
Fehlt das Fenster, kann man es über das Menü DEBUG→Windows → Autos einschalten und bei Bedarf irgendwo in der IDE andocken.

Zunächst passiert weiter überhaupt nichts, das Programm steht ja. Aber man hat die Möglichkeit, es schrittweise weiter laufen zu lassen. Dafür gibt es diese Bedienungshilfen in der Symbolleiste, das meist genutzte Symbol dürfte der mittlere der drei blauen Pfeile, der von einer Anweisung zur nächsten springt (Alternativ erfüllt die Funktionstaste F10 den gleichen Zweck).

Zum Eingrenzen setzt man zu Beginn einen Breakpoint am Anfang des Programms und schaut dem Programmablauf zu. Dabei achtet man auf Programmzweige, die schlicht übersprungen werden, Schleifen, die deutlich länger (oder kürzer) laufen als erwartet, oder Funktionsergebnisse, die überhaupt nicht in die Theorie passen. Ein Quell von Problemen ist die Verarbeitung von Dateien, da gibt es beliebig viele Dinge, die schief gehen können, aber nicht so leicht gefunden werden.

Bei **NetBeans** gibt es all diese Funktionen auch, sie sehen nur etwas anders aus. Statt eines runden roten Punktes gibt es ein rotes Quadrat, das die Zeilennummer ersetzt. Der Ausführen-Pfeil ist grün und zum Fortsetzen dient der linke der orangen Pfeile (alternativ die Taste F8). Und die Variablen haben einen Reiter neben dem Output-Fenster statt eines eigenen Fensters. Also rein optische Unterschiede gibt es, im Funktionsumfang sind beide IDE identisch.





11.8 Fehlermeldungen und was dahinter steckt

Nur sehr wenige Programme laufen nach dem Eingeben fehlerfrei durch den Compiler. Im Regelfall treten Warnungen oder mehr oder weniger schwerwiegende Fehler. In der folgenden Tabelle finden sich einige typische Fehlermeldungen von VisualStudio aufgelistet (sortiert nach der Fehlernummer) und ein Tipp zur Lösung. Leider ist der Compiler kein Hellseher und liefert oftmals auch Fehlermeldungen, die überhaupt nichts mit dem wahren Fehler zu tun haben, weil der eigentliche Fehler nicht richtig erkannt wurde.

Fehlernummer	Fehlertext (gekürzt)	Ursache, Abhilfe, Tipp...
C1004 **C1071**	unexpected end of file found unexpected end of file found in comment	Die Datei ist zu früh zu Ende, oft fehlt eine } Kommentar geht über das Dateiende hinaus
C2015	too many characters in constant	Eine char-Konstante enthält mehr als ein Zeichen
C1083	Cannot open filetype file: 'dateiname': message	Die Datei „dateiname“ kann nicht gefunden werden
C2046 **C2047**	`illegal case` `illegal default`	Das case oder default ist natürlich nicht illegal, befindet sich aber außerhalb des switch-Blocks. Meistens fehlt die { hinter dem switch oder eine } hat sich zusätzlich in einen der Fälle gemogelt
C2050	switch expression not integral	Die Variable in der switch-Anweisung hat einen nicht-abzählbaren Type. Hier sind nur int, char und enum zulässig
C2051	case expression not constant	Der Wert hinter case muss eine Konstante sein, Variablen sind nicht erlaubt
C2057	expected constant expression	Der Ausdruck muss eine Konstane ergeben (also bereits zur Compile-Zeit feststehen)
C2065	`'x' : undeclared identifier`	Die Variable x ist (in diesem Scope) nicht definiert, also entweder hat man sich beim Eingeben vertippt (Groß-/Kleinschreibung?) oder man hat die Definition wirklich vergessen.
C2065	'id' : undeclared identifier	Eine Variable muss vor ihrer Verwendung deklariert werden.
C2086	'id' : redefinition	Der Bezeichner/Variable id wurde mehrfach definiert, evtl. sogar mit unterschiedlichen Typangaben

Fehlernummer	Fehlertext (gekürzt)	Ursache, Abhilfe, Tipp...
C2106	'op' : left operand must be l-value	Auf der linken Seite der Zuweisung muss eine Variable stehen. Oder sollte es ein Vergleich == werden?
C2108	subscript is not of integral type	Der Index (eines Arrays) muss abzählbar sein (also NICHT double...)
C2144	`syntax error : 'int' should be preceded by ';'` `(int steht stellvertretend für alle Datentypen)`	Der häufigste Fehler überhaupt: das vergessene Semikolon. Aber Achtung, es fehlt üblicherweise am Ende der vorherigen Zeile!
C2440 C2446	'conversion' : cannot convert from 'type1' to 'type2' 'operator' : no conversion from 'type1' to 'type2'	Die Umwandlung von type1 in type2 ist nicht möglich
C2447	`'{' : missing function header (old-style formal list?)`	Der Funktionsrumpf beginnt nicht mit einer öffnenden { Oder beim Kopieren des Funktionskopfes aus der Header-Datei wurde das Semikolon mitkopiert.
C2543 C2544	expected ']' for operator '[]' expected ')' for operator '()'	De schließende Klammer] bzw.) fehlt
C2561	`'id' : function must return a value`	Die Funktion id muss einen Wert zurückgeben
C2562	'id' : 'void' function returning a value	Eine als void deklarierte Funktion **darf** keinen Wert zurückgeben
C2601	'func' : local function definitions are illegal	Es gibt in C++ keine Funktionen innerhalb von Funktionen. Wurde die } vergessen?
C2660	'func' : function does not take *number* parameters	Die Anzahl der Parameter beim Aufruf unterscheidet sich von der Anzahl der Parameter in der Funktionsdefinition
C4100	'id' : unreferenced formal parameter	Einer der formalen Parameter wird im Funktionsrumpf nicht verwendet
C4101	'id' : unreferenced local variable	Die angegebene Variable wird zwar definiert, aber nicht benutzt
C4244	'conversion' conversion from 'type1' to 'type2', possible loss of data	Bei der (automatischen) Umwandlung von type1 in type2 besteht die Gefahr von Umwandlungsverlusten (z.B. `double` nach `int`)

Fehlernummer	Fehlertext (gekürzt)	Ursache, Abhilfe, Tipp...
C4700	`uninitialized local variable 'x' used`	Die Variable x hat zum Zeitpunkt ihrer Verwendung keinen (sinnvollen) Wert. Mit anderen Worten: diese Variable sollte mindestens einmal auf er linken Seite einer Zuweisung stehen, bevor sie auf der rechten Seite verwendet (benutzt) wird.
C4716	'func' must return a value	Die Funktion func **muss** einen Wert zurückliefern
LNK2001	unresolved external symbol "name"	Eine verwendete Funktion ist nicht zu finden. Häufig ein Tippfehler beim Funktionskopf.

12 Informatik im Schnelldurchgang

Viele Studierende setzen Informatik und Programmieren gleich, nur die Informatik-Studenten selbst (und deren Professoren) sehen das etwas anders. Für die ist Informatik die Wissenschaft von und über die Verarbeitung von Informationen, das Programmieren spielt dabei zwar eine große Rolle, ist aber letztendlich nur ein Werkzeug, aber ein Werkzeug, das man als Informatiker beherrschen muss.

Dieses Kapitel gab es in früheren Versionen dieses Skriptes überhaupt nicht, auch ich habe mich bisher auf das Programmieren an sich beschränkt. Die Vorlesung heißt aber fast immer *Informatik* oder *Einführung in die Informatik* oder oder oder. Das magische Wort *Programmieren* kommt dann erst beim genauen Lesen der Modulbeschreibung vor. In der Praxis nimmt das Programmieren dann aber doch den größten Teil der Veranstaltung, insbesondere der Übungen, ein. Da verhält es sich ähnlich wie beim Erlernen einer Fremdsprache, da steht auch *Landeskunde* im Lehrplan, aber die meiste Zeit beschäftigt man sich mit Grammatik und Vokabeln lernen.

Ich habe versucht dieses Kapitel so weit es irgendwie geht, vom Rest des Skriptes unabhängig zu gestalten. Wenn es Ihnen nur ums Programmieren in C++ geht haben Sie die relevanten Seiten schon hinter sich gebracht. Je nach Studiengang werden Sie sich amüsieren, wenn Sie hier über Schaltalgebra, digitale Schaltungen und Hardware lesen, und dabei gleichzeitig eine eigene Vorlesung dazu belegt haben. Aber das ist nicht bei jedem Leser in der Zielgruppe so.

Der hier behandelte Stoff umfasst ein weites Gebiet und es ist nicht einfach, eine durchgehende Linie zu finden. Viele Abschnitte stehen gleichberechtigt nebeneinander und bauen *nicht* aufeinander auf. Alle zusammen bilden das Fundament für die Informatik. Scheuen Sie sich daher nicht, die einzelnen Abschnitte nach Lust und Laune in beliebiger Reihenfolge zu studieren, je nach Dozent kommen einzelne Punkte mehr oder weniger, vielleicht auch gar nicht in der Vorlesung vor. Dieses Kapitel bietet also viele Informationen zum Nachschlagen und Nachlesen oder einfach zum Schmökern.

12.1 Informatik immer und überall

Wie bei jedem neuen Gebiet, mit dem man sich beschäftigen muss oder möchte, stehen am Anfang viele Begriffe, deren Bedeutung man glaubt zu wissen oder die einem völlig fremd sind. Ein paar Begriffsbestimmungen und Definitionen bleiben uns also nicht erspart. Beginnen wir doch mit dem Begriff *Informatik* an sich und bemühen dazu die Wikipedia:

»nformatik ist die Wissenschaft von der systematischen Darstellung, Speicherung, Verarbeitung und Übertragung von Informationen, besonders der automatischen Verarbeitung mithilfe von Digitalrechnern« Historisch hat sich die Informatik einerseits aus der Mathematik entwickelt, andererseits als Ingenieursdisziplin aus dem praktischen Bedarf nach der schnellen und insbesondere automatischen Ausführung von Berechnungen.

Der Begriff wurde schon 1957 das erste Mal erwähnt, seit 1968 fand er Eingang in die Wissenschaft und setzte sich im deutschen Sprachraum durch. Nur im Englischen wird im allgemeinen der Begriff *Computer Science* verwendet.

Relativ schnell haben sich Teildisziplinen heraus kristallisiert, die auch heute noch die Basis der Informatik bilden. Wundern Sie sich nicht, wenn in der folgenden Übersicht Begriffe auftauchen, die Ihnen noch spanischer vorkommen wie die Informatik an sich. Sie müssen diese Begriffe nicht sofort verstehen, sie werden hier nur beispielhaft erwähnt, damit Sie dann bei Gelegenheit tiefer in die Materie einsteigen können.

12.1.1 Theoretische Informatik

In der theoretischen Informatik spielen vor allem mathematische und formale Aspekte eine Rolle. Die Nähe zur Mathematik ist hier am größten und unübersehbar. Obwohl hier die Verzahnung der Teilgebiete sehr stark ist, trennt man auf dem Papier noch einmal zwischen

Formale Sprachen
Formale Sprachen beschreiben in der Theoretischen Informatik im wesentlichen Denkmodelle und mathematische Darstellungen und sind nicht für den praktischen Einsatz beim Programmieren gedacht oder geeignet, sehr oft fehlen hier die dafür notwendigen Kommunikationseinheiten (Ein- und Ausgabe)

Automatentheorie
Automaten sind eine Art Modellrechner, auf denen man wunderbar Probleme lösen kann. Theoretisch. Man findet sie häufig bei lexikalischen Scannern und Parsern im Compilerbau und für den Entwurf von Programmiersprachen.

Komplexitätstheorie
Wenn man ein Verfahren zur Lösung eines Problems (einen Algorithmus) gefunden hat, beschäftigt sich die Komplexitätstheorie damit, wie sich dieser Algorithmus auf Rechenzeit, Speicherbedarf und ähnliche Merkmale eines Rechners auswirkt. Und versucht natürlich, diese Anforderungen in jeder Hinsicht zu optimieren.

12.1.2 Technische Informatik

Die technische Informatik beschäftigt sich im wesentlichen mit Entwurf, Realisierung und Betrieb von Rechnern und der dazugehörigen Peripherie. Hier spielt die Nähe zur Elektrotechnik und Elektronik eine große Rolle, aber neben der Digitaltechnik sind auch Logik und diskrete Mathematik nicht zu vernachlässigen.

Rechnerarchitektur
Die Rechnerarchitektur behandelt den Entwurf, Aufbau und Organisation von Rechnern. Dazu zählen Prozessoren, Speicher und Bussystem und deren Zusammenspiel.

Automatisierungstechnik
Die Automatisierungstechnik ist eigentlich ein Teilgebiet des Anlagenbaus und der Ingenieurwissenschaften aus Maschinenbau und Elektrotechnik. Die Automatisierung setzt aber immer mehr auf Rechner und wurde so zu einem Teilgebiet der Informatik.

Eingebettete Systeme (Embedded systems)
spielen in immer mehr Bereichen eine tragende Rolle. Kleine und kleinste Computer übernehmen die Aufgaben der Steuerung und Kontrolle von Geräten und Objekten. Dazu zählen viele Geräte der Unterhaltungsindustrie, aber auch Router, Set-Top-Boxen, Geräte in der Medizintechnik oder im Auto.

12.1.3 Praktische Informatik

Die Praktische Informatik entwickelt -im Gegensatz zur Theoretischen Informatik- Methoden und Konzepte zur Lösung von Problemen der realen Welt. Ein zentrales Thema ist die Softwaretechnik zur systematischen Entwicklung von (großen) Softwaresystemen.

Die von der Theoretischen Informatik entwickelten Compiler werden von der Praktischen Informatik realisiert und dann auch genutzt, um andere Softwaresysteme zu entwickeln.

Programmiersprachen
stellen einen wesentlichen Aspekt der Praktischen Informatik dar. Im Einsatz sind Dutzende von Programmiersprachen, die entweder für spezielle Einsatzgebiete entwickelt wurden oder ganz allgemein eingesetzt werden können (mit einem Vertreter dieser Kategorie bekommen Sie es dann im Praktikum zu tun).

Übersetzerbau
schafft erst die Möglichkeiten für den Einsatz von Programmiersprachen. Der konkreten Programmiersprache und dem darin erstellten Programm auf der einen Seite steht der konkrete Rechner auf der anderen Seite gegenüber. Der Compiler ist jenes Softwaresystem, dass beide Seiten miteinander verbindet.

Betriebssysteme
stellen die Verknüpfung eines konkreten Rechners (der Hardware) mit allen anderen Programmen und Softwaresystemen her. Sie sorgen dafür, dass alle Komponenten eines Rechners genutzt und angesprochen werden können und machen so erst den Einsatz von anderen Programmen möglich.

Simulation
Die Simulation von technischen oder natürlichen Vorgängen ist ein weites und ständig wachsendes Gebiet. In vielen Fällen ist es nämlich günstiger, auf eine Simulation zu setzen und nicht auf den realen Prozess. Der Flugsimulator steht hier stellvertretend für die Gruppe der technischen Simulationen, ohne die eine Ausbildung zum Piloten heute nicht denkbar wäre.

Aber auch natürliche Prozesse wie Wetterphänomene oder Wachstumsprozesse lassen sich durch Simulation besser verstehen und problemlos nachbilden.

Künstliche Intelligenz
Die Künstliche Intelligenz treibt seit mehreren Jahren die Praktische Informatik an und führte bereits zu vielen Innovationen. Ziel ist auf der einen Seite, ein System zu schaffen, welches kreativ, emotional und überaus intelligent ist, andererseits geht es darum, einzelne Eigenschaften der menschlichen Intelligenz auf konkrete Probleme anzuwenden.

12.1.4 Angewandte Informatik

Die Angewandte Informatik sammelt jene Fachgebiete unter sich, die sehr starke und konkrete Berührungspunkte mit anderen Wissenschaften haben

Bioinformatik
Auf den ersten Blick haben Biologie und Informatik nicht all zu viel gemeinsam, aber das täuscht. Insbesondere bei der Genforschung fallen immense Datenmengen an, die ohne Computerunterstützung nicht in den Griff zu bekommen wären.

Wirtschaftsinformatik
Die Wirtschaftswissenschaften befassen sich mit allen möglichen Fragen und Problemstellungen rund um Betriebs- und Volkswirtschaft. Computer werden dazu genutzt, die bei Verwaltung und Auftragsabwicklung anfallenden Datenmengen zu ordnen und nutzbar zu machen.

Computerlinguistik
Schon vor Alexa, Cortana und Siri war die Verarbeitung von natürlicher Sprache ein wichtiges Feld der Informatik. Aber dank Computerlinguistik und künstlicher Intelligenz ist diese Sparte der Informatik heute in vielen Wohnzimmern anzutreffen.

Geoinformatik
Die Geoinformatik liefert die Basis für Navigationssysteme, 3D-Karten und Nützliches wie Google Earth. Aber auch Länder und Kommunen nutzen sie für die Speicherung von Katasterdaten, Straßen- und Flussverläufen usw.

Sozioinformatik
Das ist der jüngste Spross der Informatik und verknüpft die Sozialwissenschaften mit der Informatik. Wenn Ihnen dazu nichts einfällt, sollten Sie ihr Facebook-Profil löschen.

Medizininformatik
Auf den ersten Blick hat auch die Medizin wenig mit Informatik am Hut, aber gerade hier finden sich sehr viele Anwendungen. Neben dem administrativen Einsatz von Computern finden sich in vielen medizinischen Geräten eingebettete Systeme, die diese steuern und überwachen. Außerdem spielt die Verarbeitung von Bildmaterial eine große Rolle, das bei Computertomographen oder MRT-Systems anfällt.

Diese Aufzählung ist bei weitem nicht vollständig, ständig kommen neue Teilgebiete hinzu, weil die Informatik in immer weitere Bereiche des täglichen Lebens Einzug hält. Und diese Durchdringung in alle Gebiete des täglichen Lebens ist letztendlich dafür verantwortlich, dass sich jeder Studierende damit beschäftigen muss.

12.2 Informatik und Gesellschaft

Nachdem wir jetzt erkannt haben, wie sehr unser Leben mit Teilen der Informatik verzahnt ist, müssen wir uns auch mit den daraus resultierenden Folgen beschäftigen. Man spricht von *Technikfolgenabschätzung* oder *Informatik und Gesellschaft.*

Niemand führt Informatik oder informatikgestützte Methoden ein weil es ihm Spaß macht (außer den theoretischen Informatikern vielleicht) . Dahinter stehen grundsätzlich knallharte Interessen, meistens in Form des Begriffs Kostensenkung. Man wendet Informatik an, um etwas schneller, besser, schöner oder eben günstiger zu machen.

Rationalisierung
Wenn ein (produktiver) Prozess in irgendeiner Form optimiert werden kann, spricht man von Rationalisierung. Diese ist deutlich älter als die ersten elektronischen Rechenanlagen, als „Erfinder" gilt Henry Ford, der im Jahre 1913 die Herstellung seines legendären Modell T auf Fließbandproduktion[29] umstellte und so viele Handgriffe der Mitarbeiter wegrationalisierte oder vereinfachte. Heute ist die Rationalisierung eine eigenständige Wissenschaft in der Betriebswirtschaftslehre.

Wandel von Arbeitsplätzen
Nahezu alle Maßnahmen der Rationalisierung führen zu einem Wandel der Arbeitsplätze, nicht immer zum Wohle der Arbeitnehmer. Bleiben wir bei Henry Ford, nach Einführung des Fließbandes musste jeder Arbeiter nur noch wenige Handgriffe beherrschen, diese dafür aber sehr schnell. Die Arbeit wurde eintöniger und anstrengender, profitiert hat davon nur Henry Ford (und seine späteren Aktionäre).

Das ist bei Rationalisierung durch Computer oder Informatik nicht anders. Heute entwirft ein Architekt mittels CAD-Programm seine Gebäude und der Plotter druckt die fertigen Pläne aus, nach denen dann das Gebäude errichtet wird. Früher waren dazu noch etliche Bauzeichner nötig, die aus den Plänen des Architekten eine Bauzeichnung erstellten. Heute ist diese Berufsgruppe quasi nicht mehr existent.

Es gibt auch Fälle, in denen die Rationalisierung nicht direkt in die Arbeitslosigkeit führt. Aber diese Fälle sind nicht so häufig, wie es uns gerne von den Vertretern der Arbeitgebern erzählt wird.

Wandel von beruflichen Anforderungen
Am deutlichsten sieht man diesen Wandel bei der industriellen Produktion. Wo früher noch Menschen am Band gestanden haben, stehen heute Roboter und erledigen die selben Handgriffe schneller, präziser und billiger. Ein Roboter macht keine Pausen, ist nicht krank, braucht keinen Urlaub und beschwert sich auch nicht beim Betriebsrat.

Natürlich benötigt man Mitarbeiter, die die Roboter konstruieren, programmieren und warten. Und selbstverständlich benötigt man jemanden, der den Produktionsprozess überwacht und zur Not eingreifen kann, wenn es an irgendeiner Stelle mal klemmt, aber das war es dann auch schon. Wer heute einen Blick in die Produktion eines namhaften Darmstädter Pharmaunternehmens wirft, sieht dort keine Menschen mehr, und das hat nichts mit den Hygienevorschriften zu tun.

Diese wenigen Menschen müssen aber qualitativ besser ausgebildet sein wie die Arbeiter in der Produktion vorher. Reichte vorher eine Anlernphase von wenigen Wochen, um die Handgriffe an einer Station des Fließbandes zu beherrschen muss man heute Techniker, Meister oder sogar Ingenieur sein, um eine Produktionsstraße zu überwachen. Man könnte ganz zynisch auch behaupten, dieser Trend passt gut zur demographischen Entwicklung in die Industrienationen, löst aber nicht die Finanzierungslücken in unserem Rentensystem sondern füllt nur die Schatullen der Aktionäre.

Disclaimer: falls der geneigte Leser es noch nicht bemerkt hat, der Autor dieser Zeilen ist Mitglied einer Gewerkschaft und als Personalvertreter (Betriebsrat) aktiv

12.3 Aufbau von Rechenanlagen

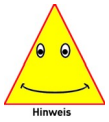
Nachdem wir uns jetzt mit den Begriffen und Gebieten der Informatik beschäftigt haben, können wir uns jetzt dem handfesten Gegenständen dieser Wissenschaft zuwenden. Da ist zunächst einmal die Hardware selbst, also all das, was man anfassen und kaputt machen kann.

29 Es gab auch vor 1913 schon Fließbänder (z.B. bei Bahlsen), aber Henry Ford hat damit eine Revolution ausgelöst und wird daher immer als Erfinder genannt

12.3.1 Hardware

Die Computertechnologie ist vor allem deshalb so erfolgreich, weil sie ein Musterbeispiel für ständige Miniaturisierung ist. Mit anderen Worten: im gleichen Volumen bekommt man heute sehr viel mehr Funktionalität.

Das Grundprinzip aller Computer ist übrigens, dass ein Basiselement durch einen äußeren Einfluss seinen Zustand ändert. Warum zwei Zustände dafür eine besonders gute Idee sind, sehen wir später noch. Dieses Basiselement muss eine weitere Bedingung erfüllen, nämlich das verwendete Medium muss in der Lage sein, sich selbst zu steuern. Falls Sie jetzt nur Bahnhof verstehen ist das okay und es wird schnell klar, was gemeint ist, wenn man für Medium den *elektrischen Strom* einsetzt und für *Basis-element* den Schalter. Ein Schalter kann zwei Zustände annehmen, nämlich an und aus, und er kann durch elektrischen Strom geschaltet werden.



In Spezialanwendungen gibt es auch heute noch „Computer" die nicht mit elektrischen Strom arbeiten, sondern Luft (Pneumatik) oder Flüssigkeiten (Hydraulik) als Medium einsetzen. Als Schalter werden dazu spezielle Ventile[30] eingesetzt, die sich aber ganz analog zu ihren elektrischen Kollegen verhalten.

Miniaturisierung, Röhren, Relais, Halbleiter, Transistor, Integrierte Schaltungen

In der Computersteinzeit bestanden diese Schalter aus Elektronenröhren[31], in denen ein Elektronenstrahl von der Kathode zur Anode floss. Eine zusätzliche Elektrode, das Gitter, war in der Lage, diesen Strom zu steuern, also auch an und aus zu schalten. Röhren haben aber ziemlich große Nachteile. Sie benötigen Unmengen an Energie, um ihre Kathoden aufzuheizen, waren groß und empfindlich gegen Stöße und andere äußere Einflüsse. Ein bekannter Vertreter dieses Spezies ist ENIAC (Electronic Numerical Integrator and Computer) aus den Jahren 1942-1946. Er bestand aus knapp 18.000 Röhren, brauchte 1700m^2 Fläche und 174kW elektrische Leistung.

Eine deutliche Verbesserung stellte das elektromagnetische Relais[32] dar, der Stromfluss wurde durch eine bewegliche Metallzunge geschlossen oder geöffnet, dazu musste man lediglich den eingebauten Elektro-magneten mit Strom versorgen. Relais waren deutlich robuster, brauchten wegen der fehlenden Heizung sehr viel weniger Energie und Kühlung und günstig zu bekommen (man nutze nämlich die Relaistechnik der Telefonanlagen, die waren perfekt geeignet). Der aus Bad Hersfeld stammende Bauingenieur Konrad Zuse baute bereits 1941 aus 600 Relais den ersten funktionsfähigen Digitalrechner weltweit, die Z3[33].

Aber die Miniaturisierung ging unaufhörlich weiter. Die Relais wurden durch den Transistor abgelöst, der noch weniger Platz und Energie benötigte. In der nächsten Stufe wurden mehrere (Hunderte oder Tausende) Transistoren in ein Gehäuse gepackt und als Integrierte Schaltung verkauft. Heutige Prozes-soren vereinen mehrere Millionen Transistoren unter einem Gehäuse. Die Miniaturisierung nähert sich aber ihrem Ende, weil die Abstände zwischen zwei (an sich getrennten Bauteilen) auf dem Chip so klein werden, dass Elektronen zwischen diesen Bauteilen wechseln können und damit für Fehlfunktionen sorgen.

30 http://learnchannel.de/de/pneumatik/logische-ventile-steuerkreisen/
https://de.wikipedia.org/wiki/Zweidruckventil

31 https://de.wikipedia.org/wiki/Elektronenr%C3%B6hre

32 https://de.wikipedia.org/wiki/Relais

33 https://de.wikipedia.org/wiki/Zuse_Z3

Unabhängig davon, wie diese elektrischen Schalter technisch aussehen, müssen sie sinnvoll verschaltet werden, um gewisse Funktionen darzustellen. Ich beschränke mich hier auf wenige Grundelemente, wer mehr wissen möchte kann die Vorlesung „Digitaltechnik" besuchen.

12.3.2 Die Basis: Aussagenlogik

Wenn man mehrere solche Basiselemente oder Schalter hat, kann man die ja auf unterschiedlichste Weise verknüpfen. Die dahinter stehende Aussagenlogik ist sehr viel älter als unsere Rechner, sie wurde von George Boole[34], einem englischen Mathematiker bereits Anfang des 19. Jahrhunderts formuliert.

Zum Einstieg belauschen wir einmal diese Unterhaltung zwischen zwei Studierenden vor der Mensa:

„Kommst Du heute Abend vorbei und bringst Getränke mit?"
„Nein, geht leider nicht!"
„Okay, dann soll Peter die Getränke besorgen oder ich mache das selbst"

Die erste Frage enthält zwei Teile, „Kommst Du heute Abend vorbei?" und „Bringst Du Getränke mit?". Beide Fragen haben einen Gehalt, den nennen wir der Einfachheit halber *Aussage*. Und um den Gehalt nicht zu wichtig zu nehmen, kürzen wir die Aussagen ab, Die Aussage A steht für „Vorbei kommen" und die Aussage B für „Getränke mitbringen".

Diese beiden Aussagen werden durch ein **und** verknüpft, sie bilden also eine *Konjunktion*. Die gesamte Konjunktion wird dann erfüllt (=wahr), wenn alle Teilaussagen auch erfüllt (=wahr) sind. Geschrieben sieht das so aus:

$A \wedge B$ (gesprochen: *A und B*)

In unserem Beispiel lautet die (verbale) Antwort „Nein", und dafür kann es mehrere Gründe geben. Es könnte A wahr sein, aber B nicht. Oder umgekehrt, also A ist falsch, aber B ist wahr. Und dann könnten schlicht beide Aussagen unwahr sein. Die Logiker stellen diesen Sachverhalt gerne als *Wahrheitstabelle* dar, eine Sitte, der ich gerne folge. Und auch der Tradition, den Begriff *unwahr* oder *falsch* mit einer 0 (Null) sowie *wahr* oder *richtig* mit einer 1 (Eins) zu beschreiben:

Aussage A	Aussage B	Konjunktion $A \wedge B$
0	0	0
0	1	0
1	0	0
1	1	1

Zur Verdeutlichung habe ich den einzigen Fall grün hinterlegt, bei dem die Konjunktion selbst wahr, also 1 , wird.

Die dritte Zeile beinhaltet auch zwei Aussagen: A = „Peter holt Getränke" und B = „Ich mache das selbst". Diese sind aber mit **oder** verknüpft, das nennt man eine *Disjunktion*. In Logik-Schreibweise

$A \vee B$ (gesprochen: *A oder B*)

34 https://de.wikipedia.org/wiki/George_Boole

Und da sieht die Wahrheitstabelle dann so aus:

Aussage A	Aussage B	Disjunktion $A \vee B$
0	0	0
0	1	1
1	0	1
1	1	1

Hier sind es also drei der vier Fälle, die als Ergebnis der Konjunktion eine wahre Aussage liefern. Übertragen auf unser Beispiel bedeutet das aber auch, dass im günstigen Fall zweimal Getränke geholt werden (das wäre im realen Leben sicher kein ernst zu nehmendes Problem).

Unser Computer ist gedanklich schon einen Schritt weiter und beschränkt sich rein auf die Aussagen, ohne damit etwas Konkretes zu verbinden. Er kennt nur die Werte 0 (kein Strom) und 1 (Strom), mehr nicht.

Um „richtig" schalten und walten zu können, brauchen wir noch ein paar weitere Verknüpfungen. Leicht einsichtig ist die Negation, die den Wahrheitswert einfach umdreht, aus 0 wird 1 und aus 1 wird 0. Die Tabelle dafür spare ich mir aber, dargestellt wird die Negation als $\overline{A}$ oder $\neg A$

Auch häufig anzutreffen sind Antivalenz und Äquivalenz, gelegentlich auch die Implikation. Hier die Wahrheitstabelle:

Aussage A	Aussage B	Implikation $A \rightarrow B$	Äquivalenz $A \equiv B$	Antivalenz $A \oplus B$
0	0	1	1	0
0	1	1	0	1
1	0	0	0	1
1	1	1	1	0

Die *Implikation* ist immer dann wahr, wenn entweder A falsch oder B wahr ist. *Äquivalenz* und *Antivalenz* sind das jeweilige Gegenteil zueinander, die Äquivalenz ist dann wahr, wenn beide Eingänge den identischen Wert haben, die Antivalenz immer dann, wenn die beiden Eingänge unterschiedlich sind. Bei den Technikern hat sich daher für die Antivalenz auch der Begriff *exklusiv-oder* oder XOR und EQUIV für die Äquivalenz eingebürgert.

12.3.3 Die Rechenregeln der boolschen Algebra

Aus der normalen Mathematik kennen Sie diverse Gesetze, etwa Kommutativ-, Assoziativ- oder Distributivgesetz. Was für die unbeschränkten Zahlenräume funktioniert, gilt natürlich erst recht für einen Zahlenraum mit nur zwei Werten. Aber aufgrund des begrenzten Wertevorrates kann man sämtliche Fälle direkt hinschreiben und sich so lange Beweise ersparen.

In den folgenden Formel wird eine weitere, aber sehr häufig eingesetzte Schreibweise verwendet. Das logische UND wird durch eine Multiplikation, das logische ODER durch ein Additionszeichen dargestellt. Konsequent ist dann auch die Verwendung des Minuszeichens für die Negation. Ebenso gebräuchlich ist

das Ausrufezeichen ! für die Negation, da es in vielen Programmiersprachen (auch in C++) dafür verwendet wird. Ich verwende alle Schreibweisen nebeneinander.

Assoziativgesetz
Bei gleichartigen Operatoren spielt die Reihenfolge der Auswertung keine Rolle. Klammern sind dann nicht nötig. Davon gibt es gleich mehrere Versionen:

UND ist assoziativ $(A \cdot B) \cdot C = A \cdot (B \cdot C) = A \cdot B \cdot C$
ODER ist assoziativ $(A + B) + C = A + (B + C) = A + B + C$
XOR ist assoziativ $(A \oplus B) \oplus C = A \oplus (B \oplus C) = A \oplus B \oplus C$
EQUIV ist assoziativ $(A \equiv B) \equiv C = A \equiv (B \equiv C) = A \equiv B \equiv C$

Kommutativgesetz
Auch die Vertauschung der Eingänge ist problemlos möglich und erlaubt.

UND ist kommutativ $A \cdot B = B \cdot A$
ODER ist kommutativ $A + B = B + A$
XOR ist kommutativ $A \oplus B = B \oplus A$
EQUIV ist kommutativ $A \equiv B = B \equiv A$

Distributivgesetz
Mischt man die Operatoren, kommt man zu zwei Varianten

$$A \cdot (B + C) = A \cdot B + A \cdot C \quad \text{und} \quad (A + B) \cdot C = A \cdot C + B \cdot C$$

$$A + B \cdot C = (A + B) \cdot (A + C) \quad \text{und} \quad A \cdot B + C = (A + C) \cdot (B + C)$$

Äquivalenz und Antivalenz sind nicht distributiv!

Neutralität, Komplement und anderes
Auch wenn es trivial aussieht (und ist), der Vollständigkeit halber sei es trotzdem erwähnt

$$A \cdot 1 = 1 \cdot A = A$$
$$A + 0 = 0 + A = A$$
$$A \cdot \overline{A} = 0$$
$$A + \overline{A} = 1$$
$$A \cdot 0 = 0 \cdot A = 0$$
$$A + 1 = 1 + A = 1$$
$$A \cdot A = A$$
$$A + A = A$$
$$A \cdot (A + B) = A$$
$$A + A \cdot B = A$$

Wenn Sie bei der ein oder anderen Formel ins Grübeln geraten, kein Problem. Erstellen Sie einfach eine passende Wertetabelle und überprüfen Sie das Ergebnis.

Das Beste kommt immer zum Schluss, ich konnte mich zwischen zwei Themen nicht entscheiden, also wird der Schluss einfach etwas länger.

Dualität
Ein boolscher Term, der nur aus ODER, UND und den Zahlen 0 und 1 besteht, bleibt wahr, wenn alle ODER und UND sowie alle Nullen und Einsen ausgetauscht werden. Würfeln Sie sich einen entsprechenden Term aus und überprüfen Sie diese fundamentale Aussage anhand einer Wahrheitstabelle.

De Morgan'schen Gesetze
De Morgan[35]hat das Dualitätsprinzip noch etwas weiter gefasst und die Negation (NOT) mit einbezogen.

$$\overline{A \cdot B} = \overline{A} + \overline{B} \quad \text{sowie} \quad \overline{A + B} = \overline{A} \cdot \overline{B}$$

Natürlich klappt das auch mit mehr als zwei Operanden...

Komplexe Ausdrücke lassen sich mittels dieser Rechenregeln mehr oder weniger leicht umformen und vereinfachen. Das wird dann relevant, wenn es sich um echte Probleme handelt, die dann auch als Schaltung realisiert werden müssen. Hier spart jede Vereinfachung bares Geld.

12.3.4 Leichter geht's mit KV

Im vorherigen Abschnitt haben wir jede Menge Rechenregeln für logische Ausdrücke kennen gelernt, damit es übersichtlich bleibt, meist nur mit zwei oder drei Aussagen A, B und C. Natürlich hindert uns niemand daran, auch mehr als zwei oder drei Aussagen miteinander zu verknüpfen. Die Kunst besteht dann darin, die komplexen Wahrheitstabellen so zu vereinfachen, dass man die Aussagen auch tatsächlich in Elektronik gießen und benutzen kann. Es gibt Leute, die lösen solche Vereinfachungen durch scharfes Hinsehen, aber nicht jeder (auch ich nicht) ist mit dieser Gabe gesegnet. Schauen wir uns doch mal eine exemplarische Wahrheitstabelle mit drei Aussagen an

A	B	C	f_1 (A,B,C)	f_2 (A,B,C)	f_3 (A,B,C)	f_4 (A,B,C)
0	0	0	1	1	0	0
0	0	1	0	1	0	1
0	1	0	0	0	1	1
0	1	1	1	1	1	1
1	0	0	1	1	0	0
1	0	1	0	1	1	1
1	1	0	1	0	0	1
1	1	1	1	1	1	0

Die ersten drei Spalten stellen unsere drei Aussagen A, B und C in allen acht denkbaren Kombinationen (2^3=8) dar. Die vier rechten Spalten stellen vier voneinander unabhängige Funktionen f_1 bis f_4 diesen drei Eingangsgrößen dar. Im folgenden betrachten wir aber nur die farbig hinterlegte Funktion f_1, die drei verbleibenden Funktionen sind zur Übung....

Disjunktive Normalform (DNF)
Eine vollständige Beschreibung einer solchen Funktion erhält man, wenn man jede Zeile, in der die Funktion eine 1 als Ergebnis liefert mittels ODER verknüpft (Disjunktion). Jedes Element dieser Disjunktion ist wiederum eine UND-Verknüpfung (Konjunktion) der drei Eingangsgrößen. Für die Funktion f_1 sieht das so aus:

35 https://de.wikipedia.org/wiki/Augustus_De_Morgan

$$f_1 = !A!B!C + !ABC + A!B!C + AB!C + ABC$$

Diese Darstellung nennt man disjunktive Normalform, weil die einzelnen Terme mittels Disjunktion (ODER) verknüpft sind. Natürlich gibt es auch eine konjunktive Normalform, die aber im täglichen Leben keine so große Rolle spielt und daher hier einfach fehlt.

Der erste und der dritte Term (in blauer Schrift) unterscheiden sich nur im fehlenden NOT vor dem A. Mit normaler Sprache ausgedrückt spielt also der Wert der Aussage A für diese beiden Terme keine Rolle, einer der beiden ist auf jeden Fall 1, egal welchen Wert A hat. Also kann das A bei diesen beiden Termen entfallen und der gesamte Term vereinfacht sich zu

$$f_1 = !B!C + !ABC + AB!C + ABC$$

Analog gilt dies auch für die beiden letzten Terme, hier kommt die Aussage C einmal mit und einmal ohne NOT vor, kann also auch entfallen:

$$f_1 = !B!C + !ABC + AB$$

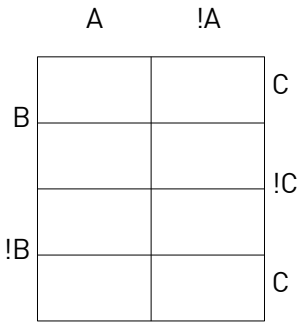
Alternativ hätte man auch den zweiten und letzten Term (wegen A und !A) zusammenfassen können:

$$f_1 = !B!C + AB!C + BC$$

Zur Übung können Sie ja die anderen drei Funktionen entsprechend vereinfachen. Falls Sie das mit mehreren Personen zusammen tun, wundern Sie sich nicht über unterschiedliche Ergebnisse, es gibt immer mehrere richtige Lösungen, die sie mittels der Gesetze der boolschen Algebra ineinander überführen können. Oder sie erstellen Wahrheitstabellen und überprüfen ihre Vereinfachung damit.

KV-Diagramme

Vor dieser Problematik standen schon einige Leute, es ist also kein Wunder, dass sich schlaue Köpfe Gedanken darüber gemacht haben, wie man das vereinfachen kann. Zwei dieser Köpfe waren die amerikanischen Informatiker Edward Veitch und Maurice Karnaugh. Sie haben eine grafische Methode entwickelt, mit der solche Umformungen und Vereinfachungen den Schrecken verlieren. Die Basis bildet eine Matrix, die die aufzustellenden Terme abbildet.



Die acht Felder der Tabelle stellen jeweils genau eine Zeile unserer Wahrheitstabelle dar, in der folgenden Grafik sind diese Terme mal eingetragen:

	A	!A	
B	ABC	!ABC	C
	AB!C	!ABC	!C
!B	A!B!C	!A!B!C	
	A!BC	!A!BC	C

Trägt man jetzt in die Felder, die in der Wahrheitstabelle zu einer 1 führen genau diese 1 auch ein, kommt man für die Funktion f1 zu diesem Ergebnis:

	A	!A	
B	1	1	C
		1	!C
!B	1	1	
			C

Bereits optisch wird hier klar, wo es etwas zu vereinfachen gibt, nämlich immer dann, wenn zweimal eine 1 nebeneinander steht. Die beiden gelb hinterlegten Termen vereinfachen sich zu BC, die beiden orange hinterlegten zu !B!C. Auch vertikal gibt es Einsparpotenzial (nicht markiert), die rechte Spalte bietet zwei Möglichkeiten, entweder !AB oder !A!C. Der Gesamtterm vereinfacht sich also zu BC+ !B!C+!AB. Damit kommt jede 1 in mindestens einem Term vor, die Lösung ist also vollständig (Eine 1 darf durchaus in mehreren Blöcken verarbeitet werden, man muss im Einzelfall abwägen, wie stark die Vereinfachung dadurch wird).

Achtung: Die Anordnung der Aussagen ist nicht fest geschrieben, die NOT-Variante kann auch mit ihrem Partner getauscht werden, das ändert nichts am Ergebnis, sondern nur an der Darstellung. Ganz wichtig ist es zu wissen, dass dieses Diagramm am Rand jeweils auf der anderen Seite fortgesetzt wird, am unteren Rand geht es also quasi mit der obersten Zeile weiter, auch so können Pärchen gebildet werden.

Findet man sogar quadratische Viererblöcke, reduziert sich die Vereinfachung auf eine Aussage (etwa !B), weil dann zwei Aussagen wegfallen. Und bei einem Achterblock ist schließlich alles egal….

Es gibt auch KV-Diagramme für vier Eingangsgrößen, das sieht dann so aus

	A		!A		
B	AB!C!D	ABC!D	!ABC!D	!AB!C!D	!D
	AB!CD	ABCD	!ABCD	!AB!C!D	D
!B	A!B!CD	A!BCD	!A!BCD	!A!B!CD	
	A!B!C!D	A!BC!D	!A!BC!D	!A!B!C!D	!D
	!C	C		!C	

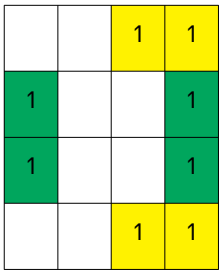
Beim Aufstellen von KV-Diagrammen gelten folgende Regeln:

- An jeder Kante steht nur eine Variable in normaler und negierter Form.
- Bei mehr als zwei Variablen müssen gegenüberliegende Kanten unterschiedlich aufgeteilt sein.
- Gegenüberliegende Kanten sind als benachbart anzusehen.

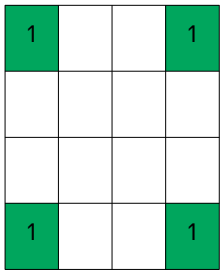
Die konkrete Aufteilung der Eingangsvariablen auf die Kanten ist also durchaus flexibel möglich, das ändert aber lediglich die Optik des Diagramms, nicht das Ergebnis (das wäre ja auch ziemlich doof).

Außerdem ist schnell klar, dass fünf Eingangsvariablen so nicht mehr darstellbar sind, das Diagramm hat ja nur vier Kanten (das wird durch eine dritte Dimension gelöst, die dann in mehrere Diagramme auf einer Ebene aufgelöst wird).

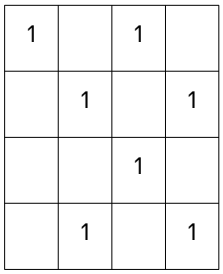
Der letzte Punkt der Regeln wird oft übersehen, wenn man nach zusammenhängenden Blöcken sucht. Die folgenden Bilder zeigen das exemplarisch



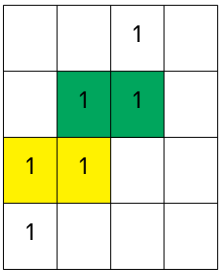
2 Viererblöcke über die Kanten hinweg



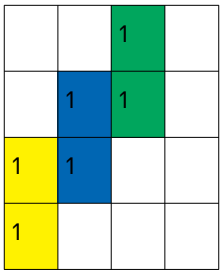
Auch das ist ein Viererblock!



Hier gibt es leider nichts zu optimieren



Es gibt sehr oft mehr als eine Variante der Optimierung



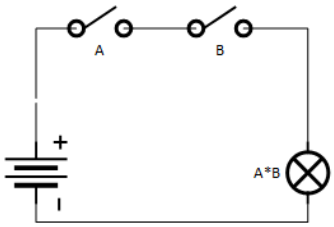
So schön und anschaulich die KV-Diagramme auch sind, ab 5 Eingangsvariablen sind sie nicht mehr praktisch verwendbar. Das macht aber nichts, denn auch dafür gibt es eine Lösung, den Algorithmus von Quine und McCluskey[36]. Das Prinzip dahinter sucht zunächst nach Zweierblöcken, also Termen, die sich nur in einer Eingangsgröße unterscheiden. Wenn die alle gefunden sind, werden diese erneut miteinander verglichen, um wiederum solche zu finden, die sich auch jetzt nur in einer Variablen unterscheiden. Das entspricht den Vierblöcken im KV-Diagramm. Durch ständiges Wiederholen dieses Verfahrens kommt man zu immer größeren Blöcken und am Ende hat man ein optimales Ergebnis. Der Algorithmus ist so gestaltet, dass er sich ideal für die maschinelle Verarbeitung eignet und sehr leicht in eine Programmiersprache übersetzen lässt.

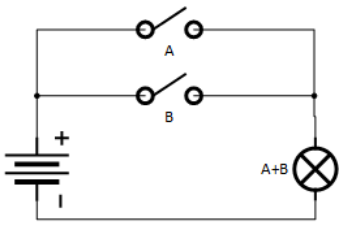
12.3.5 Und warum das Ganze?

Genau wie in der klassischen Mathematik machen es uns diese Formeln möglich, komplexe boolsche Ausdrücke umzuformulieren und vor allem zu vereinfachen. Und solche komplexen Ausdrücke erhält man ganz schnell, wenn man sich einmal an einer Problemstellung aus dem wahren Leben, wie zum Beispiel einer Aufzugsteuerung, versucht. Hier führt jede Vereinfachung der Steuerungslogik zur Einsparung von Bauteilen und reduziert damit die Anfälligkeit für Fehler und Ausfälle.

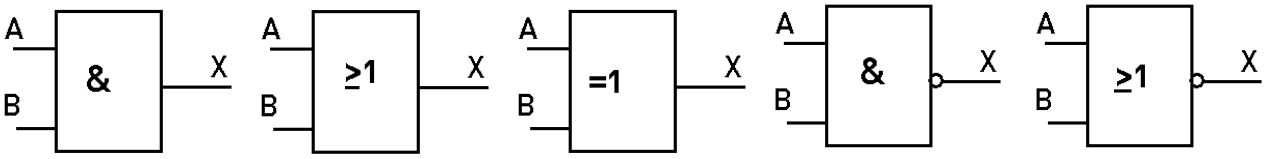
Damit ist unser eher theorielastiger Exkurs in die boolsche Algebra beendet und wir kehren zurück zur Hardware. Ganz unten basiert unser Rechner ja auf Schaltern, die irgendwie aus Transistoren gebildet werden. Und diese Schalter werden nach den eben vorgestellten der boolschen Algebra miteinander verknüpft. Durch das Betätigen der Schalter durch andere Schalter entsteht unser universell verwendbarer Rechner. Exemplarisch hier die UND- und die ODER-Verknüpfung als Schaltung

36 Eine (sehr mathematische) Beschreibung findet sich in https://de.wikipedia.org/wiki/Verfahren_nach_Quine_und_McCluskey





Okay, die Lampe steht symbolisch als Anzeiger, ob Strom fließt oder nicht. Im Rechner findet man diese Art der Schaltung eher nicht. Und weil es beim Design von logischen Schaltungen auch relativ egal ist, wie die Schalter innen drin funktionieren gibt es eine Norm, die sich auf die Funktion beschränkt. Über die optische Qualität dieser Norm kann man wunderbar streiten, aber anwenden muss man sie trotzdem.



UND
AND

ODER
OR

XOR
XOR

Nicht-UND
NAND

Nicht-Oder
NOR

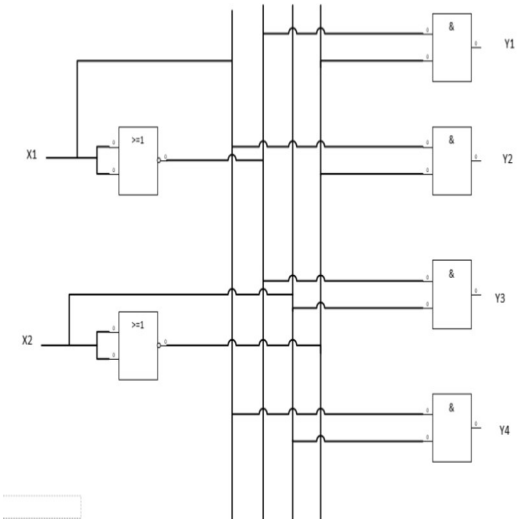
Das sind die üblichen Verdächtigen zum Aufbau auch größter digitalen Schaltungen. Von links nach rechts die UND-Verknüpfung (oder auch AND), am & noch halbwegs eingängig erkennbar. Das ODER-Glied (OR) mit >=1 erschließt sich manchmal nach einigem Nachdenken, aber beim XOR mit =1 braucht man schon sehr viel Phantasie.

AND und OR gibt es zusätzlich in einer NOT-Version, leider nur sehr schlecht am winzigen Kreis am Ausgang zu erkennen. Diese Variante ist in der Realität sehr beliebt, sie benötigt nämlich einen Transistor *weniger* wie die normale Variante ohne Negation.

Zur Übung basteln wir aus diesen Komponenten einen Decodierer. So einen Decodierer benötigt man, wenn man von mehreren Ausgängen genau einen auf 1 setzen möchte, und zwar den, dessen Nummer dual kodiert an den Eingängen anliegt. Als Wahrheitstabelle ist das leichter zu erkennen:

Eingänge		Ausgänge			
x1	x2	y1	y2	y3	y4
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Dazu passt folgende Schaltung:



Die beiden NOR-Gatter links sind als NOT geschaltet, sie invertieren lediglich das anliegende Signal. Damit steht jedes Eingangssignal als Original als auch als negiertes Signal zu Verfügung.

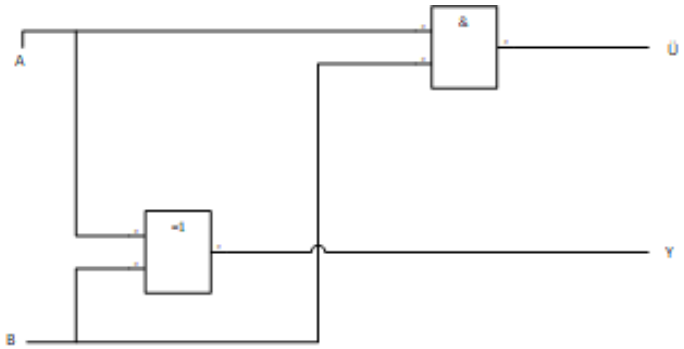
Diese (jetzt vier) Signale werden auf die vier Sammel-schienenn in der Mitte der Grafik aufgeschaltet, von denen

sich dann jedes der vier AND-Gatter die beiden heraus pickt, die auf 1 liegen müssen, damit der zugehörige Ausgang auch auf 1 geht.

Eine solche Schaltung (mit deutlich mehr Ein- und Ausgängen) findet sich zum Beispiel auf unseren Speicherriegeln, um den gerade gewünschten anzusprechen.

Nun ist es nicht das Ziel dieses Skriptes, Sie zu Schaltungsdesignern zu machen, aber mit Blick auf das nächste Kapitel zeige ich Ihnen noch den Halbaddierer, also eine Schaltung, die zwei einstellige Dualzahlen addieren kann

A	B	Y	Ü
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



A und B sind die beiden Ziffern die addiert werden sollen. Das Ergebnis der Addition steht am Ausgang Y zur Verfügung. Sollte es zu einem Übertrag auf die nächste Stelle kommen, geht der Ausgang Ü auf 1.

Ich überlasse es dem geneigten Leser, diesen Halbaddierer zum Volladdierer zu erweitern, so dass er nicht nur die beiden Ziffern A und B summieren kann, sondern auch einen eventuellen Übertrag aus der vorherigen Stelle mit berücksichtigt[37].

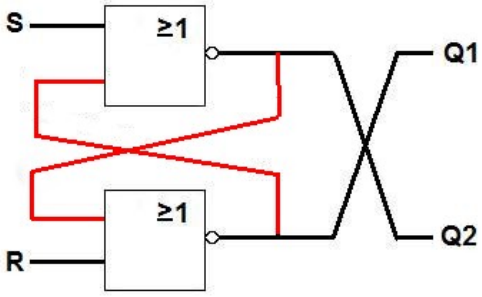
Alle unseren Schaltungen bisher ist gemeinsam, dass sie statisch arbeiten, der aktuelle Eingangszustand führt zu einem definiertem Ausgangszustand. Um unseren Rechner tatsächlich bauen zu können fehlt uns aber noch ein wesentliches Bauteil, wir haben nämlich noch keine Möglichkeit, den Eingangszustand unverändert beizubehalten, auch wenn die Eingangssignale selbst bereits gewechselt haben. Wir brauchen einen Speicher, der sich auf Tastendruck auf 1 setzen lässt, diesen Zustand aber auch dann beibehält, wenn man die Taste wieder los lässt. Erst ein (anderer) Tastendruck soll den Ausgang wieder auf 0 setzen.

Bausteine, die sich so verhalten, nennt man Flip-Flops.

Flip-Flops als Speicher für 1 Bit

Ein Flip-Flop ist eine bistabile Kippstufe, d.h. die Schaltung kippt durch einen äußeren Einfluss von einem stabilen Zustand in den anderen und wieder zurück. Charakteristisch für alle Flip-Flop-Schaltungen ist eine Rückkopplung vom Ausgang auf den Eingang. Im einfachsten Fall sieht das so aus:

37 Die Lösung finden Sie hier: https://de.wikipedia.org/wiki/Volladdierer



Unser RS-Flip-Flop besteht aus zwei NOR-Gattern, es klappt aber auch mit NAND-Gattern. Entscheidend sind die beiden rot gezeichneten Rückkopplungsleitungen, die das Signal am Ausgang Q1 bzw. Q2 wieder auf den jeweils anderen Eingang zurück koppeln.

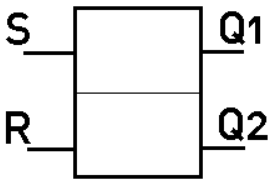
Legt man an den Eingang S (=Set/Setzen) eine 1, geht der Ausgang Q1 auch auf 1 (und Q2 auf 0). Durch die Rückkopplung bleibt dieser Zustand auch dann erhalten, wenn am Eingang schon längst wieder eine 0 anliegt. Unsere 1 wird also gespeichert.

Umgekehrt reicht eine kurzzeitige 1 am Eingang R (=Reset/Rücksetzten) aus, um die Signale am Ausgang umzukehren. Jetzt ist Q1 wieder 0 und Q2 wieder auf 1. Über den Fall, wenn S und R gleichzeitig auf 1 liegen streiten sich die Geister, aber das klammern wir hier mal aus.

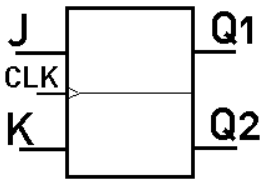
Weitere Bauformen und Ausführungen von Flip-Flops

Unser obiges Flip-Flop ist die einfachste, denkbare Bauform und steht eher symbolisch als Speicher für ein Bit. Für den Bau von Computern benötigt man sehr, sehr viele dieser Flip-Flops, für ein Byte bereits 8 Stück. Für ein Megabyte sind es schon gut 8 Millionen.

Nachteil dieses RS-Flip-Flops ist es zum Beispiel, dass man nicht bestimmen kann, wann genau die Signale am Eingang gültig sind und die Flip-Flops darauf reagieren sollen. Man fügt einen Takteingang hinzu, der dafür sorgt, dass die Eingangsignale genau dann gültig werden, wenn dieser Takteingang auf 1 wechselt (damit ist der Übergang von 0 auf 1 gemeint, man spricht von Flankensteuerung). Diesen Takteingang erkennt man an dem kleinen Dreieck und dem zusätzlichen Eingang (hier CLK für Clock=Takt) auf der linken Seite des Symbols.



RS-Flip-Flop



JK-Flip-Flop

Man kann jetzt mehrere dieser einzelnen Flip-Flops hintereinander oder nebeneinander schalten. Schaltet man die Flip-Flops nebeneinander, erhält man Register zum Speichern von Zahlen, also etwa Speicherzellen oder die Register im Prozessor.

Schaltet man die Flip-Flops hintereinander erhält man ein Schieberegister[38], je nach konkreter Ausführung eignet sich eine solche Schaltung dann als (binärer) Zähler oder Multiplikator. Diese Schaltungen benötigt man auch dann, wenn man parallele Daten in serielle Daten (und umgekehrt) umwandeln muss.

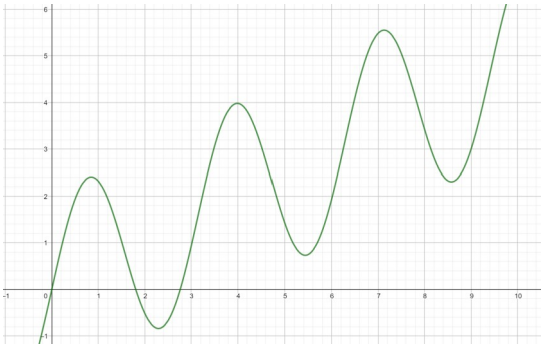
38 https://www.electronics-tutorials.ws/de/sequentielle/schieberegister.html Auf dieser Webseite findet man auch sehr viel weitere Details zu den i her behandelten Themen

Zusammenfassung
Wir haben jetzt das technische Handwerkszeug, um uns aus einer Handvoll Gatterbausteinen einen Rechner zusammen zu bauen, wir können Bits (und damit auch Zahlen) abspeichern. Im nächsten Abschnitt erfahren wir, wie wir aus allgemeinen Größen des realen Lebens diese Zahlen machen und welche Operationen wird im Rechner auf diese Daten anwenden können.

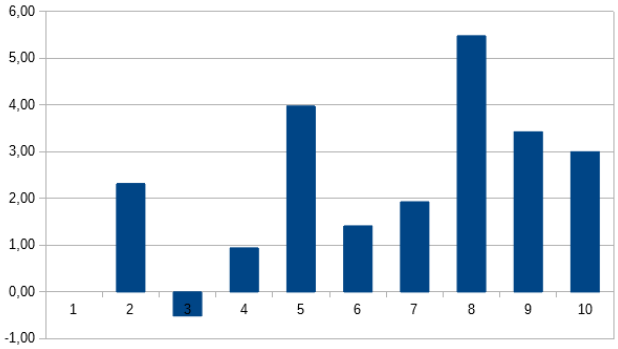
12.4 Das Dualsystem-alles wird zu 0 oder 1

Wenn Sie einmal den Blick vom Bildschirm Ihres Smartphones oder Monitors in die freie Natur wenden und sich an den Physikunterricht in der Schule zurück erinnern, stellen Sie hoffentlich fest, dass alle Vorgänge in der Natur analog ablaufen. Pflanzen wachsen kontinuierlich, der Lauf der Sonne ändert sich stetig, Wasser fließt mal mehr, und mal weniger stark im Bachbett.

Schauen wir uns einmal am Beispiel eines Tons an, wie aus diesem analogen Signal Zahlenwerte für den Rechner entstehen. Klassisch sieht unser Ton auf einem Oszilloskop das etwa so aus:

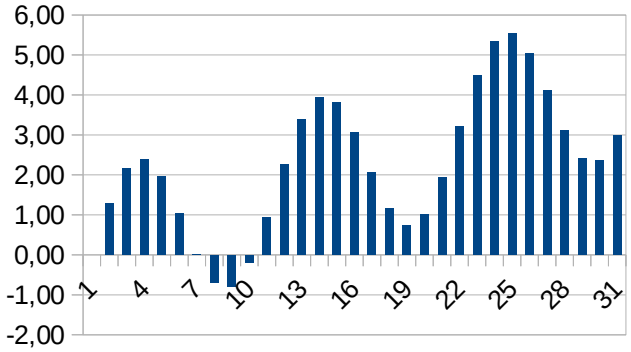


Diese Kurve können wir so nicht in den Rechner bekommen, wir gehen also her und notieren den Funktionswert y in Abhängigkeit des x-Wertes. Dazu nehmen wir in gleichmäßigen Abständen (x-Achse) den Funktionswert (y-Achse) der Funktion auf. Im Rechner erledigt das ein spezieller Baustein, der Analog-Digital-Wandler. Dank moderner Programmen wie einer Tabellenkalkulation geht das recht einfach, beginnen wir mal mit 10 Werten.



Von Ähnlichkeit mit der Original-Kurve ist hier wenig zu erkennen, und das menschliche Ohr würde sich bedanken....

Wir müssen also die *Auflösung* oder auch *Abtastrate* deutlich erhöhen. Gehen wir mal auf 30 Werte, das sieht doch unserer Ausgangskurve schon sehr viel ähnlicher.

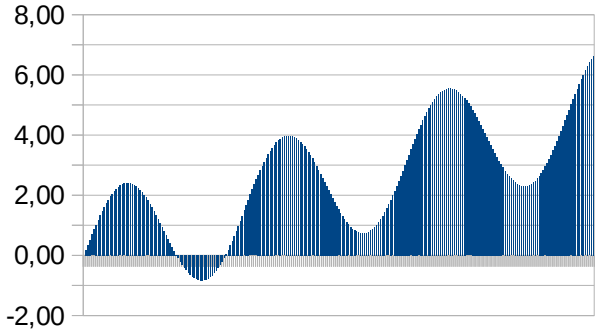


Aber schön sieht anders aus, etwa so. Diesmal mit 256 Werten abgetastet:

Als Besitzer eines MP3-Players haben Sie schon von Abtastraten mit 96, 128, 256 oder noch mehr kHz (=kiloHertz) gehört, eine klassische Audio-CD begnügt sich mit 44,1kHz. Für eine Sekunde Audio fallen damit zweimal (Stereo!) 44100 Zahlenwerte an. Oder anders ausgedrückt, unsere 256 Werte reichen bei dieser Auflösung gerade mal für knapp 6 Millisekunden....

Als Ergebnis unseres kleinen Ausflugs in die Abtasttheorie[39] halten wir fest: Analoge Werte sind zwar präzise, aber schlecht abzulesen und zur Speicherung und Weiterverarbeitung kaum geeignet. Digitale Daten dagegen sind (meistens) gerundet, aber perfekt geeignet zur Verarbeitung und Speicherung in Rechenanlagen. Und diese Vorteile überwiegen bei weitem.

Computer arbeiten seit einigen Jahrzehnten bereits mit elektrischem Strom (die ersten Rechenautomaten funktionierten noch rein mechanisch), und elektrischer Strom kennt idealerweise nur zwei Zustände:



entweder er ist an oder er ist aus. Wenn man also Zahlen und Werte elektrisch speichern möchte, sollte man das auch mit Hilfe des elektrischen Stromes machen und sich auf die Werte an und aus oder 0 und 1 beschränken. Da geht am einfachsten mit dem Dualsystem.

12.4.1 Das Dualsystem

Alle Zahlensysteme der Welt basieren auf dem Stellenwertsystem, der Wert einer Ziffer hängt davon ab, an welcher Stelle der Zahl sie steht. In unserem vertrauten Dezimalsystem erhöht sich die Wertigkeit einer Ziffer mit jeder Stelle weiter links um das zehnfache. Die Zahl[40] 1234_{10} ist also mathematisch die Summe aus $1{\cdot}10^3+2{\cdot}10^2+3{\cdot}10^1+4{\cdot}10^0 = 1000+200+30+4$.

Im Dualsystem verdoppelt sich der Wert einer Ziffer von Stelle zu Stelle, die Zahl 10011010010_2 ist also zu lesen als $1{\cdot}2^{10}+1{\cdot}2^7+ 1{\cdot}2^6+1{\cdot}2^4+1{\cdot}2^1$, da der Faktor immer die 1 ist geht das auch kürzer als $2^{10}+2^7+ 2^6+2^4+2^1$, in Dezimalschreibweise 1024+128+64+16+2 und, welch Zufall!, das ergibt 1234_{10}.

Und weil das für den Menschen so schwer zu lesen ist, fasst man mehrere Stellen zu einer neuen Ziffer zusammen. Bleibt man dabei sinnvollerweise in den Zweierpotenzen, bieten sich drei oder vier Stellen an. Drei binäre Stellen ergeben eine Oktalstelle, vier Binärstellen eine Hexadezimalstelle. Um die 16 verschiedenen Ziffern einer Hexadezimalzahl darstellen zu können muss man neben den bekannten Ziffern 0 bis 9 noch die Buchstaben A bis F zu Hilfe nehmen. A_{16} steht also für 10_{10}, B_{16} für 11_{10} und F_{16} für 15_{10}. Unsere dezimale 1234 wird oktal als 2322 dargestellt und hexadezimal als 4D2. Damit das auch ohne tiefgestellte Basiszahl funktioniert, hat es sich eingebürgert, die Oktalzahlen immer mit einer führenden Null zu beginnen, also 02322 und hexadezimale Zahlen mit 0x, also 0x4D2.

39 https://de.wikipedia.org/wiki/Nyquist-Shannon-Abtasttheorem

40 Die tiefgestellte 10 soll hier angeben, dass die Zahl im Dezimalsystem (=10er-System) dargestellt wird. Analog steht eine 2 für das Dualsystem, eine 8 für das Oktalsystem und eine 16 für das Hexadezimalsystem.

Kommen wir zurück zur Darstellung im Rechner. Eine binäre 1 wird elektrisch als geschlossener und eine binäre 0 als offener Schalter abgebildet. Für die Darstellung unserer Zahl bräuchte wir also (mindestens) 10 Schalter.

Bit und Byte und die Zahlen

Bei der Kapazitätsangabe von Rechnern findet man aber nur ganz selten eine Angabe, wie viele solcher „Schalter" sich in einem Rechner befinden, da ist die Rede von Byte, Kilobyte, Megabyte, Gigabyte und Terabyte. Es stellt sich nämlich sehr schnell heraus, dass man zum Auffinden einer abgespeicherten Zahl wissen muss, wo sich diese befindet, man braucht ihre *Adresse*. Im Rechner genügt dazu die Angabe der Hausnummer, Straßen- und Ortsnamen findet man dort selten. Man muss also die Schalter durchgehend nummerieren, bei Abermilliarden von Schaltern kein Vergnügen. Man macht es sich dabei etwas einfacher und nummeriert nur jeden 8., 16. 32. oder 64. Schalter. Unter dieser *Adresse* bekommt man dann aber auch nicht mehr nur einen Schalter, sondern auch gleich eine ganze Gruppe von 8, 16, 32 oder 64. Und von Schaltern spricht auch niemand sondern immer von Bit. 8 Bit ergeben ein Byte, 16 Bit ein Wort, 32 Bit ein Doppelwort und 64 Bit ein Quadwort.

Man baut die Kapazitätsangaben daher auf der kleinsten Einheit auf und spricht dann von Byte, Kilobyte (1024 Byte), Megabyte (1024 Kilobyte), Gigabyte (1024 Megabyte) usw. Anders als im metrischen System üblich wird also nicht der Faktor 1000 verwendet, sondern 1024. So bleibt alles schön im Dualsystem.

Wir stellen fest: Zahlen werden im Rechner durch ihre äquivalenten Werte im Dualsystem dargestellt. Und weil es bisher nicht explizit erwähnt wurde, dieses Verfahren gilt nur für ganze Zahlen, also das, was die Mathematiker als positive, natürliche Zahlen $\mathbb{N}$ bezeichnen. Negative Zahlen und rationale Zahlen betrachten wir später.

Buchstaben, Zeichen, Worte

Kaum waren Rechner als universelles Werkzeug erkannt, wollte man nicht nur Zahlen, sondern auch Buchstaben und Worte verarbeiten. Allgemein spricht man von Zeichen. Man benötigt eine Vorschrift, die jedem Zeichen eindeutig eine Zahl zuordnet. Und genial wäre es, wenn das jeder Rechner überall auf der Welt in genau der gleichen Weise machen würde. Lassen wir den Rest der Welt zunächst mal außen vor und überlegen uns eine eigene Vorschrift. Unser Alphabet hat 26 Buchstaben, jeweils als Groß- und Kleinbuchstabe, das sind schon mal 52 Zeichen. Dazu kommen die Ziffern von 0 bis 9 (macht 62), eine gute Handvoll Rechen- und Satzzeichen, ein paar Klammern, das nicht ganz so unwichtige Leerzeichen und einige Sonderzeichen. Damit bleiben wir auf eine Zahl von unter 127 verschiedenen Zeichen und kämen so mit 7 Bit für die Darstellung aus. Da unsere kleinste, einzeln adressierbare Einheit aber bereits ein Byte ist, können wir den verfügbaren Platz auch mit weiteren (Sonder-)Zeichen füllen.

Welches Zeichen mit welcher Zahl dargestellt wird, wurde bereits 1963 im *American Standard Code for Information Interchange*, kurz *ASCII*, festgelegt und ist bis heute gültig. Die ersten 32 Zeichen dienen zur Steuerung der Textausgabe auf Bildschirmen und Druckern, darunter fallen Zeichen für den Tabulator oder den Wagenrücklauf und Zeilenvorschub. Daran schließen sich ein paar Zeichen (Klammern und Rechenzeichen) und dann die Ziffern (ab 48) an. Die Großbuchstaben beginnen bei 65 und die Kleinbuchstaben bei 97. Mit 127 Zeichen (also 7 Bit) käme man also aus. Da aber der Rechner intern mit 8 Bit arbeitet, wurde auch der ASCII-Code auf 8 Bit verbreitert, die zusätzlichen 128 Zeichen sind international nicht einheitlich belegt und führen daher immer wieder zu lustigen Effekten auf unterschiedlichen Rechnersystemen. In diesem Bereich finden sich dann auch die deutschen Umlaute und andere internationale Sonderzeichen.

Für die Praxis ist es wichtig zu wissen, dass die Ziffernzeichen direkt hintereinander folgen und auch die Groß- und Kleinbuchstaben alphabetisch aufgereiht hintereinander liegen. Das wird uns im täglichen

Leben noch sehr hilfreich sein. Welcher Zahlenwert sich konkret hinter welchem Zeichen versteckt ist dabei weniger wichtig zu wissen.

Bilder, Grafiken und Pixel

Auch Bilder und Grafiken werden zur internen Darstellung in Zahlen umgewandelt. Die Digitalisierung fängt hier aber schon vorher an. Ein analoges Bild muss zunächst einmal in eine digitale Fläche umgewandelt werden. Dazu überzieht man das Bild mit einem mehr oder weniger feinen Gitter, in jedem Gitterfeld befindet sich dann noch ein Farbpunkt, das Pixel. Aus einem analogen Bild von 4*6cm wird so beispielsweise eine digitale Grafik von 400*600 Pixeln. Auch hier spricht man von Auflösung, je mehr Pixel desto näher kommt man dem Original.

Im zweiten Schritt wird jetzt jedem Pixel ein Farbwert zugeordnet. Besteht das Bild nur aus einer Farbe (zuzüglich dem Hintergrund, also etwa schwarz und weiß), wird eine 1 abgespeichert, wenn das Pixel gefärbt ist und eine 0, wenn es sich um den Hintergrund handelt. Handelt es sich um ein Graustufenbild verwendet man üblicherweise ein Byte pro Pixel, kann also 256 Grauwerte in den Rechner übertragen.

Für farbige Darstellungen gibt es mehrere Möglichkeiten, das Grundprinzip ist aber ähnlich. Man zerlegt das Pixel in seine Grundfarben, etwa Rot, Grün und Blau. Damit kann ein Monitor prima arbeiten und farbenfrohe Bilder anzeigen. Sollen die Daten später auf Papier, bieten sich die Grundfarben Cyan, Magenta und Gelb an, die findet man nämlich in Druckern und Druckmaschinen wieder. Und viele Grafikprogramme arbeiten nach dem HSB-Modell, hier werden Farben nach Farbton (Hue), Sättigung (Saturation) und Helligkeit (Brightness) zerlegt.

Unabhängig vom Verfahren der Zerlegung erhält man drei Farbwerte für ein Pixel mit je einem Byte[41].

Songs, Klänge, Töne

Auch Klänge, Geräusche und Töne müssen durch einen Digitalisierer. Der tastet das analoge Signale mehrere Tausend mal pro Sekunde ab und erzeugt daraus 16 Bit-Werte. Je nach Art der weiteren Verarbeitung wird diese Datenmenge dann durch geschickte mathematische Verfahren weiter komprimiert (Stichwort mp3) oder direkt auf eine Audio-CD geschrieben.

Zusammenfassung

Informationen aus der realen Wert müssen durch technische Maßnahmen in digitale Informationen umgewandelt werden, damit ein Computer damit arbeiten kann. Praktisch passiert das in darauf spezialisierten Bausteinen, Baugruppen oder Geräten. Vorsicht ist geboten, weil bei der Wandlung in den meisten Fällen die Genauigkeit leidet.

Im Rechner können die nun digital vorliegenden Daten dann mittels Programmen bearbeite werden. Welche elementaren Rechenoperation ein Rechner durchführen kann sehen wir im nächsten Abschnitt.

12.4.2 Von dezimal zu dual und zurück-Die Mathematik im Dualsystem

Wir haben bis jetzt schon viel über duale Zahlen gehört und wie man sie einsetzt, aber noch keine Dezimalzahl in eine Dualzahl umgewandelt. Und schon gar nicht mit ihnen gerechnet. das holen wir jetzt nach.

41 Viele professionelle Bildbearbeitungsprogramme nutzen intern für die Berechnung der Bilder auch 10 oder sogar 12 Bit, um Bildfehler durch Rechenungenauigkeiten zu vermeiden

Umwandlung von Dezimal- in Dualzahlen

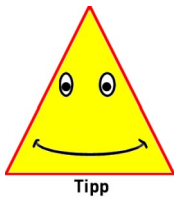
Eine Dualzahl setzt sich ja aus der Summe von Zweierpotenzen zusammen. Diese Zweierpotenzen erhält man am einfachsten, indem man die Dezimalzahl immer wieder durch 2 dividiert. Der Teil, der sich am häufigsten dividieren lässt ist die höchste enthaltenen Zweierpotenz. Betrachten wir das Zeichen * für diesen Zweck. Dieses Zeichen hat den ASCII-Wert 42, wie man aus der ASCII-Tabelle entnehmen kann.

Diese 42 dividieren wir solange durch 2, bis nichts mehr davon übrig bleibt. Ganz wichtig dabei ist, dass wir dabei einen Zeitsprung zurück in die Grundschule machen und uns an die dort übliche ganzzahlige Division mit Rest erinnern! Falls Sie das nicht mehr schaffen, 17 dividiert durch 3 ist 5 Rest 2. Dieser Rest spielt gleich eine entscheidende Rolle!

42 / 2 = 21	Rest 0
21 / 2 = 10	Rest 1
10 / 2 = 5	Rest 0
5 / 2 = 2	Rest 1
2 / 2 = 1	Rest 0
1 / 2 = 0	Rest 1

Dieses Verfahren endet, wenn bei der Division 0 heraus kommt. Und wo ist jetzt unsere Dualzahl? Schauen Sie mal auf die Reste, nur 0 und 1, das sieht doch schon sehr nach dualer Schreibweise aus, nur als Dualzahl geht das noch nicht durch.

Man kann aber einfach die Reste, von unten nach oben gelesen, hintereinander aufschreiben. So erhalten wir die Zahl 101010. Die sechste (und letzte) Division durch 2 entspricht ja der Division durch 2^6 oder 64 und war nicht erfolgreich, weil 42 nicht durch 64 teilbar ist. Die nächst kleiner Zweierpotenz 2^5 (=32) ist aber enthalten, denn 42 durch 32 ist 1 Rest 10. Die Probe beweist die Richtigkeit unseres Verfahrens: $2^5+2^3+2^1=32+8+2=42$.



Dieses Verfahren funktioniert in gleicher Weise für die Umwandlung in jedes beliebige Zahlensystem. Auch in solch exotische wie das 6-er System. Für die Zahl 1024_{10} also:

1024 / 6 = 170 Rest 0
170 / 6 = 28 Rest 0
28 / 6 = 4 Rest 4
4 / 6 = 0 Rest 4
Ergibt 4400_6
Probe: $4*6^3 + 4*6^2 = 4*216 + 4*36 = 864 + 144 = 1024$

Einen Nachteil der dualen Zahlendarstellung auf Papier sehen wir aber hier sehr schnell: die Zahlen werden sehr schnell sehr groß, die 42_{10} braucht bereits sechs Stellen als Dualzahl 101010, die 1024_{10} ist mit 10000000000 zwar schön anzusehen, aber ziemlich schwer zu lesen. Daher fasst man immer vier duale Stellen zusammen und stellt diese dann hexadezimal dar. Unsere 1024 wird damit zu 0100 0000 0000 und schließlich zu 400_{16} oder 0x400. Diese „Umrechnung" erfolgt aber nur für uns menschliche Leser, der Rechner bleibt im Dualsystem.

Addition von Dualzahlen

Jetzt können wir Zahlen von Dezimal- in Dualdarstellung umwandeln, nun bringen wir dem Rechner das Rechnen damit bei. Auch hier hilft der Rücksturz in die Vergangenheit, die schriftliche Addition aus dem zweiten Schuljahr wird benötigt.

Erinnern wir uns zunächst einmal, wie das im Dezimalsystem war und addieren die Zahlen 42 und 49:

```
    4 2
+   4 9
```

```
Übertrag 1
=      9 1
```

Erinnern Sie sich? Man beginnt an der äußersten rechten Stelle und addiert die beiden Ziffern (hier 9 und 2). Die Summe wird zur letzten Stelle der Summe. Manchmal kommt es vor, dass die Summe größer als 9 wird, dann wird die erste Ziffer (maximal eine 1) als Übertrag bei der nächsten Spalte berücksichtigt. Die zweite Spalte (von rechts) hat jetzt also drei Summanden, 4, 4 und den Übertrag 1. Macht etwa 9. Natürlich kann es auch hier passieren, dass es einen Übertrag gibt, der sorgt dann für eine zusätzliche Spalte.

Übertragen wir das auf das Dezimalsystem (aus taktischen Gründen auf 8 Stellen erweitert)[42]:

```
       0 0 1 0  1 0 1 0
+      0 0 1 1  0 0 0 1
Übertrag     1
=      0 1 0 1  1 0 1 1
```

Alles wie gehabt, auch die Sache mit dem Übertrag läuft hier ganz genauso wie bei den Dezimalzahlen. Werden wir mutiger und wagen uns an die Subtraktion. Dazu brauchen wir aber erst die Darstellung der negativen Zahlen.

Negation

Sie werden sich fragen wo liegt das Problem? Einfach ein Minuszeichen vor die Zahl und fertig ist der negative Wert. Und wie stellen Sie das Minuszeichen in elektrischen Strom dar? Das muss also anders gehen. Wenn unsere Rechenvorschrift für die Addition weiter gelten soll müssen wir ein wenig umformen: Statt Was ergibt 0011 0001 – 0010 1010 (49 – 42) fragen wir, welche Zahl muss ich zu 0011 0001 (=49) addieren, um 0010 1010 (=42) zu erhalten?

```
       0 0 1 1   0 0 0 1
+      ? ? ? ?   ? ? ? ?
+ Ü
       0 0 1 0   1 0 1 0
```

Scharfes Hinsehen und etwas Nachdenken führt zu folgendem Ergebnis:

```
       0 0 1 1   0 0 0 1
+      1 1 1 1   1 0 0 1
+ Ü   1 1 1 1           1
       0 0 1 0   1 0 1 0
```

Der letzte Übertrag (ganz links) fällt einfach unter den Tisch, denn unsere Zahl ist ja nur 8 Stellen breit, mehr nutzen wir nicht. Die duale Darstellung der Zahl -7 lautet also 1111 1001. Vergleichen wir die positive Zahl 7 einmal mit ihrem negativen Pendant:

```
0 0 0 0  0 1 1 1      +7
1 1 1 1  1 0 0 1      -7
```

Überall, wo bei der positiven Zahl eine 0 stand, steht jetzt eine 1. Und umgekehrt. Lediglich die letzte Stelle hält sich nicht an diese Regel, es wurde nämlich eine (duale) 1 addiert, um die negative Zahl zu ermitteln. Dieses Vorgehen nennt man *Zweierkomplement* und die formale Regel sieht so aus:

Das Zweierkomplement einer Zahl erhält man, indem man alle Stellen der Zahl invertiert und dann 1 dazu addiert.

Wenn Sie zur Übung ein paar Zahlen in ihr Gegenteil konvertiert haben, wird Ihnen aufgefallen sein, dass bei den negativen Zahlen immer das erste (linke) Bit gesetzt ist. Und daraus folgt eine wichtige

42 Die Lücke nach jeweils 4 Stellen dient nur der besseren Lesbarkeit und hat keine weitere Funktion

Einschränkung für den darstellbaren Zahlenraum. Mit 8 Bit sind insgesamt 256 verschiedene Zahlenwerte darstellbar, entweder die von -128 bis +127 ODER die von 0 bis 255. Diese Entscheidung muss bei der Definition der Zahl getroffen werden und kann danach nicht mehr geändert werden.[43]

Subtraktion von Dualzahlen

Mit Kenntnis des Zweierkomplements wird die Subtraktion auf eine Addition zurückgeführt und damit zum Kinderspiel. Zunächst wird der Subtrahend (die Zahl hinter dem Minuszeichen) mittels Zweierkomplement in seine Gegenzahl umgewandelt. Danach wird sie zum Minuenden (der Zahl vor dem Minuszeichen) addiert. Als Ergebnis erhält man die Differenz:

Als Beispiel berechnen wir $66_{10} - 25_{10}$, in 8-Bit 0100 0010 – 0001 1001. Das Zweierkomplement von 0000 1101 ergibt sich zu 1110 0111. Und die Rechnung sieht dann so aus:

```
          0 1 0 0   0 0 1 0
+         1 1 1 0   0 1 1 1
+ Ü   1 1           1 1
          0 0 1 0   1 0 0 1
```

Und das ist schnell überprüft $2^5 + 2^3 + 2^0 = 32 + 8 + 1 = 41$. QED[44].

Multiplikation von Dualzahlen

Wenn Sie es sich einfach machen wollen, führen Sie die Multiplikation einfach auf eine wiederholte Addition zurück. Das Rechnen soll ja sowieso eine Maschine übernehmen, da fällt das dann überhaupt nicht mehr auf. Doch, spätestens dann, wenn viel gerechnet wird und die Laufzeiten des Programms dann allmählich sehr lang werden.

Aber wie schon mehrfach in diesem Abschnitt erwähnt, die Grundschüler sind uns hier einfach voraus. Wir müssen unser verschüttetes Wissen erst mühsam wieder hervor holen.Tun wir das:

2	**4**	*	**7**	**5**
	1	6	8	
+		1	2	0
Ü		1		
=	**1**	**8**	**0**	**0**

Man multipliziert also den ersten Faktor mit jeder Stelle des zweiten Faktors, schreibt die Ergebnisse spaltentreu untereinander und addiert diese dann. Leere Felder werden dabei als 0 betrachtet[45]. Das geht natürlich auch dual:

1	**1**	**0**	**0**	**0**	*	**1**	**0**	**0**	**1**	**0**	**1**	**1**
		1	1	0	0	0						

43 In C++ wird standardmäßig mit Vorzeichen gerechnet. Um vorzeichenlos zu arbeiten muss der Datendeklaration ein `unsigned` vorangestellt werden.

44 q.e.d.= (lat.) quod erat demonstrandum = was zu beweisen war

45 Achtung: hier werden aus Darstellungsgründen die führende Nullen nicht aufgeführt, wir nutzen hier (mindestens) 24 Bit, um die Zahlen darzustellen. In 8-Bit wäre die Multiplikation schlicht nicht machbar.

+			0	0	0	0	0				
+				0	0	0	0	0			
+					1	1	0	0	0		
+						0	0	0	0	0	
+							1	1	0	0	0
+								1	1	0	0	0
Ü			1	1	1	1	1				
=	**0**	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**

Auch hier wird die Multiplikation auf eine Addition zurück geführt, aber deutlich effektiver. Man benötigt nämlich nur noch genauso viele Additionen, wie der zweite Faktor Stellen hat. Und wenn man richtig hinschaut, benötigt man sogar nur so viele Additionen wie der zweite Faktor 1 hat (grün markiert).

Eine Besonderheit gibt es, wenn der zweite Faktor eine Zweierpotenz ist, dann reicht es aus, den ersten Faktor um die entsprechende Anzahl Nullen zu verlängern. Aber das ist im Dezimalsystem ja auch so....

Division von Dualzahlen

Dreht man die Multiplikation einfach um, kommt man zur Division. In erster Näherung kann man auch hier einfach den Divisor so lange vom Dividenden subtrahieren, bis nichts mehr da ist. Und genau wie bei der Multiplikation ist das auf den zweiten Blick keine gute Idee.

1	**0**	**0**	**1**	**0**	**1**	**1**	**/**	**1**	**1**	**0**	**1**	**=**	**1**	**0**	**1**	**R**	**1**	**0**	**1**	**0**
-	1	1	0	1																
		1	0	1	1															
	-				0															
		1	0	1	1	1														
		-	1	1	0	1														
		R	1	0	1	0														

Man dividiert die führenden Stellen des Dividenden durch den Divisor und notiert das Ergebnis. Der Rest fällt unter den Tisch, die Anzahl der benötigten Stellen wird so lange erhöht, bis die Division ausgeführt werden kann (alternativ erhält man führende Nullen als Ergebnis).

Dann wird das Ergebnis mit dem Divisor multipliziert und stellentreu von den verwendeten Stellen des Dividenden subtrahiert. An dieses Ergebnis fügt man die nächste Stelle des Dividenden an, man holt also eine weitere Stelle von oben herunter (grün) und dividiert diese neue Zahl wieder durch den Divisor. Sollte dabei eine 0 heraus kommen, muss diese aber notiert werden, es handelt sich ja nicht mehr um eine führende 0. Das Ganze wiederholt man, bis alle Ziffern des Dividenden verarbeitet wurden. Am Ende bleibt ggfs. ein Rest.

Dazu noch zwei Bemerkungen: im Beispiel wurde die Subtraktion auch als Subtraktion notiert und nicht das Zweierkomplement dargestellt. Gerechnet wird natürlich über das Zweierkomplement, das würde aber den Rechenweg völlig verschleiern. Und da bei der Division von dualen Zahlen immer nur 1 oder 0 herauskommen kann, ist die Division selbst in Wirklichkeit auch eine Subtraktion. Prüfen Sie es nach!

In vielen Problemstellungen interessiert neben dem Ergebnis der Division auch der Rest. Fast alle Programmiersprachen haben daher zwei Rechenoperationen an Bord, die klassische Division (meist mit

dem / als Rechenzeichen) und den Modulo-Operator (oft das % als Rechenzeichen), der genau den Rest der Division liefert. Entsprechende Aufgaben werden Ihnen im Praktikum begegnen.

12.4.3 Mit Punkt oder Komma-nicht ganz ganze Zahlen

Nun sind Sie dem Grundschulalter entwachsen und wissen, dass es außer den ganzen Zahlen auch andere gibt, die reellen oder rationalen Zahlen. Ohne uns jetzt um die konkreten mathematischen Unterschiede zu kümmern, sprechen wir umgangssprachlich von Kommazahlen. Ich persönlich bevorzuge den Begriff *kaputte Zahlen*, weil er exakt den Unterschied zu den ganzen Zahlen beschreibt.

Reelle Zahlen kommen immer dann zum Einsatz, wenn es um sehr große oder sehr kleine Zahlenwerte geht, bevorzugt wird dann die Schreibweise in wissenschaftlicher Exponentialschreibweise. Eine solche Zahl setzt sich zusammen aus der Mantisse, dem vorderen Teil und einem Exponenten. Die Basis wird nicht angegeben, aber da kommt nur die 10 in Frage. Die Darstellung erfolgt also in der Form Zahl=Mantisse*$10^{Exponent}$. Die Mantisse ist für die Genauigkeit verantwortlich, der Exponent legt dann nur noch fest, wie viele Nullen davor (negativer Exponent) oder dahinter (positiver Exponent) stehen. Den E-Technikern unter Ihnen dürfte die Zahl $-1{,}6021766208 \cdot 10^{-19}$ bekannt vorkommen, die Ladung eines Elektrons (in Coulomb). Ohne Exponentialschreibweise müssten Sie die als 0,000000000000000000016021766208 angeben. Die Frage nach der besseren Lesbarkeit kann jeder für sich selbst beantworten.

Normal ist 754

Aber auch solche Zahlen müssen intern im Rechner irgendwie als Dualzahl(en) dargestellt werden. Dabei bietet es sich an, Mantisse und Exponent getrennt zu betrachten, dazu dann noch ein oder zwei Bit für die Vorzeichen (von Mantisse und Exponent). Nehmen wir mal an, wir hätte eine bestimmte Anzahl von Bits zur Verfügung, wie viele Bit sollte man dann für die Mantisse verwenden und wie viele für den Exponenten? Man muss bei dieser Entscheidung abwägen, ob man die Genauigkeit möglichst hoch haben möchte oder lieber den Wertebereich. Nur gut, dass es das *Institute for Electrical and Electronics Engineers* gibt, das diese Entscheidung einmal für uns alle getroffen und in der Empfehlung[46] IEEE-754 festgeschrieben hat. Festgelegt wurde dies gleich zweimal[47], einmal für einfach genaue Zahlen (32 Bit, in C++ `float`) und für doppelt genaue Zahlen (64 Bit, `double`). Die folgende Tabelle zeigt die Details (basierend auf dem Wikipedia-Artikel):

Typ	Größe	Exponent	Mantisse	Wert des Exponenten	Biaswert	Dezimal-stellen	kleinster Wert	größter Wert
single/ float	32 Bit	8 Bit	23 Bit	-126 .. 127	127	7 .. 8	2^{-126} $\approx 1*10^{-38}$	$(1-2^{-24})*2^{128}$ $\approx 3^{\cdot 1038}$
double	64 Bit	11 Bit	52 Bit	-1022 .. 1023	1023	15 .. 16	2^{-1022} $\approx 2*10^{-308}$	$(1-2^{-53)})*2^{1024}$ $\approx 2 \cdot 10^{308}$

Die Tabelle erfordert noch etwas Hintergrundinformationen. So wird die Mantisse *normalisiert* dargestellt, d.h. die Vorkommastelle der Mantisse in dualer Schreibweise muss eine 1 sein. Und wenn das so ist, kann man die nämlich im Rechner einfach weglassen, ein Bit gewonnen. Nebenbei stellt dieses Vorgehen sicher, dass die Mantisse in der höchstmöglichen Genauigkeit abgespeichert wird.

46 Das ist in der Tat eine Empfehlung, ganz analog zur DIN. Niemand wird gezwungen sich daran zu halten, aber jeder der es nicht tut, hat ein Problem

47 In Wirklichkeit wurden sechs Zahlenformate festgelegt, siehe: https://de.wikipedia.org/wiki/IEEE_754

Für den Exponenten hat man sich einen anderen Trick überlegt, eine schlichte Darstellung im Zweierkomplement scheidet aus, weil die Zahl 0 dann nicht mehr exakt dargestellt werden kann. Also wird der Exponent als vorzeichenlose Zahl betrachtet, zum Ermitteln des echten Wertes wird der *Bias* subtrahiert. Ein abgespeicherter Exponent von 130 bedeutet bei einem Bias von 127 also in Wirklichkeit einen Exponenten von 3.

Ein paar weitere Feinheiten gehen aus der Tabelle auch nicht hervor. Sind alle Stellen von Exponent und Mantisse Null spielt das Vorzeichenbit keine Rolle und es wird mit 0 gearbeitet. Ist nur der Exponent 0 und die Mantisse nicht, handelt es sich um ein Zwischenergebnis, welches nicht normalisiert ist.

Besteht der Exponent aus lauter Einsen hängt der Wert der Zahl von der Mantisse ab. Ist die Null, bedeutet das plus oder minus Unendlich (je nach Vorzeichenbit), ist sie ungleich Null, ist der Wert NaN (Not a Number)[48].

Und was hat man jetzt von diesem Aufwand? Dazu muss man sich die Rechengesetze für Potenzen ins Gedächtnis rufen. Da gilt etwa: Potenzen zur gleiche Basis werden multipliziert, indem man die Exponenten addiert und die Mantissen multipliziert.

Bei anderen Rechenoperationen muss man unter Umständen erst dafür sorgen, dass die Exponenten der Operanden gleich sind. Dazu muss die Mantisse des kleineren Wertes *denormalisiert* werden, das Komma wird also verschoben und führende Nullen eingefügt. Nachteil: die vorne eingeschobenen Nullen schieben hinten Stellen raus, die Genauigkeit sinkt also. Denn diese Stellen fehlen spätestens dann, wenn der Wert wieder normalisiert wird. So schnell werden aus Einsen Nullen.

Ungenauigkeit als Programm

Die Darstellung nach IEEE-754 hat aber noch andere Effekte, die ich kurz erwähnen möchte. Auf den ersten Blick nicht zu erkennen ist, dass die Verteilung der Zahlen über den Wertebereich alles andere als gleichmäßig ist. Zwischen der Zahl $1{,}0*2^{87}$ und $1{,}00000000000000000000001*2^{87}$ gibt es nur ein Bit Unterschied in der Mantisse, weniger geht nicht. Mathematisch bedeutet das aber einen Abstand von 2^{64} zwischen den beiden Werten, oder im Dezimalsystem 18.446.744.073.709.551.616, eine Zahl, die ich nicht in Worte fassen kann[49]. Reizt man dieses Spiel bis zum maximal möglichen Exponenten 126 wird der Abstand zwischen zwei benachbarten Zahlen noch um ein paar Zehnerpotenzen größer, die Zahl hat 32 Stellen.

Aber auch Denormalisieren und Normalisieren kann heftige Auswirkungen auf die Rechengenauigkeit haben. Die ganz einfach erscheinende Aufgabe 0,2 + 1024 -1024 liefert im Dezimalsystem eine klare 0,2. Rechnet man das Ganz aber in 32 Bit Arithmetik nach IEEE-754, sieht das schon ganz anders aus. Der Wert 0,2 stellt nämlich einen unendlichen Bruch im Dualsystem dar, ähnlich dem 1/3 im Dezimalsystem. $0{,}2_{10}$ = 0,0011001100110011001... Zur internen Darstellung muss der Wert normalisiert werden, also muss eine 1 vor das Komma, das ergibt $1{,}100110011001..*2^{-3}$. Intern ergibt sich folgende Darstellung:

0	0	1	1	1	1	1	0	0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	1

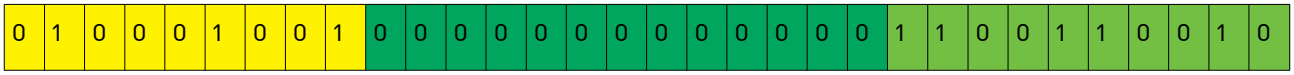
Gelb hinterlegt ist der Exponent, der wegen des Bias von 127 den Wert -3 + 127 = 124 hat. Grün ist die Mantisse, die führende 1 vor dem Komma ist kraft IEEE-754 ja gesetzt und fehlt daher in der Darstellung, dafür wurde die letzte Stelle (ganz rechts) zur 1, die wurde nämlich schlicht gerundet.

---

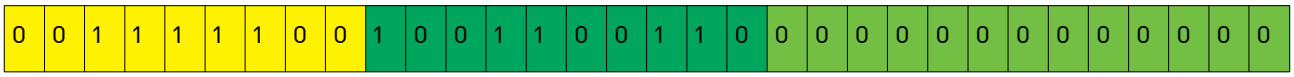
48 $\pm\infty$ entsteht z.B. bei der Division von ±1 / 0. NaN wäre das Ergebnis von 0 / 0.

49 Das ist in etwa die Zahl der Weizenörner die der Sage nach der Erfinder des Schachspiels von seinem Herrscher als Belohnung für das Spiel erbeten haben soll: auf das erste Feld ein Weizenkorn, auf das nächste zwei usw., immer das doppelte des vorherigen Menge. Siehe https://de.wikipedia.org/wiki/Sissa_ibn_Dahir

Die 1024 ist dagegen pflegeleicht, als Zweierpotenz hat sie die Mantisse Null und den Exponenten 10 + 127 = 137. Um die Zahlen addieren zu können, müssen wir die Exponenten gleich machen, der kleinere Wert ist in der Pflicht und wird denormalisiert. Der Exponent erhöht sich um 13 von -3 auf +10 (also mit Bias von 124 auf 137), die Mantisse wird um 13 Stellen nach rechts verschoben, damit das mathematisch wieder stimmt (und natürlich muss die 1 vor dem Komma auch eingefügt werden, die darf nicht unter den Tisch fallen):



Jetzt können wir die beiden Operanden addieren, einfach die Mantissen addieren. Der zweite Rechenschritt ist die Subtraktion des dritten Operanden, zufällig auch 1024. Jetzt haben beide Operanden bereits den gleichen Exponenten, man muss also nichts mehr denormalisieren und kann direkt die beiden Mantissen voneinander subtrahieren. Das Ergebnis ist mit der Darstellung oben identisch. Alles kein Problem, aber wir sind ja noch nicht fertig. Die Norm schreibt vor, dass Werte normalisiert abgespeichert werden, also die erste Stelle der Mantisse eine 1 sein muss. Wir normalisieren durch schieben nach links um 13 Stellen. Und was passiert am rechten Rand? Dort werden natürlich Nullen nachgeschoben, was denn sonst. Unsere $0{,}2_{10}$ sieht jetzt plötzlich so aus:



Die Genauigkeit unserer Darstellung hat durch zwei ganz einfache Rechenoperationen massiv gelitten. Aus der 0,2 wurde eine 0,199951.

Fazit: wenn es mathematisch exakt werden muss, ist diese Art der Arithmetik mit äußerster Vorsicht zu verwenden oder besser etwas anderes (das gibt es, aber das würde hier jetzt zu weit führen). Wenn es Ihnen egal ist, sprechen Sie mit ihrer Bank und sorgen Sie dafür, dass alle Rundungsfehler aufsummiert und an karitative Organisationen gespendet werden.

12.5 Ganz tief drin-der harte Kern der Informatik

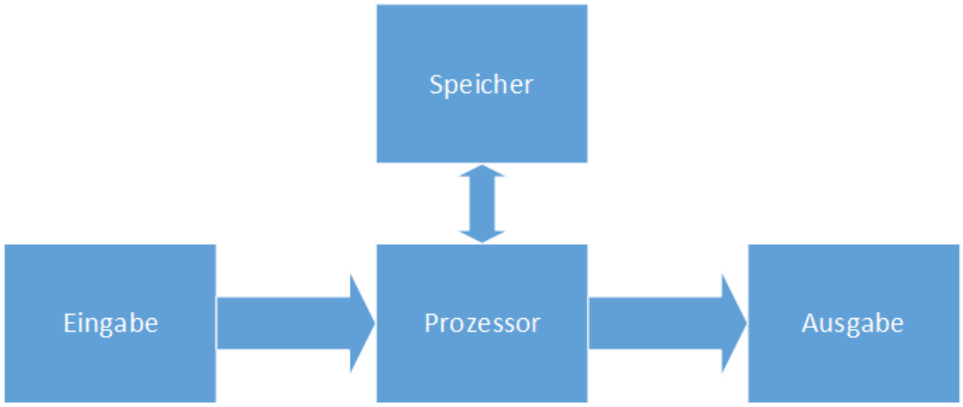
Gehen wir jetzt einen Schritt weiter und werden eine Stufe allgemeiner und größer. Wir leisten uns den Luxus, unsere Bausteine nicht mehr auf Gatterebene zusammen zu löten, sondern kaufen uns die nächst höher integrierten Bauelemente: Prozessor, Speicher und Co. Und sehen uns dann an, wie so ein Programm in den Speicher gelangt und was der Prozessor damit macht.

Damit das nämlich funktioniert, müssen wir uns noch etwas klar machen, was uns zwar aus heutiger Sicht völlig normal erscheint, aber in den Anfangsjahren der Computerei nicht selbstverständlich war. Ich meine das gemeinsame Abspeichern von Programm und Daten in ein und demselben Speicher. Das Programm, der Ablauf der Datenverarbeitung, steht also nicht etwa als Lochstreifen zur Verfügung, sondern teilt sich den Speicher mit den Daten. Und dann ist es nur noch ein kleiner Schritt, wenn ein Programm sich selbst verändert, denn woran sollte es denn den Unterschied zwischen Daten und Programm erkennen? Und -schwupps- sind wir bei der Künstlichen Intelligenz und den sich selbst modifizierenden und damit lernenden Programmen....unsere Programme werden diese Stufe allerdings nie erreichen.

12.5.1 Von-Neumann-Architektur

Allen (modernen) Rechnern ist ein grundsätzlicher Aufbau gemeinsam. Auf der einen Seite steckt man Eingaben (Daten) hinein, die werden in irgendeiner Form (nach Programm) verarbeitet und kommen auf der anderen Seite wieder heraus.

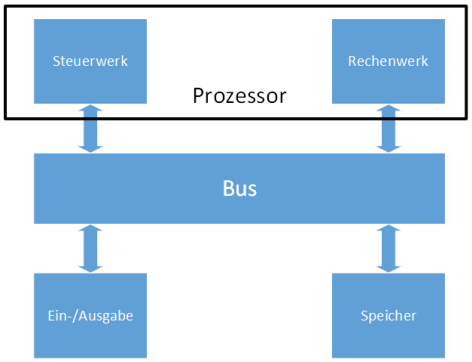
Dieses grundlegende Prinzip von Eingabe → Verarbeitung → Ausgabe, oder auch EVA-Prinzip gilt bis heute. Es ist auch schwer vorstellbar, dass wir erst die Ausgabe betrachten, dann die Verarbeitung anstoßen und am Ende die Daten eingeben. Markieren Sie sich diese Zeile bitte fett mit einem Leuchtmarker Ihrer Wahl, Sie werden in den Praktika übrigens mindestens einmal gegen genau dieses Prinzip verstoßen und Ihr Betreuer wird Sie dann nur nach den drei magischen Buchstaben fragen.



Beachten Sie die Richtung der Pfeile. Die Eingabe geht nur in den Prozessor hinein, die Ausgabe kommt nur vom Prozessor. Lediglich zwischen Speicher und Prozessor findet ein echter Austausch statt, Daten werden vom Speicher in den Prozessor geladen und wieder im Speicher abgelegt.

Die Pfeile spielen dabei eine besondere Rolle, man nennt sie *Bus*. Das ist übrigens keine kryptische Abkürzung, sondern steht wirklich für das Verkehrsmittel, nur werden hier statt Personen eben Daten von einem Ort zum anderen transportiert.

Man kann das alternativ auch so darstellen:



Das Bussystem verbindet alle Komponenten untereinander, hier sind jetzt Ein- und Ausgabe auch bidirektional angebunden, weil beide Richtungen zusammen gefasst wurden. Der Prozessor wurde nebenbei in seine beiden wichtigsten Teile zerlegt, das Steuerwerk, welches die Befehle (das Programm) analysiert und ausführt und das Rechenwerk, das die eigentlichen (arithmetischen) Operationen ausführt.

Das Steuerwerk

Das Steuerwerk ist der heimliche Kommandeur im Prozessor. Es bestimmt, welcher Befehl wann und durch was mit welchen Daten abgearbeitet wird und was mit den Ergebnissen passiert. Das geschieht in vier aufeinanderfolgenden Stufen:

1. Holen (fetch): Der nächste Befehl wird aus dem Speicher geholt. Dazu wird der Wert des Programmzählers (Programcounter), einem Register im Steuerwerk, ausgelesen, auf den Adressbus gelegt und der unter dieser Adresse abgespeicherte Wert gelesen.

2. Dekodieren (decode): Der so erhaltene Wert wird anhand einer Tabelle dekodiert. Erst jetzt ist klar, was passieren soll.
3. Ausführen (execute): Je nach Befehl wird jetzt etwas geschehen. Handelt es sich etwa um eine Addition, wird diese durch das Rechenwerk durchgeführt. Dazu sind eventuell weitere Aktionen des Steuerwerks nötig, um die benötigten Operanden in die Register des Rechenwerkes zu schaffen.
4. Schreiben (write): Das Ergebnis der Operation (z.B. die Summe) muss wieder im Speicher oder einem anderen Register abgelegt werden, sonst wäre die ganze Aktion ja umsonst gewesen.

Rechenwerk und Register

Das Rechenwerk, oder auch *Arithmetic Logic Unit=ALU*, ist die zweite große Einheit im Prozessor. Hier werden die eigentlichen Berechnungen durchgeführt. Es besteht im wesentlichen aus mehreren Registern, die Anzahl der dafür eingesetzten Bit bestimmt die Datenbreite des Rechners. Haben die Register nur Platz für 8 Bit, können auch nur Werte im Bereich 0 bis 255 in einem Schritt verarbeitet werden (trotzdem können 32 Bit-Additionen durchgeführt werden, es dauert halt nur deutlich länger). Das wichtigste Register ist dabei der *Akkumulator* (oder kurz Akku, von lat.accumulare=sammeln), hier steht immer der erste der (beiden) Operanden und er enthält am Ende einer Rechenoperation das Ergebnis. Weitere Register nehmen den zweiten Operanden auf oder speichern Zwischenergebnisse. Dazu kommen einige 1 Bit-Register, die in Abhängigkeit einer Rechenoperation gesetzt oder gelöscht werden. Diese Register werden auch *Flags* genannt, sie zeigen an, ob das Ergebnis einer Rechenoperation 0 ist, ob bei der Addition ein Überlauf oder Übertrag aufgelaufen ist, ob das Ergebnis negativ wurde usw. (die Anzahl und Bedeutung hängen sehr stark vom Hersteller ab).

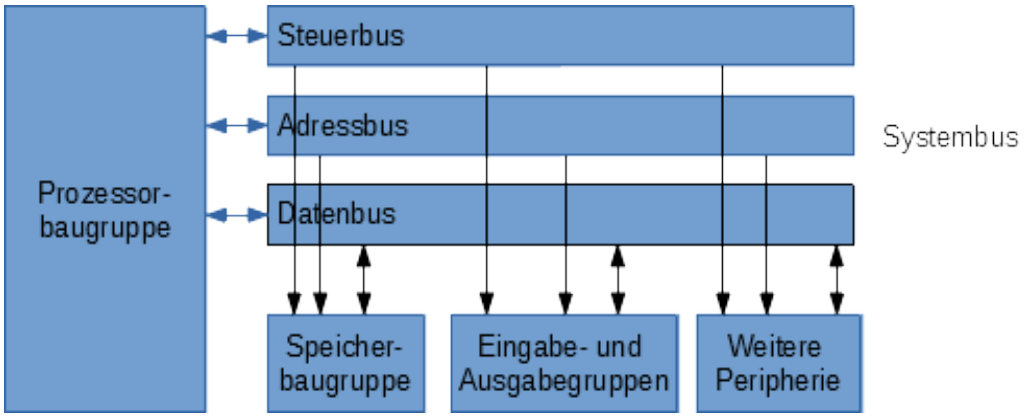
Wichtig in diesem Zusammenhang ist die Erkenntnis, dass die ALU nur mit ganzzahligen Werten umgehen kann. Um Fließkommazahlen zu verarbeiten muss es entweder eine spezielle *Floating Point Unit (FPU)* neben der ALU geben oder die Fließkommaoperationen werden mühsam in eine Folge von ganzzahligen Operationen umgeändert. Aktuelle Rechner verfügen alle über (mindestens) eine FPU.

Der Prozessor

enthält alle zentralen Einheiten des Rechners, also Steuerwerk, Rechenwerk, Register und Flags in einem Gehäuse, Dabei ist es durchaus üblich, einige dieser Einheiten auch mehrfach vorzuhalten, um in gleicher Zeit mehr Rechenoperationen abschließen zu können.

Der Bus

Der Bus besteht bei aktuellen Rechnern in Wirklichkeit aus mehreren Bussen für Daten, Adressen und Steuerdaten (Datenbus, Adressbus und Steuerbus). Dem Steuerbus kommt hier eine besondere Bedeutung zu, weil er regelt, welche Daten wann von welcher Adresse gelesen werden oder wann der Akku einen Wert im Speicher ablegen darf usw.. Der Steuerbus gibt also den Takt vor, mit dem der Rechner arbeitet. Dieses Konzept kann sehr schnell an sein Limit gelangen, wenn etwa der Prozessor Daten schneller liefert als sie der Speicher ablegen kann. Auch die Anzahl der Busteilnehmer hat entscheidenden Einfluss auf den Datendurchsatz, denn wenn zwei Teilnehmer Daten über den Bus austauschen sind alle anderen Komponenten zum Nichtstun verdammt. Um dieses Dilemma zu lösen werden immer mehr Busse eingeführt oder der Bus wird geteilt, damit auf beiden Teile unterschiedliche Aktionen stattfinden können.



Exemplarischer Programmablauf

Zum Abschluss dieses Abschnittes fügen wir unser Wissen über den internen Aufbau eines Rechners und seine Abläufe zusammen und bauen uns daraus einen fiktiven Computer. Natürlich vereinfachen wir an allen Ecken und Enden, so hat unser von-Neumann-Rechner nur 8 Bit Datenbreite, er kann nur mit ganzen Zahlen umgehen (keine FPU) und wir machen uns zunächst auch keine Gedanken darüber, wie wir ein Programm in den Rechner hinein bekommen oder wie man Ergebnisse ausgeben kann.

Die folgende Tabelle[50] zeigt die Belegung unseres Speichers dar, für unsere Zwecke reichen acht Byte aus. Die erste Spalte gibt die Adresse der Speicherstelle an, in Spalte 2 steht der Inhalt dieser Speicherstelle (üblicherweise ein Befehl oder ein Wert). In Spalte 3 wird die Funktion des Befehls umgangssprachlich erläutert und in der letzten Spalte wird exemplarisch diese Bedeutung in Mnemonics bzw. Assembler dargestellt.

Adresse	Befehl	Bedeutung	Mnemonics
0000	10010111	Speichere die Eingabe an Adresse 0111	IN AX, 1 MOV 07h, AX
0001	00100000	Beschreibe das Register AX mit dem Wert 0	MOV AX, 0
0010	00110111	Addiere zum Wert im Registers AX den Wert, der sich an Adresse 0111 befindet	ADD AX, 07h
0011	01000111	Vermindere den Wert an Adresse 0111 um 1	DEC 07h
0100	10000010	Setze das Programm mit dem Befehl an Adresse 0010 fort, falls das Ergebnis des letzten Befehls nicht 0 ist	JNZ 3h
0101	10110000	Das Programm endet. Der Inhalt des Registers AX soll ausgegeben werden	OUT 2, AX
0110	00000000	Freie Speicherstelle	
0111	00000000	Freie Speicherstelle, wird aber vom Programm benutzt	

50 nach: Informatik für Dummies, E. G. Haffner, Wiley-VCH-Verlag 2017

Bei einem von-Neumann-Rechner teilen sich Daten und Programm ja den Speicher, man kann also nicht ohne weiteres erkennen, ob eine Speicherzelle jetzt Daten oder Programmanweisungen enthält. Diese Entscheidung wird innerhalb des Programms getroffen.

Unser Programm würde also ohne weitere Unterteilung diese Folge von 0 und 1 ergeben:

10010111001000000011101110100011110000010101100000000000000000000 (ohne Gewähr für die Richtigkeit). Damit kann keiner etwas anfangen (außer dem Rechner), versuchen wir es alternativ mal in hexadezimaler Schreibweise, das sieht dann etwa so aus: 9720374782B00000. Kürzer, aber auch nicht besser. Deswegen wurden schon sehr früh die Mnemonics erfunden, die mit kurzen Begriffen beschreiben, was so ein Befehl macht. Das ist zwar schon ein wenig besser verständlich, aber noch nicht anschaulich genug. Schauen wir zurück in den vorderen Teil und schreiben dieses Programm in C++:

```
int register ax = 0;
int eingabe;
cin >> eingabe;
do
{
      ax = ax + eingabe;
      eingabe = eingabe - 1;
} while (eingabe > 0);
cout << ax;
```

Besser? Vielleicht im Ansatz, aber verstehen werden Sie das erst nach der 2 Übung. Sie sollten sich aber immer daran erinnern, dass jedes Wort, das Sie in den Rechner in einer Programmiersprache eintippen, zunächst in *Mnemonics* (oder auch Assembler) und dann in eine Folge von 0 und 1 übersetzt werden muss.

Haben Sie eigentlich auch heraus bekommen, was dieses Programm macht? Wenn ja, sind Sie sehr gut, wenn nicht sollten Sie sich einfach mal mit Papier und Bleistift hinsetzen und das Programm Schritt für Schritt durchgehen und die sich ändernden Werte notieren. Die Lösung verrate ich aber nicht (besuchen Sie die Vorlesung).

Wie die Programme in den Rechner kommen

Wir haben jetzt gesehen, welche Abstraktionsebenen unser Programm durchläuft, um von einer Hochsprache in eine Folge von Nullen und Einsen übersetzt und damit ausführbar zu werden. Entscheidend ist, dass die Programmausführung in einem Rechner immer bei Adresse 0x0000 beginnt, denn der Programmzähler (Programcounter) wird beim Einschalten des Rechners einfach auf genau diese Adresse gesetzt. Üblicherweise befindet sich an dieser Stelle des Speichers unveränderlicher Speicher (Read-Only-Memory=ROM), der seinen Inhalt einmal fest eingebrannt bekam und sich auch nachträglich nicht mehr ändern lässt. Unsere Programme können also rein technisch schon mal überhaupt nicht an Adresse 0x0000 abgelegt werden!

Im ROM befindet sich das Basic-Input-Output-System (BIOS), das den Rechner grundsätzlich zum Leben erweckt und die vorhandenen Bauteile initialisiert und startet. Die letzte Aktion des BIOS ist dann das Laden des eigentlichen Betriebssystem von der Festplatte oder SSD. Und eines der ersten Programme, die geladen werden ist der Loader (manchmal auch Relocator genannt), der in der Lage ist, selbst wieder Programme zu laden, dabei aber die darin befindlichen Adressen so umrechnet, dass das Programm an jeder Stelle im Hauptspeicher liegen kann und trotzdem funktioniert. Damit können jetzt auch mehrere Programme gleichzeitig im Speicher liegen, solange sich ihre Adressbereiche nicht überlappen. Dafür sorgt das Betriebssystem (egal wie es heißt), es sorgt nicht nur dafür, dass jedes Programm in einem

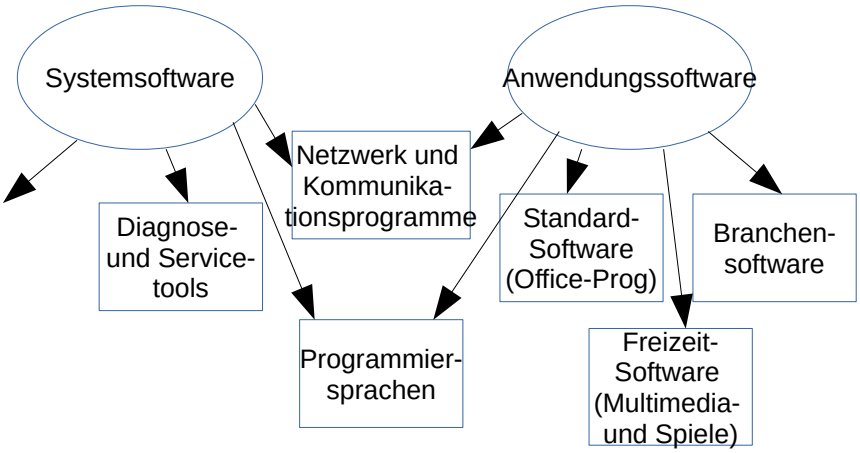
eigenen Adressraum läuft, sondern auch dafür, dass jedes dieser Programme aufgerufen wird und arbeiten kann. Jetzt ist unser Rechner betriebsbereit und wir können uns um die Software kümmern.

12.5.2 Software

Software ist all das, was man an einem Rechner nicht anfassen oder kaputt machen kann. Man unterscheidet zwei Arten von Software aufgrund ihrer Bedeutung und Funktionalität. Zum einen die Systemsoftware, die notwendig ist, um einen Rechner überhaupt zu betreiben. Das beginnt mit den Programmen im ROM, ohne die der Rechner keinen Pieps machen würde und geht über die Treiber für die vorhandenen (oder zusätzlich eingebauten) Bauteile bis zum eingesetzten Betriebssystem. Auf der anderen Seite steht unsere Anwendungssoftware, mit der wir tagtäglich produktiv mit dem Rechner arbeiten (sollten). Dazu gehören solche banalen Dinge wie eine Textverarbeitung, eine Medienplayer oder die allseits beliebte Tabellenkalkulation.

In diese Kategorie gehören aber auch die gesamten Branchenlösungen wie von SAP, Datev, Lexware oder anderen Herstellern, die im kommerziellen Umfeld weit verbreitet sind und auf abertausenden Rechnern installiert sind.

Und dann gibt es ein paar Programme, die sich nicht so recht entscheiden können, in welche Gruppe sie gehören. Das sind in erste Linie alle Programme zur Kommunikation und Steuerung des Netzwerkes. Und in zweiter Linie sind es die Programmiersprachen, die auch nicht wissen, was sie eigentlich sind.



Bleibt eines noch zu klären: was ist eigentlich ein Programm? Formal ist ein Programm die Formulierung eines *Algorithmus* in einer Programmiersprache. Ein unbekanntes Wort durch ein anderes Fremdwort erklären nennt man *Rekursion*, hilft aber nicht weiter.

Ein Algorithmus ist nichts weiter als ein Kochrezept, mal nicht für einen leckeren Kuchen sondern eine Vorschrift zur Lösung eines Problems. Das Besondere daran ist, dass dieser Algorithmus jedes Detail des Verfahrens beschreibt, so dass jedermann (und jede Maschine) unter identischen Umständen zu den gleichen Ergebnissen kommt. Man nennt dieses Verhalten deterministisch, also vorhersehbar. (Bei Kuchenrezepten ist das Ergebnis nicht immer vorhersehbar). Aber: ein Algorithmus ist nicht an eine Programmiersprache gebunden, prinzipiell ist jede Programmiersprache geeignet, einen Algorithmus umzusetzen (das sagt nichts über die Eignung einer konkreten Programmiersprache für einen konkreten Algorithmus aus).

Vom Problem zum Programm

Bevor man aber ein Programm zur Lösung eines Problems in Händen (oder im Rechner) hält, muss man aber wirklich arbeiten, diese Arbeit stellt einen wesentlichen Teil der Informatik dar und wird von

Studierenden gerne unterschätzt. Es beginnt mit der vollständigen Beschreibung des Problems, insbesondere mit der Abgrenzung, welche Fälle mit dem Problem gelöst werden sollen und welche nicht.

Daran schließt sich die Problemanalyse an. Dabei wird das Problem Schritt für Schritt immer weiter vereinfacht (Top-Down-Analyse), bis nur solche Schritte oder Anweisungen übrig bleiben, die sich leicht in eine Programmiersprache umsetzen lassen.

Nächster Schritt ist das Übertragen des Algorithmus in eine Programmiersprache und der ausgiebige Test. Dabei stellen sich üblicherweise Fehler heraus, die entweder eine Änderung im Algorithmus erfordern aber zumindest Korrekturen im Programm erzwingen. Um aber überhaupt Testen zu können, benötigt man Testfälle, die auch tatsächlich alle Varianten des Programms testen. Auch das macht Arbeit.

Mehr dazu finden Sie im Kapitel Lösungswege erarbeiten ganz weit vorne (S. 14).

13 Über den STeLlerrand geblickt

Dieses Kapitel müsste eigentlich „Weiter als weitergehend: Über den Tellerrand…“ überschrieben werden. Aber das wäre dann doch etwas verwirrend. Bis hierher habe ich versucht, so allgemein wie möglich über C++ zu berichten. Das gebe ich jetzt auf und verlange mindestens die Version C++11 oder höher.

Im Februar 2019 wurde gerade der C++-Standard 20 (also für das Jahr 2020) verabschiedet, dieses Skript beschreibt (mit wenigen Ausnahmen) den Stand von 1998. Für einen Anfänger im Programmieren macht das überhaupt nichts, da die vermittelten Konzepte und Verfahren nach wie vor aktuell sind. Nichts desto trotz hat sich aber bei der Sprache einiges getan, das man durchaus in eigenen Projekten einsetzen kann. Außerdem gibt es in C++ noch das ein oder andere Konzept, das zwar nicht besonders neu ist, aber einem das Leben vereinfacht. Davon haben wir ja bereits gelegentlich Gebrauch gemacht, unsere Ein- und Ausgabe sowie der „Datentyp“ String möge als Beispiel dienen. Von diesen objektorientierten Erweiterungen möchte ich hier jetzt noch in Ausschnitten berichten

13.1 Immer der richtige Typ: auto

Diese Überschrift ist nicht ganz korrekt, aber die Richtung stimmt. Warum soll man mühsam den richtigen Datentyp angeben, wenn doch aus dem Kontext ganz klar ist, welcher Datentyp einzig eingesetzt werden kann. Für diesen Fall gibt es den Datentyp `auto`. Er darf überall dort stehen, wo aus dem Zusammenhang eindeutig klar ist, um was es geht.

```
auto a = 3;  // int
auto pi = 3.14159; // double
auto pi2 = 3.14159f;// float
auto s = "1234567"; // char[]
auto n = 1234567u  //unsigned int
```

In der dritten Zeile sieht man auch bereits die Grenzen des Verfahrens. Für den Compiler ist der Unterschied zwischen `float` und `double` nicht erkennbar, er nimmt double an. Um dann eine `float`-Variable zu bekommen, muss man diesen Fall durch das nachgestellte „f“ manuell regeln. Analog gilt das für ganzzahlige Variablen mit oder ohne Vorzeichen, der Sonderfall muss extra geregelt werden.

So richtig interessant wird es erst, wenn der Datentyp recht komplex ist oder der Programmierer selbst nicht so genau weiß, wie es sich mit den Daten verhält. Einen Programmierer, der nicht genau weiß, welche Typen sich hinter den Daten verstecken-kaum denkbar! Doch, nämlich immer dann, wenn diese Datentypen in einer Bibliothek definiert sind, an die man nur schlecht heran kommt. Dieser Fall wird uns im folgenden beschäftigen.

In den C++Versionen vor C++11 wurde das Schlüsselwort `auto` in der Praxis verwendet, um eine Speicherklasse zu beschreiben. Das wurde aber so gut wie nie benötigt und hat daher die hier beschriebene neue Bedeutung bekommen

13.2 Das bessere Array: vector

Ein Array fasst unter einem Bezeichner eine Reihe von gleichartigen Daten (oder Objekten) zusammen. Größter Nachteil ist es, dass die (maximale) Anzahl der Elemente bereits beim Programmieren feststehen muss und es sehr häufig passiert, dass man aus Versehen über die Grenzen des Arrays hinaus zugreift und damit unschöne Laufzeitfehler und Programmabstürze herbeiführt.

Das kann man durch den Einsatz von dynamischen Speicherverfahren ein wenig mildern, fängt sich damit aber neue Probleme und Fehlerquellen ein. Die Lösung bildet die Klasse `vector`, die alle Funktionen, die man sich für ein Array wünscht, enthält: ein Vektor kann dynamisch wachsen und schrumpfen, der Zugriff wird überprüft, Bereichsüberschreitungen haben keine katastrophalen Nebenwirkungen. Und so manche Fähigkeit, die man bei einem Array selbst mühsam programmieren muss, gibt es beim Vektor als Zugabe auf Knopfdruck.

Betrachten wir das folgende Fragment:

```
vector <int> v(5);
v[0] = 17; v[1] = -45; v[2] = 2;
v.push_back(1);
v.push_back(27);
v.push_back(14);
v.pop_back();
for (auto i = 0; i < v.size(); i++)
        cout << i << "\t" << v.at(i)  << endl;
```

In Zeile 1 wird ein Vektor definiert. Da so ein Vektor ja aus allen möglichen und unmöglichen Datentypen bestehen kann, muss man dem Compiler in spitzen Klammern diesen Typ angeben[51]. Wir haben uns hier zur Vereinfachung mal an den ganzen Zahlen versucht. Als letzter, aber wichtigster Teil folgt der Name des Vektors, hier v und die gewünschte Startgröße 5 (die runden Klammern sind hier kein Fehler!). Diese Startgröße darf entfallen, wenn man mit einem leeren Vektor starten möchte.

Zeile 2 ist nichts Neues im Vergleich zu den Arrays. Hier werden die ersten drei Elementen mit Werten gefüllt. Und die restlichen beiden? Die werden brav vom Compiler auf 0 (Null) gesetzt, ein echter Fortschritt!

Die drei folgenden Zeilen eröffnen völlig neue Möglichkeiten. Mit `push_back()` wird am Ende des Vektors ein neues Element hinzugefügt, der Vektor wird also um ein Element länger. Eine Fähigkeit, die wir bei den Array schmerzlich vermisst haben. Natürlich gibt es auch die gegenteilige Funktion `pop_back()`, die das letzte Element aus dem Vektor entfernt. Und weil sich die Größe des Vektors so ständig ändern kann, benötigt man eine Funktion, die die aktuelle Größe ermitteln kann. Diese findet sich mit `size()` in der `for`-Schleife.

51 Die Ähnlichkeit zu den template-Funktionen ist kein Zufall, der Vektor (und vieles andere) ist nämlich in der STL definiert, der *Standard Template Library*

Neben diesen grundlegenden Funktionen gibt es noch weitere, die einen den Vektor noch angenehmer machen:

`reserve (n)`	Vergrößert die Kapazität um n Elemente
`resize(n)`	legt die neue Größe auf n Elemente fest (es wird ein neuer Vektor dieser Größe angelegt und der Inhalt des alten Vektors kopiert. Was nicht hinein passt, fällt einfach weg)
`bool empty()`	liefert `true` zurück, wenn der Vektor leer ist
`insert (pos, element)`	Fügt das Element an Position `pos` in den Vektor ein, die vorhandenen Elemente werden verschoben
`pos erase (pos)`	Löscht das Element an der angegebenen Position und liefert die Folgeposition zurück. Die nachfolgenden Elemente rücken auf. (`pos` ist ein Iterator!)
`pos erase (von, bis)`	Löscht die Elemente an der angegebenen Position und liefert die Folgeposition zurück. Die nachfolgenden Elemente rücken auf. (`pos`, `von` und `bis` sind Iteratoren!)

13.3 Iteratoren und Co.

Analog zur Weiterentwicklung des Arrays zum Vektor musste sich auch der Index weiter entwickeln. Aus dem Index wird der Iterator. Wenn Sie einmal zurück denken, wie oft sie eine `for`-Schleife über alle Elemente eines Arrays programmiert haben! Das geht mittels eines Iterators schneller, sauberer und kürzer.

Unser Beispielvektor aus dem letzten Abschnitt lässt sich auch so ausgeben:

```
for (auto it = v.begin(); it < v.end(); it++)
        cout << *it  << endl;
```

Zunächst fallen die Start- und Endwerte in der `for`-Schleife auf. Dort stehen jetzt keine ganzzahligen Werte, sondern die Funktionen `begin()` und `end()`. Auch hier schlägt die Tatsache zu, dass so ein Vektor (und auch alle andere Containerklassen aus der STL) mit beliebigen Datentypen umgehen können und es daher eine gute Idee ist, wenn die einzelnen Elemente dynamisch im Speicher abgelegt werden. Das erklärt auch den Zugriff auf das einzelne Element über `*it`, also einen Zeiger.

Es geht aber noch eine Stufe kompakter:

```
for (auto  elem : v)
        cout << elem << endl;
```

Mit vier Worten kann man mit dieser Range-`for`-Schleife einen kompletten Vektor (genauer genommen einen kompletten Container) durchlaufen! Das ist Rekord für Schreibfaule.

13.4 Sehr listig: list

Im Kapitel 9.5.5 haben wir uns mit verketteten Listen beschäftigt. Ein mühsames Geschäft. Dabei geht auch das viel einfacher, wenn man nur weiß, wie. Etwa so:

```
list <int> l(5);
```

```
int k = 1;
for (auto it = l.begin(); it != l.end(); it++) {
        *it = k;
        k++;
}
for (auto elem : l)
        cout << elem << endl;
```

In Zeile 1 definieren wir uns eine Liste l mit 5 Elementen vom Datentyp `int`. Das sieht der Definition des Vektors noch sehr ähnlich. Auch das Füllen der Listenelemente mit Inhalt ist kaum von der entsprechenden Operation bei einem Vektor zu unterscheiden. Wenn Sie aber mal auf die Seite 112 zurück blättern und sich den Wust an Zeigern ansehen, die erforderlich waren, um eine Liste von vorne nach hinten zu durchlaufen oder um neue Elemente in die Liste aufzunehmen ist die `for`-Schleife hier doch wirklich eine Erleichterung. Aber immerhin ein Zeiger (der Iterator) ist uns geblieben!

Auch das Hinzufügen oder Entfernen von Elementen, egal ob am Anfang, am Ende oder mittendrin reduziert sich auf den Aufruf der passenden Funktion:

```
l.pop_front();  //erstes Element entfernen
l.push_front(22); // zusätzliches Element vorne anfügen
l.push_back(88); // zusätzliches Element hinten anfügen
```

Noch nie waren Listenoperationen so einfach!

13.5 Die Blackbox der STL: algorithm

Nun enthält die STL nicht nur die oben bereits erwähnten Container-Klassen (und einige mehr), sondern auch eine sehr umfangreiche Sammlung von Funktionen, die sich auf diese Container anwenden lassen. Ein paar davon haben wir schon kennengelernt (size(), begin(), end()...), es gibt aber rund 70 davon. Eine davon möchte ich hier exemplarisch vorstellen.

Früher oder später enthält eine Übungsaufgabe den Punkt Sortieren, je nach aktuellem Kenntnisstand von einer Folge Zahlen oder von Strukturen. Sehr beliebt, weil auch überall als fertiges Programm zu finden das Bubblesort-Verfahren. Ein wenig effizienter geht das Sortieren aber auch mit dem Selection-Sort, hier als Funktion:

```
void selection_sort(vector<int> &v)
// sortiert den Vektor v mittels Selection-Sort
{
        int temp;
        int  mini;  // der aktuell minimale Wert
        int anz = v.size();
        int pos = 0;
        for (int a = anz - 1; a >= 0; a--) {//äußere Schleife
                pos = a;
                mini = v[a];
                for (auto i = 0; i <= a; i++) {//innere Schleife
                        if (mini < v[i]) {
                                pos = i;
```

```
                        mini = v[i];
                    }
                }
                temp = v[pos];
                v[pos] = v[a];
                v[a] = temp;
        }
}
```

Dazu gehört folgendes Hauptprogramm:

```
vector <int> vec { 4, 12, -7, 21, 76, -5, -9, 22, 9, 0 };
selection_sort(vec);
for (auto elem : vec)
        cout << elem << "\t";
cout << endl;
```

Ohne das Sortierverfahren im Detail zu erläutern genügt uns hier die Erkenntnis, dass im Kern zwei ineinander geschachtelte Schleifen mit gegenseitiger Abhängigkeit (der Start- und Endwerte) mit einer Bedingung und einem Tauschverfahren eine zentrale Rolle spielen.

Nutzen wir die Fähigkeiten der STL, schrumpft unser Sortierverfahren auf dieses Hauptprogramm, da wir ja eine vorgefertigte Funktion nutzen:

```
vector <int> vec2{ 4, 12, -7, 21, 76, -5, -9, 22, 9, 0 };
sort(vec2.begin(), vec2.end());
for (auto elem : vec2)
        cout << elem << "\t";
cout << endl;
```

Das Ergebnis sieht wenig überraschend so aus:

```
C:\Users\Joerg\source\repos\vektor\Debug\vektor.exe
Unsortiert:
4       12      -7      21      76      -5      -9      22      9       0
Selection Sort:
-9      -7      -5      0       4       9       12      21      22      76
STL-Sort:
-9      -7      -5      0       4       9       12      21      22      76

Drücken Sie eine beliebige Taste . . .
```

Allerdings: wir wissen nicht, wie das Sortieren in der Bibliothek tatsächlich erfolgt, wir können aber davon ausgehen, dass es ein sehr effizientes und millionenfach getestetes Verfahren ist.

Ein Nachteil dieser großartigen Funktion sei aber an dieser Stelle auch genannt: sort() funktioniert nur, wenn es direkten Zugriff (also über die eckigen Klammern[]) auf die Elemente hat und für die Elemente der Kleiner-Operator definiert ist. Handelt es sich um eigene Klassen oder Objekte muss man daher den Kleiner-Operator vorher überladen.

13.6 Gängige Funktionen der STL

13.6.1 Suchen: find()

Die Funktion `find(von, bis, wert)` sucht im Container nach dem (ersten) Auftreten eines Elementes. Der Suchbereich wird durch die beiden ersten Parameter angegeben, der dritte ist das zu suchende Element. Es wird ein Iterator auf das erste gefundenen Element zurückgegeben, gab es keine Übereinstimmung zeigt der Rückgabewert hinter das letzte, gültige Element.

13.6.2 Sortieren: sort()

`sort (von, bis)` sortiert in einem Container den angegebenen Bereich, `sort()` funktioniert nur, wenn es direkten Zugriff (also über die eckigen Klammern[]) auf die Elemente hat und für die Elemente der Kleiner-Operator definiert ist. Handelt es sich um eigene Klassen oder Objekte muss man daher den Kleiner-Operator vorher überladen.

13.6.3 Kopieren: copy()

Die Funktion `copy (von, bis, ziel)` kopiert den Bereich von...bis in einen neuen Container `ziel`. Dabei wird die Größe von ziel nicht überprüft, sondern ggfs. gnadenlos überschrieben.

13.6.4 Umkehren: reverse()

Diese Funktion `reverse(von, bis)` kehrt die Reihenfolge der Element in einem Container(bereich) um, das erste Element wird zum letzten und das letzte Element zum ersten usw.

13.6.5 Füllen: fill();

Die Funktion `fill (von, bis, wert)` füllt einen Container(bereich) mit dem Wert `wert`.

13.6.6 Vergleichen: equal()

Die Funktion `equal (von, bis, mit_von)` vergleicht, ob der Bereich `von...bis` des Containers mit dem entsprechenden Bereich des Containers `mit_von` übereinstimmt

13.7 Warum noch selber machen?

Das werden Sie sich jetzt bestimmt auch fragen. Auf diese Frage gibt es aber eine einfache Antwort: um es zu verstehen. Als Programmiereinsteiger müssen Sie zunächst lernen, wie Algorithmen und Datenstrukturen funktionieren und welche Prozesse dabei im Computer ablaufen. Denn nur so können Sie erkennen, wann es sich lohnt, auf eine fertige Bibliothek zurückzugreifen und wann man selbst Hand anlegen muss. Gestandene Programmierprofis verbringen in der Tat mehr Zeit damit, eine passende Bibliothek und/oder Funktion für ihr Problem zu finden und weniger damit, diese selbst zu programmieren. Sie dürfen ja mit dem frisch erworbenen Führerschein auch nicht gleich Bus und Lkw fahren, sondern erst nach ein paar Jahren Erfahrung und Routine.

14 Literatur

Für dieses Skript wurde jede Menge Literatur genutzt und gelesen. Hier die hoffentlich vollständige Liste:

C++ Das Übungsbuch Testfragen und Aufgaben mit Lösungen Peter Prinz – Ulla Kirch-Prinz mitp, 3. Auflage. ISBN 978-3-8266- 1765-2	**C++ Lernen und professionell anwenden** Peter Prinz – Ulla Kirch-Prinz mitp, 6. Auflage, ISBN 978-3-8266-9195-9 
Der C++ Programmierer C++ lernen-Professionell anwenden-Lösungen nutzen Ulrich Breymann Hanser 2009, ISBN 978-3-446-41644-4 	**Programming Principles and Practice** Using C++ Bjarne Stroustrup Addison-Wesley, 2. Auflage, ISBN 978-0-321-99278-9 
C++ Einführung und Leitfaden Stanley B. Lippman 2. Auflage Addison-Wesley, 2. Auflage, ISBN 978-3-893-19375-2	**C Programmieren** Ein Kurs zum Selbststudium mit Musterlösungen Guido Krüger Addison-Wesley, ISBN 978-3-893-19372-1
C++ Einführung in die objektorientierte Programmierung Stephen Prata The Waite Group, ISBN 978-3-893-62701-1	**C++ IT-Tutorial** C++ didaktisch und praxisgerecht lernen Herbert Schild mitp, 1. Auflage, ISBN 978-3-826-60980-0
C++ Programmieren mit Stil Eine systematische Einführung Thomas Strasser dpunkt.verlag, 1. Auflage, ISBN, 978-3-920-99369-0	**C++ für C-Programmierer** Jürgen Dankert B. G. Teubner, ISBN 978-3-519-02641-9
C++ Der Einstieg Arnold Willemer Wiley/wrox-press ISBN 978-527-76044-2 	**Grundkurs C++** Jürgen Wolf Rheinwerk Computing ISBN 978-3-8362-3895-3 
C++ Das komplette Starterkit für den einfachen Einstieg... Dirk Louis Hanser, ISBN 978-3-446-44597-0	**Informatik für Dummies** **Das Lehrbuch** E. G. Haffner Wiley-VCH-Verlag 2017 ISBN 978-527-71024-9

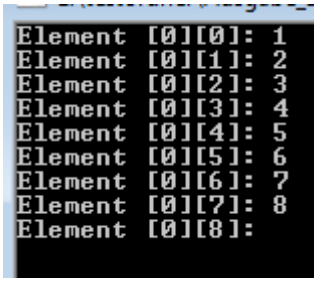
Die mit „Tipp“ versehenen Bücher sind besonders gut für Einsteiger geeignet und werden seit vielen Jahren von den Studierenden gerne und erfolgreich genutzt. Die „Profi“-Bücher sind in erster Linie für (angehende) Informatiker gedacht oder für Leute, die bereits eine andere Programmiersprache beherrschen und C++ zusätzlich lernen wollen/müssen.

15 Programmierhäppchen

Hier finden Sie in loser Folge ein paar Programmfragmente, die häufig benötigt werden, aber nicht in den normalen Fluss des Skriptes passen.

15.1.1 Ein-/Ausgabe eines Arrays (ein- und zweidimensional)

```
int feld[10][10];
	for (int z = 0; z < 10; z++)  // Zeilenweise
	{
		for (int s = 0; s < 10; s++)  //über alle Spalten
		{
			cout << "Element ["  // Text vor der Eingabe
				<< z << "][" << s << "]: ";
			cin >> feld[z][s];  // einlesen
//Alternativ:		cout << feld[z][s]; // ausgeben
		}
	}
```



Muss nur eine Dimension eingelesen werden, kann man auf alles, was mit der Variablen z zu tun hat entfernen:

```
	int feld[10];
	for (int s = 0; s < 10; s++)  //über alle Spalten
		{
			cout << "Element ["  // Text vor der Eingabe
				<<< s << "]: ";
			cin >> feld[s];  // einlesen
		}
```

15.1.2 Einlesen einer Zeile mit Leerzeichen

```
	string eingabe_zeile;
	// "Das ist eine Eingabe mit Leerzeichen!";
	getline(cin, eingabe_zeile);
	cout << "Die Eingabe: " << eingabe_zeile;
```

Statt `cin` darf natürlich hier auch eine Datei-Variable als Parameter an die Funktion getline() stehen, das funktioniert ganz genauso. Natürlich muss die Datei vorher geöffnet worden sein.

15.1.3 Zufallszahlen erzeugen

Der übliche Zufallsgenerator in C++ erzeugt Zufallszahlen zwischen Null und 2,14 Millionen nach einem bestimmten Verfahren. Die Folge der Zufallszahlen ist ohne weiteres Zutun immer dieselbe, d.h. bei

jedem Programmstart wird auch immer die gleiche Folge von Zufallszahlen erzeugt. Für Testzwecke ist das ganz prima, um Spiele zu programmieren etwas unfair.

Man muss also mittels eines zusätzlichen Wertes dafür sorgen, dass die Zufallsfolge bei jedem Programmstart an einer anderen Stelle der Zufallsfolge startet.

Die folgende Funktion liefert Zufallszahlen im Bereich 1 bis n:

```
int zufallszahl(int n)
{
	return rand() % n + 1;
	//Zufallszahl im Bereich 1 bis n
}
```

Mit dem passenden Hauptprogramm

```
	srand(time(0)); //Startwert
	for (int i = 1; i <= 20; i++)
		cout << zufallszahl(100) << ",";
```

erhält man z.B. das hier:

```
40,56,77,58,66,86,29,10,71,43,28,68,51,19,1,71,64,56,97,21,
```

Die Zeile `srand(time(0));` darf im Programm nur einmal vorkommen, auf keinen Fall mit in die Funktion zur Zufallszahlen-Erzeugung packen. Denn `time(0)` liefert die Sekunden seit dem 1.1.1970, und damit bei den üblichen Aufgaben im Praktikum, die nur Sekundenbruchteile Rechenzeit benötigen, immer dieselbe Zufallszahl.

15.1.4 Aufzählungswerte ein- und ausgeben

Aufzählungstypen werden ja ohne besondere Behandlung als ganze Zahlen (`int`) ausgegeben, weil sie sich problemlos darin umwandeln lassen (siehe Seite 52). Wenn man aber schon mühsam einen solchen Datentyp geschaffen hat, möchte man ihn in der Regel auch ein- oder ausgeben.

Die Eingabe kann auf diese Weise erfolgen:

```
	enum Farben {rot, orange,gelb, gruen, blau, violett};
	Farben regenbogen;
	int i;

	cout << "1=rot, 2=orange, 3=gelb, 4=gruen, 5=blau, 6=violett:";
	cin >> i;
	switch (i)
	{
	case 1:regenbogen = rot; break;
	case 2:regenbogen = orange; break;
	case 3:regenbogen = gelb; break;
	case 4:regenbogen = gruen; break;
```

```
        case 5:regenbogen = blau; break;
        case 6:regenbogen = violett; break;
        }
```

Die Ausgabe sieht dann ganz ähnlich aus:

```
cout << "Der Regenbogen hat die Farbe ";
        switch (regenbogen){
        case rot: cout << "rot"; break;
        case orange: cout << "orange"; break;
        case gelb: cout << "gelb"; break;
        case violett: cout << "violett"; break;
        case blau: cout << "blue"; break;
        case gruen: cout << "gruen"; break;
        }
```

Hier sieht man, dass es zwischen den einzelnen Elementen des Aufzählungstyps und der Ausgabe keinen Zusammenhang gibt, in den Anführungszeichen darf stehen, was will. Sieht dann für den Anwender etwas merkwürdig aus, hat aber sonst keinen Effekt. Also besser keine Tippfehler einbauen.

15.1.5 Aktuelles Datum und Uhrzeit

Um die interne Uhr des Rechners auszulesen, kann man folgendes Programmstück verwenden:

```
#define _CRT_SECURE_NO_WARNINGS 1
#include <iostream>
#include <string>
#include <ctime>
using namespace std;

int main()
{
        time_t sek;   // Sekunden
        time(&sek); // ermittelt die aktuelle Zeit/Datum
        string zeit = ctime(&sek);  // Zeit wird in einen String konvertiert
        cout << zeit;  // Fri Feb 12 10:22:21 2016\n\0
}
```

Die `#define` Anweisung gleich zu Beginn ist bei manchen Entwicklungsumgebungen erforderlich (z.B. bei Visual Studio 2013)

Als Ergebnis erhält man einen String mit 26 Zeichen mit dem Aufbau

`Wochentag Monat Tag Std:Min:Sek Jahr\n\0` (siehe Listing)

15.1.6 Tastatur direkt auslesen

Mit diesem Codefragment lassen sich Tastatureingaben ermitteln, die keine Return-Taste erfordern.

```
        int ch;
```

```
	do
	{
		cout << "Bitte eine Taste druecken:....";
		while (!_kbhit());   // auf beliebigen Tastendruck warten
		ch = _getch();              // Tastaturcode einlesen
		cout << "Tastencode: " << ch << endl;
	} while (ch != 27);         //ESC-Taste beendet
```

Einsetzbar für die Eingabe von Menüpunkten oder zur Steuerung von Spielen. Funktions- und Sondertasten liefern zwei Codes hintereinander: das erste Zeichen ist 0 oder 224, danach kommt der eigentliche Tastaturcode

15.1.7 Zeit stoppen

Wenn die zu messende Zeit relativ lang ist, kann man sich so behelfen:

```
#include <ctime>
#include <iostream>
using namespace std;
int main() {
	time_t start, ende;
	start = time(0);
	// hier kommen die zu messenden Programmteile
	ende = time(0);
	cout << "hat gedauert: " << ende - start <<"Sek"<< endl;
}
```

Die Auflösung von 1 Sekunde kann aber nur einen groben Richtwert geben.

Genauer wird es, wenn man direkt die Millisekunden misst, die auch tatsächlich von der CPU für den aktuellen Prozess verbraucht wurden. Es ändert sich nur der Datentyp der Variablen zur Zeitmessung und die verwendete Funktion zur Ermittlung der aktuellen (CPU-) Zeit.

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

int main() {
	clock_t start, ende;
	start = clock();
	// hier kommen die zu messenden Programmteile
	ende = clock();
	cout << "hat gedauert: " << ende - start <<" ms"<< endl;
}
```

16 Glossar

Algorithmus: eine Ablaufbeschreibung oder -vorschrift, die exakt und präzise den Lösungsweg beschreibt

ASCII: American Standard Code for Information Interchange, Zuordnungsvorschrift zwischen den einzelnen Zeichen eines Zeichensatzes zu einer (ganzen) Zahl. Der Bereich unterhalb 32 enthält (nicht druckbare) Steuerzeichen für Drucker und Bildschirme, der Bereich 32 bis 126 die international einheitlichen Buchstaben, Ziffern und Sonderzeichen, oberhalb von 127 werden landes- und sprachspezifische Zeichen untergebracht (nicht genormt)

Anweisung (Statement): In C++ ein elementares Programmkommando, wird mit einem Semikolon abgeschlossen

Bibliothek (Library): Sammlung von Funktionen (oder Klassen) zu einem bestimmten Thema, können vom Compilerhersteller geliefert werden, zugekauft oder selbst erstellt werden. Der Quelltext der Bibliotheken muss zur Benutzung nicht bekannt sein.

Datei (File): Speicherung von Informationen oder Daten auf einem Datenträger (intern oder extern)

Definition: Mit einer Definition wird eine Variable deklariert und gleichzeitig der notwendige Speicher angelegt. Jede Variable darf nur genau einmal definiert werden. Bei Funktionen versteht man darunter die Abfolge von Anweisungen, die beim Funktionsaufruf durchlaufen werden.

Deklaration: Eine Deklaration dient dem Compiler dazu, Informationen über Variablen und Funktionen zu bekommen. Es wird kein Speicher reserviert und keine Anweisung definiert. Deklarationen dürfen beliebig oft wiederholt werden.

Editor: Werkzeug zur Eingabe von Quelltext. Im Gegensatz zu einer Textverarbeitung werden keinerlei zusätzliche Informationen (Formatierungen, Schriftarten etc.) mit abgespeichert.

Header-Datei: Die Header-Datei sammelt Deklarationen von Variablen und Funktionen, um sie an einem Ort zu bündeln.

Heap: Speicherbereich des Hauptspeichers, der im wesentlichen für globale Variablen genutzt wird

Index: In einem Array der Zeiger auf das Feldelement

Kommentar (comment): zusätzliche Erläuterungen im Quelltext, die der Compiler nicht beachtet

Konstante (const): eine Variable, deren Wert im Laufe der Programmausführung nicht geändert werden kann und darf

L-Value: Ein Ausdruck oder eine Variable, die in der Lage ist, einen Wert per Zuweisung aufzunehmen. Üblicherweise stehen L-Values auf der linken Seite einer Zuweisung. Nicht alle Datentypen sind als L-Value geeignet.

Modulo: Operator, der den Rest einer ganzzahligen Division ergibt (z.B. 22 / 4 = 5, 22 % 4 = 2);

Operand: Ausdruck, der mittels Operator mit anderen Operanden verknüpft werden kann, in der Regel ein R-Value

Operator: Verknüpfungsvorschrift für Operanden

Parameter: Argumente an eine Funktion oder einen Konstruktor

Quelltext (Listing): Der im Editor erfasste Programmtext in C++-Syntax

R-Value: Ergebnis eines Ausdrucks, steht in der Regel auf der rechten Seite einer Zuweisung

Referenzparameter (call by reference): Übergabe einer Adresse als Parameter

Rekursion: sich selbst aufrufende Funktion (eventuell auch mit einem Umweg über eine andere Funktion)

Schnittstelle (interface): Funktionskopf mit den Parametern, liefert alle notwendigen Informationen zur Verwendung einer Funktion, gehört in die Header-Datei.

Stack (Stapel): Speicherbereich im Hauptspeicher, der in erster Linie für lokale Variablen und Parameter beim Aufruf von Funktionen verwendet wird.

STL (Standard Template Library) eine Sammlung von universellen Datentypen (Containerklklassen) und zugehörigen Funktionen für (fast) alle Fälle des Programmmiereralltags

Syntax: Die Regeln einer Programmiersprache

Top-Down-Entwurf: Herangehensweise an größere Aufgaben durch schrittweise Verfeinerung, bis schließlich nur noch elementare Ablaufstrukturen übrig bleiben.

Überladen: Funktionen (aber auch Operatoren) mit gleichen Namen, aber unterschiedlichen Parametern (in Anzahl und/oder Typ)

Variablen: Mit einem Namen versehene Speicherbereiche zur Aufnahme von veränderlichen Daten

Vektor: eindimensionales Feld

Wertparameter (call by value): Übergabe von Parametern an eine Funktion durch das Kopieren der Werte auf den Stack. Ist der Normalfall, wenn keine besonderen Maßnahmen getroffen werden.

Zeichenkette: eine Folge von Elementen des Datentyps `char`, abgeschlossen durch eine Null.

Zeiger (pointer): Variable, die keinen Wert eines bestimmten Datentyps enthält, sondern die Adresse eines Speicherbereichs, in dem ein Datum abgelegt werden kann.

17 ASCII-Tabelle

dez	oktal	hex	Zeichen	dez	oktal	hex	Zeichen	dez	oktal	hex	Zeichen
32	040	20	leer	64	100	40	@	96	140	60	`
33	041	21	!	65	101	41	A	97	141	61	a
34	042	22	"	66	102	42	B	98	142	62	b
35	043	23	#	67	103	43	C	99	143	63	c
36	044	24	$	68	104	44	D	100	144	64	d
37	045	25	%	69	105	45	E	101	145	65	e
38	046	26	&	70	106	46	F	102	146	66	f
39	047	27	'	71	107	47	G	103	147	67	g
40	050	28	(	72	110	48	H	104	150	68	h
41	051	29	)	73	111	49	I	105	151	69	i
42	052	2A	*	74	112	4A	J	106	152	6A	j
43	053	2B	+	75	113	4B	K	107	153	6B	k
44	054	2C	´	76	114	4C	L	108	154	6C	l
45	055	2D	-	77	115	4D	M	109	155	6D	m
46	056	2E	.	78	116	4E	N	110	156	6E	n
47	057	2F	/	79	117	4F	O	111	157	6F	o
48	060	30	0	80	120	50	P	112	160	70	p
49	061	31	1	81	121	51	Q	113	161	71	q
50	062	32	2	82	122	51	R	114	162	72	r
51	063	33	3	83	123	53	S	115	163	73	s
52	064	34	4	84	124	54	T	116	164	74	t
53	065	35	5	85	125	55	U	117	165	75	u
54	066	36	6	86	126	56	V	118	166	76	v
55	067	37	7	87	127	57	W	119	167	77	w
56	070	38	8	88	130	58	X	120	170	78	x
57	071	39	9	89	131	59	Y	121	171	79	y
58	072	3A	:	90	132	5A	Z	122	172	7A	z
59	073	3B	;	91	133	5B	[	123	173	7B	{
60	074	3C	<	92	134	5C	\	124	174	7C	|
61	075	3D	=	93	135	5D	]	125	175	7D	}
62	076	3E	>	94	136	5E	^	126	176	7E	~
63	077	3F	?	95	137	5F		127	177	7F	(DEL)

Aus Platzgründen wurde auf die nicht druckbaren Zeichen (kleiner 32) verzichtet. Die wichtigsten finden sich in der Tabelle im Abschnitt 6.1.3 auf Seite 45.

Die Zeichen oberhalb von 127 sind nicht einheitlich, es sollte dem geneigten Leser aber durch das bisher gewonnene Wissen mühelos gelingen, mittels einer `for`-Schleife diese Tabelle fortzusetzen.