# Computer Aided Image Processing I

## Einführung – Computersysteme und Informatik

### *Was ist ein Computer und wie funktioniert er ?*

***Computer*** sind technische Geräte, die umfangreiche Informationen mit hoher Zuverlässigkeit (meistens!) und großer Geschwindigkeit (von Ausnahmen abgesehen) automatisch verarbeiten und aufbewahren können.

Computer haben nichts mit Automaten zu tun, die nur vorher festgelegte Aktionen ausführen können (Kaffeeautomat etc.). Man kann Computern die Anweisungen, nach denen Sie arbeiten sollen, immer wieder neu vorgeben. Diese Arbeitsanweisungen bezeichnet man als ***Algorithmen*** (Singular: ***Algorithmus***):

Damit ein Computer Algorithmen ausführen kann, müssen diese in einem eindeutigen und präzisen Formalismus beschrieben werden. In diesem Fall werden die Algorithmen Programme genannt. Ein Computer kann ohne Programme nicht arbeiten, deshalb spricht man von ***Computersystemen***, sofern das technische Gerät Computer und die Programme zur Steuerung des Computers gemeint sind.

Alle materiellen Teile eines Computersystems bezeichnet man als ***Hardware***, alle immateriellen Teile (z.B. Programme) als ***Software***.

Alles was ein Computersystem kann, kann ein Mensch prinzipiell auch. Computersysteme haben gegenüber dem Menschen jedoch drei wesentliche Vorteile:

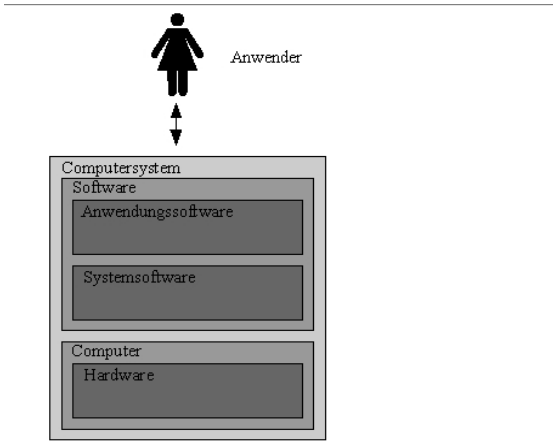
- ***Hohe Speicherfähigkeit***
  In einem Computersystem können mehrere Billionen Informationseinheiten gespeichert und schnell wiedergefunden werden
- ***Hohe Geschwindigkeit***
  Ein Computersystem addiert heute in einer Sekunde ca. 100 Millionen Zahlen. Nimmt man an, ein Mensch würde in einer Sekunde zwei solcher Zahlen addieren, dann würde er in einem Jahr ca. 32 Millionen Zahlen addieren können und hätte nach 3 Jahren genau so viele Zahlen addiert wie ein gängiges Computersystem in einer Sekunde.
- ***Hohe Zuverlässigkeit***
  Der Computer wird im Gegensatz zum Menschen nicht müde. (Manche Software erweckt allerdings den Eindruck Computer seien prinzipiell unzuverlässig!)

Der Begriff ***Software*** ist noch umfassender als der Begriff Programm. ***Software*** sind Programme, zugehörige Daten und **Dokumentation**. Es wird zwischen Anwendungssoftware und Systemsoftware unterschieden.

***Systemsoftware*** ist Software, die für eine spezielle Hardware entwickelt wurde, um den Betrieb genau dieser Hardware zu ermöglichen. Beispiele: Betriebssysteme (Linux etc.), Compiler, Datenbanken etc.

***Anwendungssoftware*** ist Software, die Aufgaben des Anwenders mit Hilfe eines Computersystems löst. Beispiele: Textverarbeitung, Tabellenkalkulation, Spiele. Anwendungssoftware benutzt normalerweise die Systemsoftware.

Anwendungssoftware, Systemsoftware und Hardware bilden gemeinsam ein ***Computersystem***.




## *Die Zentraleinheit*

Ein Computer besteht aus

- einer Zentraleinheit (*Central Unit*) und
- einer Ein- / Ausgabesteuerung.

In der Zentraleinheit werden die Programme abgearbeitet. Sie besteht aus:

- dem Prozessor (*CPU – **C**entral **P**rocessing **U**nit*) und
- dem Arbeitsspeicher

Die Zentraleinheit kommuniziert über die Ein- / Ausgabesteuerung mit den Peripheriegeräten (Ein- / Ausgabegeräte, externe Speicher, Netzwerkanschluss, WLAN ...).

Der Arbeitsspeicher, auch ***RAM*** (***r**andom **a**ccess **m**emory* = Direktzugriffsspeicher) genannt, dient dazu Informationen zur Programmlaufzeit aufzubewahren und zur Verfügung zu stellen. Die Informationen im RAM gehen nach Abschalten der Stromzufuhr verloren. Der Hauptvorteil der Arbeitsspeichers gegenüber externen Speichermedien besteht in der deutlich schnelleren Zugriffszeit auf die gespeicherten Informationen.

Die kleinste Einheit eines Arbeitsspeichers bezeichnet man als ***Speicherzelle***. Eine Speicherzelle kann im allgemeinen eine Zahl oder mehrere Zeichen aufbewahren. Jede Speicherzelle besitzt eine Adresse (Namen). Durch die Angabe dieser Adresse kann der Inhalt der Speicherzelle angesprochen werden.

Die Informationsmenge, die in einem ***Speicher*** aufbewahrt werden kann, wird als dessen Speicherkapazität bezeichnet. Die Maßeinheit für ***Speicherkapazität*** heißt ***Byte***. In einem Byte kann normalerweise ein Textzeichen gespeichert werden. Ein Byte wird zu folgenden Einheiten zusammengefasst:

| | | | |
|---|---|---|---|
| 1 kB = | 1 Kilobyte = | 1 024 Bytes | |
| 1 MB = | 1 Megabyte = | 1 024 kB = | 1 048 576 Bytes ≈ 1 Million Bytes |
| 1 GB = | 1 Gigabyte = | 1 024 MB ≈ | 1 Milliarde Bytes |
| 1 TB = | 1 Terabyte = | 1 024 GB ≈ | 1 Billion Bytes |
| 1 PB = | 1 Petabyte = | 1 024 TB ≈ | 1 Billiarde Bytes |

Ein Byte wiederum besteht aus 8 Binärzeichen, abgekürzt ***Bit*** (*binary digit*) genannt. Ein Bit kann einen zweielementigen Zeichenvorrat darstellen, z.B. wahr – falsch, eins – null.

Der Arbeitsspeicher eine Computers hat derzeit in der Regel eine Kapazität von einigen GB.

Externe Speicher haben folgende Kapazitäten:

- CD-Rom: 650 MB
- DVD: 4,7 / 8,5 GB
- Blu-ray: 25 GB bis 500 GB (im Labor)
- USB-Stick: einige GB
- Festplatte: 100 GB bis einige 1 TB

Die zweite Komponente der Zentraleinheit ist der Prozessor. Der ***Prozessor*** eines Computers arbeitet die Programme ab. Programme sind stark formalisierte Algorithmen. Der Formalismus den ein Prozessor versteht nennt man Programmiersprache, genauer Maschinensprache.

Ein ***Programmiersprache*** ist eine formalisierte Sprache

- deren Sätze und Worte aus einem festgelegten Zeichenvorrat gebildet werden,
- deren Sätze nach genau festgelegten Regeln gebildet werden (*Syntax*) und
- die die Bedeutung jeden Satzes festlegt (*Semantik*).

Ein ***Programm*** ist als nichts anderes als ein ***Algorithmus***, formuliert in einer ***Programmiersprache***.

Der Prozessor verfügt zum Verarbeiten von Programmen über folgende Fähigkeiten:

- Informationen aus dem Arbeitsspeicher in den Prozessor transportieren
- Informationen vergleichen, addieren, subtrahieren, multiplizieren, dividieren, logisch verknüpfen
- Informationen aus dem Prozessor in den Arbeitsspeicher transportieren.

### *Ein- und Ausgabegeräte*

***Bildschirm (evtl. Touchscreen)***, ***Tastatur*** und ***Maus*** sind die wichtigsten Geräte über die der Benutzer mit dem Computersystem kommuniziert. Weitere wichtige Geräte sind ***Scanner***, ***digitale Videokameras*** und ***Drucker***. ***Mikrophon*** und ***Lautsprecher*** ermöglichen den Einsatz gesprochener Sprache (oder von Musik bzw. Geräuschen) in Computersystemen.

Zunehmend an Bedeutung gewinnen ***Multimedia***-Anwendungen, die Text, Daten, Grafiken, Bilder, Filme, Klänge und gesprochene Sprache kombinieren.

### *Externer Speicher*

Informationen im Arbeitsspeicher gehen verloren, sobald der Computer ausgeschaltet wird. Für die langfristige Aufbewahrung von Informationen werden ***externe Speicher*** verwendet, die eine höhere Speicherkapazität besitzen und wesentlich preiswerter als Arbeitsspeicher sind.

Es gibt verschiedene externe Speichermedien. Sie unterscheiden sich in der Zugriffsgeschwindigkeit, in der Speicherkapazität und im Preis. Sie lassen sich in drei große Klassen einteilen:

- Festplattenspeicher (hohe Kapazität, sehr schnell)
- USB-Massenspeicher (mittlere Kapazität, schnell)
- CD-Rom- / DVD- / Blu-ray -Speicher (mittlere Kapazität, mittlere Zugriffsgeschwindigkeit)

### *Vernetzung*

Computersysteme können heute in der Regel mit anderen Computersystemen Informationen austauschen, sie sind vernetzt. Die Verbindung erfolgt physikalisch durch Kabel oder Funk. Die Kabel und Funkstrecken, die die Verbindungen herstellen werden in ihrer Gesamtheit als ***Netz*** bezeichnet. Zusätzlich benötigt man noch ***Netz-Software***, die dafür sorgt, dass die elektronische Informationsübermittlung stattfindet.

Zwei wichtige Netze sind

- das ***Intranet*** und
- das ***Internet***.

Ein ***Intranet*** verbindet Computersysteme eines Unternehmens oder einer Organisation.

Das ***Internet*** verknüpft weltweit Intranets oder einzelne Computersysteme miteinander.

In Abhängigkeit von der Übertragungsgeschwindigkeit werden folgende Übertragungsmedien benutzt:

- verdrillte Zweidrahtleitungen (*twisted pair*) oder Koaxialkabel: 10 Mbit/s bis 1000 Mbit/s
- Lichtwellenleiter bzw. Glasfaserkabel: 100 Mbit/s bis 10.000 Tbit/s
- Wi-Fi (WLAN): 2 Mbit/s bis 3,5 Gbit/s (in Vorbereitung)

Die Übertragung von Informationen in Netzen erfolgt paketweise, d.h. die zu übertragenden Informationen werden in ein oder mehrere Pakete fester oder variabler Länge aufgeteilt und dann getrennt über das Netz transportiert.

Durch ***Übertragungsprotokolle*** wird geregelt auf welche Weise ein Computersystem Zugang zum Übertragungsmedium erhält. In Abhängigkeit vom Übertragungsmedium und dem verwendeten Übertragungsprotokoll sind unterschiedliche Netzanschlusssysteme nötig:

- Ein ***Netzwerk-Adapter*** für den Anschluss an ein lokales Netzwerk (LAN, local area network). Lokale Netzwerke verbinden Computersysteme über kurze Entfernungen (einige Meter bis wenige Kilometer).
- Ein ***WLAN-Adapter*** für den Zugang zu einem Funknetzwerk (WLAN).

Über ein Netzwerk können auch Ressourcen (Drucker, Scanner, externe Speicher) gemeinsam von verschieden Computersystemen genutzt werden.

Ein ***Server*** ist ein Gerät, dass festgelegte Dienste auf Anforderung für andere Computersysteme erbringt (z.B. Druck-Server, Archiv-Server etc.).

Die Computersysteme, die Dienste von einem Server in Anspruch nehmen heißen ***Clients***.

### *Das Betriebssystem*

Zur Steuerung und Verwaltung der einzelnen Komponenten eines Computersystems dient das Betriebssystem. Das Betriebssystem erledigt unter anderem folgende Aufgaben:

- Voreinstellung der Hardware nach dem Einschalten des Computers auf definierte Anfangswerte
- Eröffnung des Dialogs zwischen Benutzer und Computer über Bildschirm, Tastatur und Maus
- Interpretation der Kommandos des Benutzers an das Betriebssystem
- Zuweisung von Ressourcen (Arbeitsspeicher, Rechenleistung etc.) an die einzelnen Programme
- Behandlung der Ein- / Ausgabeanforderungen
- Verwaltung der Peripherie
- Verwaltung der Dateien

Das Betriebssystem selbst ist ein Programm wie jedes andere auch. Da es ständig benötigt wird befindet sich ein Teil des Betriebssystems ständig im Arbeitsspeicher. Nicht benötigt Teile des Betriebssystems (und auch von anderen Programmen) werden auf die Systemfestplatte ausgelagert („Swapping″) und bei Bedarf in den Arbeitsspeicher geladen.

Es gibt unterschiedlich leistungsfähige und unterschiedliche stark verbreitete Betriebssysteme:

- Windows 10 (Microsoft)
- Linux (Open Source)
- OS X (Apple)
- MVS bzw. OS/390 (IBM, für IBM Großrechner)
- Android / iOS (Google et al. / Apple, mobile Geräte)

# Internet, WWW und HTML

## *Definition des Begriffs Internet*

Das ***Internet*** besteht aus

- einer Vielzahl von Computern,
- die direkt oder indirekt miteinander verbunden sind
- die das selbe Übertragungsprotokoll (TCP/IP) verwenden,
- auf denen Dienste angeboten und / oder genutzt werden,
- einer Vielzahl von Benutzern, die von ihrem Computersystem aus Zugriff auf diese Dienste haben
- einer Vielzahl weiterer Computernetzwerke, die über Kommunikationscomputer (***Gateways***) erreichbar sind.

Das Internet stellt eine Kommunikationsinfrastruktur zum gleichberechtigten Informationsaustausch zur Verfügung, analog zum Telefonnetz, das die Infrastruktur für die Sprachkommunikation bereitstellt.

## *Der Aufbau des Internet*

Das Internet wurde durch keine zentrale Organisation aufgebaut. Vielmehr schlossen sich staatliche und wissenschaftliche Organisationen zusammen und mieteten Leitungen zwischen den einzelnen Mitglieder mit gemeinsamen Nutzungsrecht *aller* Teilstrecken für *alle* Mitglieder.

Beispiel: Gibt es zwischen A und B sowie zwischen B und C eine Verbindung, dann kann auch A mit C über den gemeinsamen Partner B kommunizieren.

Die Kommunikation zwischen Computersystemen im Internet geschieht durch das Übertragungsprotokoll ***TCP/IP*** (transmission control protocol / internet protocol). Die zu übermittelnden Datenströme werden dabei in Pakete einheitlicher Größe aufgeteilt, die voneinander unabhängig auf verschiedenen Wegen zu unterschiedlichen Zeiten zum Ziel kommen können. Jedes Paket wird mit Kopfdaten (*header*) versehen, die Angaben über den Absender und Zielcomputer enthalten.

Die Auswahl der Teilstrecken wird von speziellen Wegplanungscomputern (*Routern*) vorgenommen. Sie analysieren die in einem eintreffenden Datenpaket gespeicherte Zieladresse und ermitteln aufgrund ihrer internen Adresstabellen (*routing tables*) den weiteren Weg eines Datenpaketes durch das Netzwerk. Beim Ausfall einer Übertragungsstrecke werden sofort alternative Wege geschaltet, so dass Netzzusammenbrüche weitgehend vermieden werden können. Im Zielcomputer werden die einzelnen Datenpakete wieder zu einem vollständigen Datenstrom in der richtigen Reihenfolge zusammengesetzt. Verloren gegangene Pakete werden automatisch wieder angefordert.

Die Aufteilung der Datenströme in Pakete ermöglicht die gleichzeitige Nutzung einer Leitung gemeinsam durch viele Computersysteme. Eine durchgehende Leitung von Sender zu Empfänger ist nicht notwendig, die Nutzung der Ressourcen ist wesentlich effizienter.

Nicht alle Computernetze verwenden TCP/IP. Zur Kommunikation solcher Netze mit dem Internet werden spezielle Kommunikationscomputer (***Gateways***) benötigt.

Durch die weite Verbreitung des Internet wird TCP/IP zunehmend auch für firmeninterne Netzwerke verwendet.

Ein ***Intranet*** verbindet Computersysteme eines Unternehmens oder einer Organisation basierend auf der Technik des Internets miteinander, ist aber selbst *kein* Bestandteil des Internet.

Ein ***Extranet*** erweitert ein Intranet für den Zugriff von außerhalb der Organisation, etwas zu Kunden oder Lieferanten. Es gibt einer definierten Benutzergruppe einen gezielten Zugriff auf ausgewählte Informationen.

## Die Adressierung im Internet

Jedes Computersystem im Internet besitzt eine eindeutige » Rufnummer «, seine ***IP-Adresse***. Sie besteht aus vier Bytes, die dezimal durch Punkte getrennt geschrieben werden (Beispiel: 192.111.98.1). Die weltweit eindeutige Kennzeichnung der Computersysteme im Internet regeln die NICs (*Network Information Centers*). Der deutsche Teil des Internet wird von DENIC in Frankfurt am Main verwaltet.

Es gibt drei Klassen von Netzadressen, die die Größe (Anzahl Computersysteme) des jeweiligen Netzwerkes festlegen:

- *Class-A* Adresse: Aufbau A.x.x.x mit $0 \le A \le 126$.
  Auf der Welt gibt es genau 126 *Class-A* Adressen (10.x.x.x wird nicht vergeben).
- Class-B Adresse: Aufbau B1.B2.x.x mit $128 \le B1 \le 191$, $0 \le B2 \le 255$.
- Mit dieser Adresse kann man ca. 16 000 Computersysteme verwalten. Über 60% aller *Class-B* Adressen sind bereits verteilt.
- Class-C Adresse: Aufbau C1.C2.C3.x mit $192 \le C1 \le 255$, $0 \le C2,C3 \le 223$.
  Sie erlaubt bis zu 255 Computer in einem Adressblock. Es sind erst ca. 4% aller *Class-C A*dressen vergeben.

Dieser Aufbau der Adressen erleichtert die Netzverwaltung. Die **h-da** hat *Class-B* Adresse 141.100.x.x. Statt nun alle Computer der **h-da** in internationale Tabellen einzutragen, genügt der Vermerk, wie der zentrale Zugangscomputer (*router*) der **h-da** zu erreichen ist. Alle **h-da-**Computer befinden sich aus Netzsicht hinter diesem Zugangspunkt.

IP-Adressen sind schlecht zu merken. Deshalb erhalten Computersysteme im Internet zusätzlich einen oder mehrere frei wählbare Namen, wie z.B. www.h_da.de.

Diese Namen werden ***Domain-Namen*** genannt, manchmal auch ***DNS*** (***domain name system***). Die letzte Silbe im im DNS bezeichnet man als *top level domain* (z.B. *de*, *org*, *com*). Die Domain-Namen werden wie die IP-Adressen ebenfalls von den NICs vergeben.

Werden DNS-Adressen zur Adressierung eines Zielcomputers verwendet, dann werden die Daten zuerst an einen ***Nameserver*** gesendet. Ein Nameserver ist ein Computersystem, in dem DNS-Adressen und die zugehörigen IP-Adressen in Tabellen abgelegt sind und das mit diesen Tabellen DNS-Adressen in IP-Adressen übersetzt.

## Grundlagen der Programmierung

### *Programm, Programmieren, Programmiersprachen*

Algorithmen, die von einem automatischen Prozessor abgearbeitet werden, nennt man Programme. Ein Programm stellt die Realisierung eines Algorithmus dar. Es wird ein bestimmter Formalismus (Programmiersprache) gewählt, in dem man mittels genau definierter Sprachelemente den Algorithmus und die verwendeten Daten beschreibt.

Eine umgangssprachliche Problemlösung kann von einem Prozessor nicht abgearbeitet werden, da die Umgangssprache zu viele Ungenauigkeiten und Mehrdeutigkeiten enthält. Zudem ist ein Prozessor dadurch gekennzeichnet, dass er nur über einen sehr begrenzten und einfachen Sprachvorrat verfügt, der je nach Prozessortyp und Hersteller unterschiedlich ausfallen kann.

Es gibt nun zwei Möglichkeiten sich dem Prozessor verständlich zu machen.

Entweder man wählt zur Beschreibung des Algorithmus einen Formalismus, der nur die Sprachelemente enthält, die der Prozessor versteht. Früher war dies sie einzige Möglichkeit, um Programme auf einem Prozessor ausführen zu lassen. Programmiersprachen dieser Art nennt man Maschinensprachen oder Assemblersprachen.

Die zweite Möglichkeit besteht darin, einen Beschreibungsformalismus zu wählen, der es ermöglicht Probleme zu formulieren, ohne Rücksicht auf einen bestimmten Prozessor zu nehmen. In zweiten Arbeitsschritt, muss dann der Formalismus von der allgemeinen Beschreibungsform in den Formalismus des jeweiligen Prozessors vorgenommen werden. Dieser Übersetzungsvorgang geht nach festen Regeln vor sich und kann deshalb selbst als Algorithmus formuliert werden. Einen solchen Algorithmus bezeichnet man als ***Übersetzer***, das Übersetzungsprogramm ist der ***Compiler***, der Beschreibungsformalismus ist die ***Programmiersprache***. Man spricht von einer ***problemorientierten Programmiersprache***, wenn die Sprachelemente *problemnah* und nicht maschinennah, d.h. auf den Sprachvorrat des Prozessors zugeschnitten sind.

Es existieren heute einige hundert problemorientierte Programmiersprachen für die verschiedensten Anwendungsgebiete. Die von den Programmiersprachen unterstützten Konzepte sind unterschiedlich. Man unterscheidet:

- Prozedurale Programmiersprachen
- Objektorientierte Programmiersprachen
- funktionale Programmiersprachen
- prädikative, logik-orientierte Programmiersprachen
- Deklarative Programmiersprachen

### *Editor, Compiler, Linker und Programmierumgebungen*

Möchten Sie ein Programm erstellen, dann müssen Sie das Programm, genauer den Programmtext (auch: ***Quellcode, engl. Source-Code***), mit einem Texteditor eingeben und in einer Datei abspeichern. Anschließend wird der Compiler gestartet und übersetzt die Datei und erzeugt als Ausgabe eine ***Objektdatei***.

Besteht ein Programm aus mehreren Quellcodedateien, so erzeugt der Compiler auch mehrere Objektdateien. Diese werden von einem weiteren Programm, dem ***Linker,*** zu dem endgültigen Programm zusammengefasst („gebunden").

Bestimmte Funktionen, die von (fast) allen Programmen benötigt werden, z.B. Ein- / Ausgabe, Gestaltung von Benutzeroberflächen etc., stehen in Form von ***Bibliotheken*** (engl. ***libraries***) zur Verfügung. Diese sind auch nur Objektdateien und werden ebenfalls vom Linker dem auszuführenden Programm hinzugefügt.

Komfortabler als die Verwendung von Einzelkomponenten sind ***integrierte Programmierumgebungen (IDEs)***, die die Einzelkomponenten (Editor, Compiler, Linker, Bibliotheken) und weitere Hilfsprogramme in integrierter Form enthalten (eclipse, Embarcadero C++ Builder, Microsoft Visual Studio).

### *Kontrollfragen*

- Welche Parallelen gibt es zwischen der Umgangssprache und Programmiersprachen?
  - Beide enthalten einen umfangreichen Sprachvorrat
  - Beide beschreiben einen Sachverhalt.
  - Beschreibung einer Methode
  - Eindeutigkeit
  - Beide können eine Problemlösung beschreiben
- Welche der folgenden Aussagen zu Übersetzern / Compilern sind richtig:
  - Ein Compiler wandelt im allgemeinen Quellprogramme einer problemorientierten Programmiersprache in Quellprogramme einer anderen Programmiersprache.
  - Compiler sind einfache Programme.
  - Für jede Programmiersprache ist ein Compiler ausreichend
  - Ein Übersetzer hat die Aufgabe alle Sätze einer Quellsprache in gleichbedeutende Sätze einer Zielsprache zu transformieren.
- Welche der folgenden Klassifizierungen dienen als Konzepte für Programmiersprachen:
  - prozedural
  - kaufmännisch
  - funktional
  - objektorientiert
  - mathematisch-technisch
- Erläutern Sie wie C++ Programme übersetzt und ausgeführt werden.

# Einführung in C++

# 0 Das erste C++ Programm

Wir beginnen mit einem kleinen C++ Programm

```
// Ein kleines C++ Programm
//
#include <iostream>

int main()
{
      std::cout << "Hello, world!" << std::endl;
      return 0;
}
```

Dieses Programm erzeugt als Ausgabe den Text

```
Hello, world!
```

in einem neuen Bildschirmfenster.

## *0.1 Kommentare*

Die erste Programmzeile ist

```
// Ein kleines C++ Programm
```

Die // Zeichen beginnen einen ***Kommentar***, der bis zum Ende der Zeile reicht. Kommentare werden vom Compiler ignoriert, ihr Zweck ist das Programm verständlicher zu machen.

## *0.2 #include*

Viele fundamentale Funktionalitäten, wie z.B. Ein- / Ausgabe, sind in C++ Bestandteil der ***Standardbibliothek (standard library)*** und nicht im C++ Sprachumfang enthalten. Diese Unterscheidung ist wichtig, weil der C++-Standard grundsätzlich vorhanden ist, während man die Elemente der Standardbibliothek die ein Programm verwenden möchte, explizit angeben muss.

Möchte ein Programm auf Elemente der Standardbibliothek zugreifen, so ist eine **`#include`** *Anweisung* notwendig. #include-Anweisungen stehen normalerweise am Anfang eines Programms. Unser Beispielprogramm benutzt nur den Ein- / Ausgabeteil der Standardbibliothek, der durch die Anweisung

```
#include <iostream>
```

eingebunden wird.

Durch `iostream` wird Unterstützung für sequentielle Ein- / Ausgabe angefordert. Weil der Name `iostream` in einer #include-Anweisung erscheint und in '<' und '>' - Zeichen eingeschlossen ist, bezieht er sich auf den Teil der C++-Bibliothek, der ***standard header*** genannt wird.

## *0.3 Die Funktion* **`main`**

Ein Teil des Programms, der einen Name hat und von anderen Programmteilen aufgerufen werden kann, heißt ***Funktion***. Jedes C++-Programm **muss** die Funktion `main` enthalten. Wenn ein Programm vom Betriebssystem gestartet wird, geschieht dies durch den Aufruf dieser Funktion.

Jede Funktion liefert ein Ergebnis (**Rückgabewert**, **return value**), die Funktion `main` einen integer (ganzzahligen) Wert, der anzeigt ob die Funktion erfolgreich ausgeführt wurde. Null bedeutet in diesem Fall erfolgreich, jeder andere Wert zeigt einen Fehler an.

Der Sourcecode zur Definition der Funktion `main` beginnt also mit

```
int main()
```

um eine Funktion mit Namen `main` zu definieren, deren Rückgabewert vom Typ `int` ist. `int` ist das von C++ verwendete Schlüsselwort, das den Datentyp integer beschreibt. Innerhalb der runden Klammern **(** und **)** können Parameter an die Funktion übergeben werden. In unserem Beispiel ist dies nicht der Fall, deshalb sind die Klammern leer.

### *0.4 Geschweifte Klammern*

Die Definition der Funktion `main` wird mit ***Anweisungen*** fortgesetzt, die von ***geschweiften Klammern (curly braces,*** oft einfach ***Klammern*** genannt***)*** umgeben sind:

```
int main()
{           // linke Klammer
            // hier können Anweisungen stehen
}           // rechte Klammer
```

Die Klammer bedeuten, dass C++ alles innerhalb der Klammern als ***Block (Einheit)*** behandelt. In unserem Beispiel definieren die Klammern, dass alle Anweisungen innerhalb der Klammern zur Funktion `main` gehören.

Anweisungen innerhalb der Klammern werden in der Reihenfolge in der sie erscheinen abgearbeitet.

### *0.5 Textausgabe mit der Standardbibliothek*

Die erste Anweisung innerhalb der Klammern verrichtet die ganze Arbeit unseres Programms:

```
std::cout << "Hello, world!" << std::endl;
```

Die Anweisung verwendet den ***Ausgabe Operator (output operator), <<,*** der Standardbibliothek um den Text `Hello, world!` auf die Standardausgabe zu schreiben und anschließend den Wert von `std::endl` ebenfalls auszugeben.

Wird einem Name `std::` vorangestellt, so bedeutet dies, dass der Name (hier `cout`) Teil des ***Namensraumes*** mit dem Namen `std` ist. Ein Namensraum ist eine Zusammenfassung von Namen, die in einer Beziehung zueinander stehen. Im Namensraum `std` sind alle Name enthalten, die die Standardbibliothek definiert. Zum Beispiel definiert der *header* `iostream` die Namen `cout` und `endl`, auf die im Programm mit `std::cout` und `std::endl` Bezug genommen wird.

Der Name `std::cout` bezieht sich auf den ***Standard Ausgabestrom (standard output stream)***, der die Einrichtung darstellt, die von der C++-Entwicklungsumgebung für die normale Ausgabe von Programmen vorgesehen ist. In unserem Beispiel ist dies das Fenster, in dem der Text angezeigt wird.

Die Ausgabe des Wertes von `std::endl` beendet die aktuelle Ausgabezeile, so dass jede weitere Ausgabe in einer neuen Zeile erscheinen würde (Probieren Sie das aus!).

### *0.6 Die* **`return`** *Anweisung*

Eine `return` Anweisung, wie z.B.

```
return 0;
```

beendet die Funktion in der sie erscheint und gibt den Wert, der zwischen `return` und dem Semikolon (`0` in unserem Beispiel) erscheint and das Programm zurück, das die Funktion aufgerufen hatte. Der Datentyp des Rückgabewertes muss mit dem Typ übereinstimmen, der bei der Funktionsdefinition angegeben wurde. In unserem Beispiel ist der Typ `int`.

Da es selbstverständlich viele sinnvolle Stellen gibt, um eine Funktion zu beenden, kann eine Funktion mehrere `return` Anweisungen beinhalten.

### *0.7 Eine etwas genauere Betrachtung*

In diesem Programm werden bereits zwei wichtige Konzepte von C++ verwendet: *Ausdrücke* und (*Geltungs-*) *Bereiche*. Beide Konzepte werden im Verlauf der Vorlesung noch ausführlich behandelt, aber es ist sinnvoll die Grundlagen hier zu behandeln.

Ein ***Ausdruck*** fordert die C++-Implementierung auf, eine Berechnung durchzuführen. Die Berechnung ergibt ein ***Ergebnis*** und kann zusätzlich ***Seiteneffekte*** haben. Seiteneffekte sind alle Dinge, die den Status des Programms ändern und nicht Bestandteil des Ergebnisses sind.

Ein Ausdruck besteht aus ***Operatoren*** und ***Operanden***, die jeweils auf ganz unterschiedlichen Art und Weise in Erscheinung treten können. Im Ausdruck

```
std::cout << "Hello World!" << std::endl;
```

sind die zwei `<<` Symbole Operatoren und `"Hello World!"` und `std::endl` sind die Operanden.

Jeder Operand hat einen ***Typ***. Eine genauere Erklärung folgt noch, grundsätzlich bezeichnet ein Typ eine Datenstruktur und die auf dieser Datenstruktur gültigen Operationen. Die Auswirkungen eines Operators hängen von den Typen seiner Operanden ab.

Typen besitzen Namen. Beispielsweise ist `int` als der Typ definiert, der Ganzzahlen repräsentiert.

Der << Operator besitzt zwei Operanden, in unserem Beispiel verwenden wir zwei << Operatoren und *drei* Operanden. Wie ist das möglich? Der << Operator ist ***links-assoziativ***. Dies bedeutet, falls << mehrfach in einem Ausdruck erscheint, dass jeder << den größtmöglichen Teil des Ausdrucks für seinen linken Operanden verwendet und so wenig als möglich für den rechten Operanden. Wenn wir Klammern verwenden um diesen Sachverhalt zu verdeutlichen erhalten wir folgenden Ausdruck:

```
(std::cout << "Hello world!") << std::endl;
```

Jeder `<<` Operator verhält sich entsprechend den Typen seiner Operanden. Das erste << hat `std::cout` als linken Operanden (Typ `std::ostream`). Der rechte Operand ist eine Zeichenkette. Mit diesen beiden Operanden schreibt << die Zeichen des rechten Operanden in den Ausgabestrom den der linke Operand beschreibt. Das Ergebnis des Ausdrucks ist der linke Operand!

Der linke Operand des zweiten << ist also ein Ausdruck, der `std::cout` ergibt und vom Typ `std::ostream` ist, der rechte Operand ist `std::endl`, ein ***Manipulator***. Die wesentliche Eigenschaft eines Manipulators ist, dass er den Datenstrom in den er geschrieben wird verändert, indem er irgendetwas anderes tut, als Zeichen auszugeben. Ist der linke Operand von << vom Typ `std::ostream` und ist der rechte Operand ein Manipulator, bewirkt `<<` im Datenstrom genau das, was der Manipulator fordert und liefert den Datenstrom als Ergebnis. In unserem Beispiel beendet `std::endl` die aktuelle Ausgabezeile.

Der gesamte Ausdruck hat also `std::cout` als Ergebnis, schreibt `Hello world!` in den Standardausgabestrom und beendet die Ausgabezeile. Das Semikolon beendet die Anweisung und führt dazu, dass das Ergebnis verworfen wird, wir benötigen es ja auch tatsächlich nicht mehr.

Der ***Geltungsbereich (scope)*** eines Namens ist der Bereich innerhalb des Programms in dem der Name eine Bedeutung hat. C++ kennt verschiedene Arten von Geltungsbereichen, zwei davon haben wir in unserem Programm kennen gelernt.

Der erste Geltungsbereich den wir benutzt hatten war der Namensbereich (*namespace*), der eine Zusammenfassung von Namen darstellt, die in irgendeiner Beziehung zueinander stehen. Die Standardbibliothek definiert alle ihre Namen im Namensbereich `std`, um Konflikte mit Namen zu vermeiden, die wir vielleicht selbst in unseren Programmen definieren. Um einen Namen aus der Standardbibliothek zu verwenden, muss angegeben werden, dass der Name aus der Standardbibliothek verwendet werden soll. `std::cout` bedeutet, dass der Name `cout` aus dem Namensbereich `std` gemeint ist.

Der Name `std::cout` ist ein *qualifizierter* Name, der den `::` Operator verwendet. Dieser Operator wird auch als ***Geltungsbereichsoperator (scope operator)*** bezeichnet. Links vom Operator steht der Name eines Geltungsbereichs, rechts vom Operator steht der Name der innerhalb des links bezeichneten Geltungsbereichs definiert ist.

Geschweifte Klammern sind die zweite Möglichkeit einen Geltungsbereich zu definieren. Der Rumpf (*body*) der Funktion `main` – wie alle Rümpfe von allen Funktionen – ist selbst eine Geltungsbereich. Diese Tatsache wird an Bedeutung gewinnen, wenn unsere Programme größer werden.

## *0.8 Details*

Obwohl Sie bisher nur ein sehr einfaches Programm geschrieben haben, konnten Sie schon eine Vielzahl wichtiger Dinge über die Programmiersprache C++ lernen. Da diese Dinge die Grundlage für alle weiteren Sprachelemente bilden, ist es außerordentlich wichtig, dass Sie dieses Kapitel **vollständig** verstanden haben, bevor wir fortfahren.

Um Sie dabei zu unterstützen gibt es am Ende dieses und auch aller weiteren Kapitel einen Absatz mit dem Titel *Details*, der die wichtigsten Inhalte zusammenfasst und anschließend einige Übungsaufgaben.

### *Programmstruktur*

C++ Programm können ***formal*** frei geschrieben werden, d.h. Das Leerzeichen (engl. *white space*) nur notwendig sind um Symbole und Schlüsselworte voneinander zu trennen und darüber hinaus beliebig verwendet werden können. Genauso verhält es sich mit Leerzeilen und Zeilenumbrüchen (engl. *newlines*). Sie sind nur eine andere Art von Leerzeichen und können beliebig verwendet werden. Leerzeichen werden hauptsächlich verwendet um die Lesbarkeit eines Programms zu erhöhen.

Drei Dinge sind an eine Form gebunden:

- Zeichenketten Zeichen werden in Doppelhochkommata eingeschlossen, müssen in einer Zeile stehen
- `#include` *name* #include-Anweisungen müssen auf einer separaten Zeile stehen
- `//` Kommentare Kommentare enden am Zeilenende

## *Typen*

Typen definieren Datentypen und Operationen auf diesen Datentypen. Es gibt zwei Arten von Typen in C++: Typen, die Sprachbestandteil sind, wie z.B. `int` und Typen die nicht Sprachbestandteil sind und zum Beispiel in der Standardbibliothek enthalten sind, wie z.B. `std:ostream`.

## *Namensräume (namespaces)*

Namensräume sind ein Mechanismus um zusammengehörende Name zu gruppieren, Beispiel: `std`.

## *Zeichenketten (strings)*

Zeichenketten beginnen und enden mit Doppelhochkommata ("). Jede Zeichenkette muss auf einer Zeile stehen. Einige Zeichen innerhalb von Zeichenketten haben eine besondere Bedeutung, wenn Sie nach einem backslash (\) stehen:

- \n Zeilentrenner (*newline*)
- \t Tabulator
- \b Backspace
- \" Behandelt Doppelhochkomma als Teil der Zeichenkette
- \' Fügt ein Hochkomma (') in die Zeichenkette ein
- \\ Fügt einen Backslash in die Zeichenkette ein

### *Definitionen und Header (-dateien)*

Jeder Name, der in einem C++ Programm verwendet werden soll, muss vorab definiert werden. Die Standardbibliothek verwendet dafür Header, die mit #include-Anweisungen in den Sourcecode eingebunden werden. Der Header `<iostream>` definierte die Elemente für Ein- und Ausgabe.

### *Die Funktion* `main`

Jedes C++ - Programm muss genau eine Funktion mit dem Namen `main` enthalten, die einen Rückgabewert vom Typ `int` hat. Die Ausführung eines C++ - Programms beginnt mit dem Aufruf der Funktion `main`. Der Rückgabewert 0 bedeutet Erfolg, jeder andere Wert signalisiert eine Programmausnahme.

### *Geschweifte Klammern und das Semikolon*

Diese beiden Symbole sind außerordentlich wichtig in C++ - Programmen. Leider sind sie leicht zu übersehen und werden oft vergessen, was schwer verständliche Compiler-Fehlermeldungen zur Folge haben kann.

Eine Folge von null oder mehr Anweisungen innerhalb geschweifter Klammern wird ***Block*** genannt. Die Anweisungen innerhalb eines Blockes werden in der Reihenfolge ihres Erscheinens ausgeführt. Die Anweisungen einer Funktion müssen innerhalb geschweifter Klammern stehen, auch wenn eine Funktion nur aus einer einzigen Anweisung besteht. Die Anweisungen innerhalb geschweifter Klammern definieren einen (Geltungs-)bereich.

Ein Ausdruck (`x = 0`) wird zu einer Anweisung, wenn ein Semikolon folgt (`x = 0;`).

### *Ausgabe*

Die Auswertung des Ausdrucks `std::cout << e` schreibt den Wert von `e` auf die Standardausgabe und verwendet dafür `std::cout` (vom `Typ ostream`) um verkettete Ausgabeoperation zu ermöglichen.

### *Aufgaben*

0.1 Was macht die folgende Anweisung: `3 + 4;`

0.2 Ist der folgende Programmtext ein lauffähiges Programm? Erklären Sie Ihre Antwort!

```
#include <iostream>
int main() {{{{{{ std::cout << "Hello, world!" << std::endl; }}}}}}
```

0.3 Wie sieht es mit dem folgenden Programmtext aus?

```
#include <iostream>
int main()
{
    /* Hier ist ein mehrzeiliger Kommentar, weil
       die Zeichenfolgen /* und */ verwendet werden */
    std::cout << "Does this work?" << std::endl;
    return 0;
}
```

0.4 ... und wie hiermit:

```
#include <iostream>
int main()
{
    // Hier ist ein mehrzeiliger Kommentar, der
    // die Zeichenfolgen // am Zeilenanfang verwendet und nicht /*
    // oder */ um Kommentare zu definieren
```

```
        std::cout << "Does this work?" << std::endl;
        return 0;
    }
```

0.5 Wie lautet das kürzeste lauffähige C++ – Programm?

# 1 Die Verwendung von Zeichenketten (strings)

In diesem Kapitel wird der Überblick über grundlegende C++ - Konzepte fortgesetzt, in dem wir einfache Programme schreiben, die Zeichenketten verwenden. Dabei werden Sie einiges lernen über Deklarationen, Definitionen, Variable und ihre Initialisierung, Dateneingabe und die C++ - `string` Bibliothek.

## *1.1 Dateneingabe*

Sie können bisher nur Text ausgeben, der nächste Schritt besteht darin Text einzulesen. Wir werden das `Hello, world!` Programm so modifizieren, dass es „Hallo" zu einer bestimmten Person sagt:

```
// Frage nach dem Namen und begrüße die Person
#include <iostream>
#include <string>

int main()
{
      // nach dem Namen fragen
      std::cout << "Bitte geben Sie Ihren Vornamen ein: ";

      // den Namen einlesen
      std::string name;       // Definition von name
      std::cin >> name;       // Eingabe nach name einlesen

      // Begrüßung ausgeben
      std::cout << "Hallo, " << name << "!" << std::endl;
      return 0;
}
```

Wenn das Programm gestartet wird, schreibt es

```
Bitte geben Sie Ihren Vornamen ein:
```

auf die Standardausgabe. Wenn Sie antworten, z.B.

```
Zaphod
```

wird das Programm

```
Hallo, Zaphod!
```

ausgeben.

Schauen wie uns die Erklärung an. Wenn Eingaben gelesen werden sollen, müssen sie an irgendeiner Stelle im Programm abgelegt werden. Diese Stelle nennt man ***Variable***. Eine Variable ist ein ***Objekt***, das einen Namen hat. Ein Objekt wiederum hat einen Typ und wird im Hauptspeicher des Computers abgelegt. Die Unterscheidung zwischen Variablen und Objekten ist wichtig, weil es Objekte ohne Namen geben kann, wie Sie noch sehen werden.

Möchten Sie eine Variable verwenden, so müssen Sie ihren Namen und ihren Typ angeben. In unserem Beispiel heißt die Variable `name` und ihr Typ ist `std::string`. Sie erinnern sich, dass die Verwendung von `std::` bedeutet, dass der Typ `string` Bestandteil der Standardbibliothek ist. Wie alle anderen Typen der Standardbibliothek auch, benötigt `string` einen Header, den wir mit der Anweisung `#include <string>` in unser Programm eingefügt haben.

Die erste Anweisung `std::cout ....` kennen Sie bereits. Die zweite Anweisung

```
std::string name;
```

ist eine Definition, die eine Variable mit dem Namen `name` und dem Typ `std::string` definiert. Die Definition der Variablen steht im Funktionsrumpf von `main`, deshalb ist die Variable eine ***lokale Variable***, d.h. Sie existiert genau so lange, wie der Programmteil innerhalb der geschweiften Klammern ausgeführt

wird. Sobald der Computer `}` erreicht, wird die Variable `name` zerstört und der von ihr belegte Speicherbereich an das Betriebssystem zurückgegeben. Die begrenzte Lebensdauer von Variablen ist ein wichtiger Grund, zwischen Variablen und Objekten zu unterscheiden.

Implizit über den Typ wird die ***Schnittstelle*** (***interface***) festgelegt – die Menge aller Operationen, die für diesen Typ erlaubt sind.

Eine dieser Operationen ist die ***Initialisierung***. Bei der Definition einer `string` Variable wird diese implizit initialisiert, weil dies in der Standardbibliothek so festgelegt ist. Sie werden in kürze lernen, wie Sie einer Variablen einen Anfangswert geben können. Falls Sie das nicht tun, enthält ihre `string` Variable keine Zeichen. Ein solche Zeichenkette heißt ***Leerstring*** (***empty*** oder ***null string***).

Nach der Definition von `name` folgt die Anweisung

```
std::cin >> name;        // Eingabe nach name einlesen
```

die von `std::cin` liest und das Gelesene in die Variable `name` schreibt.

Die Standardbibliothek verwendet den **>> *Operator*** für die Eingabe analog zum **<< *Operator*** für die Ausgabe. Das Einlesen einer Zeichenkette beginnt damit, dass alle ***Zwischenraumzeichen*** (***whitespaces***, Leerzeichen, Tabulator, backspace, Zeilenende) aus der Eingabe entfernt werden, dann werden Zeichen eingelesen bis wieder ein Zwischenraumzeichen oder das Dateiende gefunden werden. Die Aufgabe von `std::cin >> name` besteht also darin ein Wort von der Standardeingabe zu lesen und in `name` abzuspeichern.

Die Ein- und Ausgabeanforderungen werden von der Standardbibliothek in einer internen Datenstruktur zwischengespeichert, die ***Puffer*** (***buffer***) genannt wird. Der Puffer wird zur Optimierung von Ein- / Ausgabeoperationen verwendet. Ein- / Ausgabeoperationen sind auf den meisten Systemen relativ zeitaufwendig und unabhängig von der Anzahl der Zeichen. Deshalb werden die Zeichen von der Standardbibliothek in einem Puffer gesammelt und dann gemeinsam ausgegeben. Der Puffer wird geleert (***flushed***) in dem die Zeichen tatsächlich auf das Ausgabegerät geschrieben werden, sofern es notwendig ist. Auf diese Art und Weise werden mehrere Ausgabeanforderungen in einem einzigen Schreibvorgang zusammengefasst.

Es gibt drei Ereignisse, die dazu führen, das der Puffer geleert wird:

- der Ausgabepuffer ist voll, er wird dann automatisch geleert
- es soll von der Standardeingabe gelesen werden, der Ausgabepuffer wird in diesem Fall sofort geleert, ***anschließend*** wird von der Standardeingabe gelesen
- der Ausgabepuffer wird explizit geleert (durch einen Programmbefehl)

## *1.2 Ein Rahmen für den Namen*

Unser Programm soll so erweitert werden, dass folgende Ausgabe erscheint:

```
****************************
*                          *
* Hallo, Zaphod Beeblebrox! *
*                          *
****************************
```

Es ist eine sinnvolle Strategie, das Programm und die Ausgabe Schritt für Schritt zu entwickeln. Zuerst werden wir den Namen einlesen, dann wird jede einzelne Ausgabezeile konstruiert. Es folgt ein Beispielprogramm:

```
// den Namen einer Person erfragen und einen eingerahmten Gruß ausgeben
#include <iostream>
#include <string>


int main()
```

```
{
	std::cout << "Bitte geben Sie Ihren Vornamen ein: ";
	std::string name;
	std::cin >> name;

	// Begrüßungstext zusammenbauen
	const std::string gruss = "Hallo, " + name + "!";

	// zweite und vierte Ausgabezeile
	const std::string spaces(gruss.size(), ' ');
	const std::string zwei = "* " + spaces + " *";

	// erste und fünfte Ausgabezeile
	const std::string eins(zwei.size(), '*');

	// Alle Zeilen ausgeben
	std::cout << std::endl;
	std::cout << eins << std::endl;
	std::cout << zwei << std::endl;
	std::cout << "* " << gruss << " *" << std::endl;
	std::cout << zwei << std::endl;
	std::cout << eins << std::endl;

	return 0;
}
```

Zu Beginn frägt das Programm nach dem Namen des Anwenders und liest diesen Namen in die Variable mit dem Namen `name`. Dann wird eine weitere Variable `gruss` definiert, die die Nachricht enthält, die ausgegeben werden soll. Die nächste Variable, `spaces`, enthält genau so viele Leerzeichen, wie die Anzahl von Zeichen in `gruss`. Die Variable `zwei` enthält die zweite Ausgabezeile, die Variable `eins` erzeugt die erste Ausgabezeile, genau so viele `'*'`, wie die Anzahl der Zeichen in `zwei` beträgt. schließlich wird Zeile für Zeile die Ausgabe geschrieben.

Die #include – Anweisungen sind bekannt, aber die Definition von `gruss` verwendet drei neue Ideen.

Die erste Idee ist, dass Variablen bereits bei der Definition ein Wert zugewiesen werden kann. Dies geschieht durch das Zeichen =, gefolgt von dem Wert der zugewiesen werden soll.

Die zweite neue Idee besteht darin, das Zeichen + zu verwenden, um ein `string` und eine Zeichenkette zu verketten. Bisher wurde das Zeichen + verwendet, um eine Addition auszuführen, hier ist die Bedeutung eine völlig andere. In beiden Fällen wird das Verhalten des + Operators von seinen Operanden bestimmt. Besitzt ein Operator verschieden Bedeutungen für Operanden verschiedenen Typs, so nennt man den Operator ***überladen (overloaded)***.

Die dritte Idee ist die Verwendung von `const` in einer Variablendefinition. Sie bedeutet, dass der Wert der Variablen im weiteren Programmverlauf nicht mehr verändert werden soll. Diese Anweisung hat keine direkte Auswirkung auf das Programm, erhöht aber die Verständlichkeit.

Bitte beachten Sie, dass eine `const` – Variable bei der Definition initialisiert werden muss, eine spätere Möglichkeit gibt es ja nicht mehr. Beachten Sie weiterhin, dass der Wert mit dem die `const` – Variable initialisiert wird, selbst keine Konstante sein muss, wie im Beispiel zu sehen ist (`name` ist eine Variable, deren Wert erst zur Laufzeit des Programms durch die Benutzereingabe festgelegt wird).

Eine Operatoreigenschaft, die niemals geändert wird ist die Assoziativität. Sie wissen bereits, dass << links-assoziativ ist, so dass `std::cout << s << t` das gleiche bedeutet, wie `(std::cout << s) << t`. Der + Operator ist ebenfalls links-assoziativ. Dies bedeutet, dass man den Wert von `"Hallo, " + name + "!"` erhält indem man `"Hallo, "` mit `name` verkettet und anschließend das Ergebnis dieser Verkettung mit `"!"` verkettet.

Jetzt haben wir also den Text festgelegt, der ausgegeben werden soll und ihn in der Variable `gruss` gespeichert. Im nächsten Schritt wir nun der Rahmen erstellt, der den Text umgeben soll. Dabei lernen Sie weitere drei neue Ideen kennen:

```
std::string spaces(gruss.size(), ' ');
```

Bei der Definition von `gruss` verwendeten wir das = Symbol zur Initialisierung. Hier folgen auf `spaces` zwei Anweisungen, die durch einem Komma getrennt sind und von zwei Klammern eingeschlossen werden. Die Verwendung der Klammern bedeute, dass die Variable `spaces` mittels zweier Ausdrücke konstruiert (***construct***) wird, wobei das Vorgehen bei der Konstruktion vom Typ der Variablen abhängt! Damit Sie diese Definition verstehen können, müssen Sie also verstehen was es bedeutet ein `string` aus zwei Ausdrücken zu konstruieren.

Der erste Ausdruck, `gruss.size()`, ist ein Beispiel für den Aufruf einer ***Elementfunktion*** (***member function***). Das Objekt mit dem Namen `gruss` hat eine Komponente mit dem Namen `size`, die eine Funktion ist und somit aufgerufen werden kann, um einen Wert zu erhalten (Return Wert). Die Variable `gruss` ist vom Typ `std::string`, der so definiert ist, dass die Auswertung des Ausdrucks `gruss.size()` als Ergebnis eine Ganzzahl (integer) ergibt, die der Anzahl der Zeichen in `gruss` entspricht.

Der zweite Ausdruck, `' '`, ist ein ***einzelnes konstantes Zeichen (character literal)***. Konstante einzelne Zeichen unterscheiden sich vollständig von ***konstanten Zeichenketten (string literal)***. Ein einzelnes Zeichen wird immer in einfachen Hochkommata eingeschlossen, während eine Zeichenkette immer von Doppelhochkommata umgeben ist. Der Typ eines Zeichens ist der eingebaute Datentyp ***char***, derjenige einer Zeichenkette ist sehr viel komplizierter, und wird deshalb an dieser Stelle noch nicht erläutert.

Der Typ ***char*** repräsentiert ein einzelnes Zeichen. Die einzelnen Zeichen innerhalb einer Zeichenkette, die eine besondere Bedeutung haben, haben genau diese Bedeutung auch als ***char***. Daher müssen Sie \ oder ' einen `backslash` voranstellen, sofern sie diese Zeichen selbst benötigen. Aus diesem Grund funktionieren '\n', '\t', '\'" und ähnliche Konstrukte genauso wie Sie es schon von Zeichenketten kennen.

Konstruieren Sie eine Zeichenkette von einem `char` und einem `int` Wert, so erhält das Ergebnis so viele Kopien des Zeichens, wie dem Wert der `int` Variablen entspricht. Beispielsweise ergäbe

```
std::string stars(10, '*');
```

folgendes Ergebnis: `**********` als Inhalt von `stars`.

In unserem Programm enthält also `space` genauso viele Zeichen wie `gruss`, aber alle Zeichen bestehen aus Leerzeichen (*blank*).

Die Definitionen von `zwei` und `eins` erfordern keine neuen Kenntnisse, ebenso sollten Ihnen der Rest des Programms bereits vertraut sein.

## 1.3 Details

### *Typen:*

- `char` Eingebauter Datentyp, der ein einzelnes Zeichen beinhaltet.

Der `string Typ` ist im Standard Header <string> definiert. Ein Objekt des Typs `string` enthält keines oder beliebig viele Zeichen. Folgende `string` Operationen kennen Sie bereits:

- `std::string s;`
  Definiert eine Variable vom Typ `std::string`, die keine Zeichen enthält.
- `std::string t = s;`
  Definiert eine Variable vom Typ `std::string`, die als Anfangswert eine Kopie der Zeichen von `s` enthält. `S` kann ein `string` oder eine Zeichenkette sein.
- `std::string z(n, c);`
  Definiert `z` als Variable vom Typ `std::string`, die als Anfangswert `n` Kopien des Zeichens `c` enthält. `c` **muss** als `char` definiert sein!
- `os << s`
  Schreibt die in `s` enthaltenen Zeichen auf den Ausgabestrom `os`.
- `is >> s`
  Liest und entfernt alle Zeichen aus dem Eingabestrom `is` bis ein Zeichen gefunden wird, das kein Zwischenraumzeichen (Whitespace) ist. Dann werden alle folgenden Zeichen von `is` nach `s` gelesen, bis das nächste Zeichen ein Zwischenraumzeichen wäre. Dabei wird der vorher vorhandene Wert von `s` überschrieben.
- `s + t`
  Das Ergebnis dieses Ausdrucks ist ein `std::string`, das eine Kopie aller Zeichen von `s` enthält, gefolgt von einer Kopie der Zeichen von `t`. Entweder `s` oder t, aber nicht beide zusammen, dürfen eine Zeichenkette oder ein einzelnes Zeichen sein.
- `s.size()`
  Das Ergebnis ist gleich der Anzahl der Zeichen in `s`.

***Variablen*** können auf drei verschiedene Arten definiert werden:

```
std::string hello = "Hello";
std::string stars(100, '*');
std::string name;
```

Variablen innerhalb von geschweiften Klammern sind ***lokale*** Variable, die nur während der Ausführung des Programmteils innerhalb der Klammern existieren. Wird die } Klammer erreicht, so werden die Variablen zerstört und der von ihnen belegte Speicherbereich an das Betriebssystem zurückgegeben.

Wird eine Variable als `const` definiert, so wird damit die Absicht dokumentiert, die Variable im weiteren Programmverlauf nicht mehr zu verändern. Solche Variablen müssen bei der Definition initialisiert werden.

### *Aufgaben*

1.1 Sind die folgenden Definitionen gültig? Warum oder warum nicht?

```
const std::string hello  = "Hello";
const std::string hello2 = "Hello" + " World !";
const std::string message = hello + ", world" + "!";
const std::string exclam   = "!";
const std::string message2 = "Hello" + ", World" + exclam;
```

1.2 Ist das folgende Programm ausführbar? Falls ja, was tut es? Falls nicht, warum nicht?

```
#include <iostream>
#include <string>
int main()
{
   { const std::string s = "a string";
     std::cout << s << std::endl; }

   { const std::string s = "another string";
     std::cout << s << std::endl; }
   return 0;
}
```

1.3 Ist das folgende Programm ausführbar? Was geschieht, wenn Sie in der dritten Zeile vom Ende }} in };} ändern?

```
#include <iostream>
#include <string>
int main()
{
   { const std::string s = "a string";
     std::cout << s << std::endl;

   { const std::string s = "another string";
     std::cout << s << std::endl; }}
   return 0;
}
```

1.4 Ist das folgende Programm ausführbar? Falls ja, was tut es? Falls nicht, ändern Sie es so, dass es funktioniert.

```
#include <iostream>
#include <string>
int main()
{
   { const std::string s = "a string";
   { const std::string x = s +", really";
     std::cout << s << std::endl; }
     std::cout << x << std::endl; }
   return 0;
}
```

1.5 Was macht das folgende Programm, wenn Sie auf die Frage nach Ihrem Namen, sofort Vor- und Nachname eingeben? Treffen Sie zuerst eine Vorhersage über das Verhalten des Programms und probieren Sie es anschließend aus!

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

int main()
{
   std::cout << "Name: ";
   std::string name;
   std::cin >> name;
   std::cout << "Hallo, " << name
             << std::endl << "Und nun Ihr Name? ";
   std::cin >> name;
   std::cout << "Hallo, " << name
        << "; nice to meet you too!" << std::endl;
   return 0;
}
```

# 2 Schleifen und Zählen

Im letzten Kapitel wurde ein Programm entwickelt, dass einen formatierten Rahmen um eine Begrüßung herum ausgibt. In diesem Kapitel wird das Programm etwas flexibler gestaltet, so dass die Größe des Rahmens geändert werden kann, ohne das Programm neu schreiben zu müssen.

## *2.1 Die Fragestellung*

Das Programm aus dem letzten Kapitel erzeugt einen Gruß mit einem formatierten Rahmen. Dabei wurde die Ausgabe zeilenweise erzeugt. Es wurden Variablen definiert, die die jeweiligen Ausgabezeilen enthielten.

Diese Vorgehensweise hat einen gravierenden Nachteil: Jede Ausgabezeile korrespondiert mit einem Programmteil und einer Variablen. Daher erfordert jede noch so kleine Änderung des Ausgabeformats ein Neuschreiben des Programms. Eine flexiblere Form der Ausgabe, bei der die einzelnen Ausgabezeilen **nicht** in Variablen abgespeichert werden, ist zu bevorzugen.

Wir werden das Problem lösen, indem jedes Ausgabezeichen separat erzeugt wird. Sie werden feststellen, dass es nicht notwendig ist die Ausgabezeichen in Variablen zu speichern, da ein Zeichen nicht mehr benötigt wird, sobald es ausgegeben wurde.

## *2.2 Die generelle Struktur*

Hier ist der Teil des Programms, der nicht geändert werden muss:

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

int main()
{
	std::cout << "Bitte geben Sie Ihren Vornamen ein: ";
	std::string name;
	std::cin >> name;

	// Begrüßungstext zusammenbauen
	const std::string gruss = "Hallo, " + name + "!";

	// Dieser Programmteil muss neu geschrieben werden

	return 0;
}
```

Der neu zu schreibende Teil beginnt ab dem `// Dieser Programmteil ...` Kommentar. `Name` und `gruss` sind hier bereits definiert und bekannt. Wir werden nun die neue Version des Programms Teil für Teil aufbauen und anschließend, in Kapitel 2.5.4, werden alle diese Teile zum kompletten Programm zusammengesetzt.

## *2.3 Die Ausgabe einer unbekannten Anzahl von Zeilen*

Sie können sich die Ausgabe als einen rechteckigen Bereich von Zeichen vorstellen, der zeilenweise ausgegeben werden muss. Wir wissen zwar nicht wie viele Zeilen die Ausgabe hat, aber wir wissen wie die Zeilenanzahl berechnet werden kann.

Die Begrüßung benötigt eine Zeile, ebenso die erste und letzte Zeile des Rahmens. Ist die Anzahl der beabsichtigten Leerzeilen zwischen Gruß und Rahmen bekannt, so können wir diese Anzahl verdoppeln, drei dazu addieren und erhalten so die Anzahl der Ausgabezeilen:

```
// Anzahl Leerzeichen die den Gruß umgeben
const int lzeichen = 1;

// Gesamtzahl der Ausgabezeilen
const int zeilen = lzeichen * 2 + 3;
```

Die Variable `lzeichen` beinhaltet die Anzahl der Leerzeichen, die den Gruß umgeben. `lzeichen` wird anschließend verwendet, um `zeilen`, die Gesamtzahl der Ausgabezeilen, zu berechnen.

`int` ist der eingebaute Datentyp für Ganzzahlen. Da nicht beabsichtigt ist, die Werte der Variablen zu ändern, sind `lzeichen` und `zeilen` als `const` definiert.

Wenn wir vereinbaren, dass die Anzahl der Leerzeichen (*blanks*) rechts und links des Grußes, gleich der Anzahl der Leerzeilen ober- und unterhalb des Grußes ist, so können wir diese Anzahl in einer Variablen abspeichern. Möchten wir nun die Größe des Rahmens ändern, so müssen wir nur dieser Variablen einen neuen Wert geben.

Die Anzahl der Ausgabezeilen ist berechnet, jetzt können wir mit der Ausgabe beginnen:

```
// Ausgabe von der Eingabe absetzen
std::cout << std::endl;

// Anzahl zeilen Zeilen ausgeben
int r = 0;

// Invariante: r Zeilen wurden bisher geschrieben
while (r != zeilen)
{
      // eine Ausgabezeile ausgeben (Erklärung folgt in Kapitel 2.4)
      std::cout << std::endl;
      r++;
}
```

Zuerst wird eine Leerzeile ausgegeben, damit die Ausgabe von der Eingabe durch eine Zeile getrennt ist. Der Rest dieses Codefragmentes enthält so viele neue Ideen, dass es sinnvoll ist, sich diese genauer anzuschauen. Wie genau eine einzelne Ausgabezeile erzeugt wird, überlegen wir in Kapitel 2.4.

### *2.3.1 Die* `while` *Anweisung*

Das Programm kontrolliert die Anzahl der Ausgabezeilen in einer ***`while`*** Anweisung, die solange bestimmte Anweisungen wiederholt, wie die in Klammern definierte Bedingung ***wahr*** (*true*) ist. Die `while` Anweisung hat folgende Form:

```
while (Bedingung)
      Anweisung
```

*Anweisung* wird auch der ***Rumpf*** von `while` genannt (***`while` body***).

Die `while` Anweisung bewertet zuerst die Bedingung. Ist die Bedingung falsch, wird der Rumpf überhaupt nicht ausgeführt. Andernfalls, wird der Rumpf einmalig ausgeführt, dann die Bedingung erneut bewertet und so weiter.

Zur leichteren Lesbarkeit des Programms, wird üblicherweise der Rumpf in eine eigene Zeile geschrieben und eingerückt.

*Anweisung* kann aus einer einzelnen Anweisung oder aus einem Block bestehen, also einer Folge von null oder mehr Anweisungen, die von { } umgeben sind.

`while` beginnt mit der Bewertung seiner *Bedingung*, die einen Ausdruck in einem Kontext darstellt, der den Wert *wahr* (*true*) erfordert. Der Ausdruck `r != Zeilen` ist ein Beispiel für eine Bedingung. Dieses Beispiel verwendet den ***ungleich Operator***, `!=`, um `r` und `Zeilen` zu vergleichen. Ein Ausdruck dieser Art, verwendet den Typ ***bool***, der den eingebauten Typ für logische (*wahr / falsch*) Werte repräsentiert. Die einzigen zwei möglichen Werte für Variable des Typs ***bool*** sind ***true*** oder ***false*** mit ihren offensichtlichen Bedeutungen.

Einer weitere Neuigkeit in diesem Programm finden Sie in der letzten Anweisung des `while` Rumpfes:

```
r++;
```

++ ist der ***Inkrement Operator***, der den Wert 1 zu einer Variable addiert. Wir hätten also genauso gut schreiben können:

```
r = r + 1;
```

Das Inkrementieren eines Objektes wird so häufig benötigt, dass ein eigener Operator dafür sinnvoll ist.

### *2.3.2 Der Entwurf der `while` Anweisung*

Es kann schwierig sein, die Bedingung einer `while` Anweisung exakt zu formulieren. Manchmal ist es auch schwierig präzise zu verstehen, was genau eine `while` Anweisung macht.

Es gibt erfreulicherweise eine sinnvolle Methode, um `while` Anweisungen zu verstehen und zu schreiben. Die Methode basiert auf zwei Ideen – eine über die Definition der `while` Anweisung, die andere über das Programmverhalten im Allgemeinen.

Die erste Idee sagt uns, dass nach Beenden einer `while` Anweisung die Bedingung `false` sein muss, denn sonst wäre die `while` Anweisung ja nicht beendet worden. In unserem Beispiel wissen wir also, dass `r != Zeilen false` ist, und somit `r` gleich `Zeilen` sein muss.

Die zweite Idee ist die einer ***Schleifen Invariante (loop invariant)***. Eine Invariante ist eine Eigenschaft, von der wir sicherstellen, dass sie jedes mal, wenn `while` seine Bedingung testet, ***wahr (true)*** ist. Wir wählen die Invariante so, dass wir davon überzeugt sind, dass sich das Programm in der gewünschten Weise verhält und umgekehrt schreiben wir das Programm genau so, dass die Invariante zu den richtigen Zeitpunkten wahr ist. Beachten Sie, dass die Invariante kein Teil des Programmtextes ist, aber ein wertvolles intellektuelles Werkzeug für den Programmentwurf darstellt. Jeder sinnvollen `while` Anweisung kann eine Invariante zugeordnet werden, die in einem Kommentar angegeben wird.

In unserem Beispielprogramm lautet der Kommentar zur `while` Schleife: `// Invariante: r Zeilen wurden bisher geschrieben`. Wir können überprüfen, ob dieses Invariante korrekt gewählt ist, indem wir verifizieren, dass die Invariante jedes mal wahr ist, wenn die Bedingung der `while` Schleife getestet wird. Dies erfordert, dass die Invariante an zwei Stellen im Programm überprüft wird.

Die erste Stelle ist unmittelbar bevor `while` seine Bedingung das erste mal überprüft. Da noch keine Ausgabezeilen geschrieben wurden, wird die Invariante offensichtlich durch die Anweisung `r = 0;` wahr.

Die zweite Stelle ist direkt vor dem Ende des `while` Blocks. Ist die Invariante an dieser Stelle wahr, so ist sie auch wahr, wenn die `while` Bedingung das nächste mal getestet wird und somit ist die Invariante immer wahr.

Die `while` Schleife kann also folgendermaßen formuliert werden:

```
// Invariante: r Zeilen wurden bisher geschrieben
int r = 0;

// durch r = 0 wird die Invariante an dieser Stelle wahr

while (r != zeilen) // wir können annehmen, dass die Invariante hier wahr ist
{
```

```
        // eine Ausgabezeile ausgeben, die Invariante wird dadurch falsch
        std::cout << std::endl;

        // Inkrementieren von r macht die Invariante wieder wahr
        r++;
    }

    // wir folgern, dass die Invariante hier wahr ist
```

Überprüfen Sie anhand des Programms und der Kommentare, dass die Annahmen über die Invariante richtig sind.

### *2.4 Die Ausgabe einer Zeile*

Nachdem Sie nun verstanden haben, wie man eine bestimmte Anzahl von Zeilen ausgibt, können wir uns jetzt auf den Inhalt einer einzelnen Zeile konzentrieren.

Sie können sich die Ausgabe als rechteckigen Bereich vorstellen, dessen Länge die Anzahl der Spalten einer Zeile darstellt. Alle Zeilen besitzen also die gleiche Länge. Die Länge einer Zeile können Sie berechnen, indem Sie zweimal die Anzahl der Leerzeichen vor dem Gruß zur Länge des Grußes addieren und zusätzlich zwei für die Sterne am Anfang und Ende der Zeile addieren:

```
const std::string::size_type spalten = gruss.size() + lzeichen * 2 + 2;
```

Der einfache Teil dieser Definition, ist die Tatsache, dass `spalten` konstant ist und somit der Wert von `spalten` nach der Definition nicht mehr geändert werden soll. Etwas schwieriger ist die Typdefinition zu verstehen, die für `spalten` verwendet wird: std::string::size_type. Sie kennen bereits den Bereichsoperator `::` und wissen dass `std::string` den Name `string` aus der Standardbibliothek zu verwenden. Der zweite `::` bedeutet ganz ähnlich, dass der Name `size_type` aus der Klasse `string` verwendet werden soll. Wie Namensbereiche und Blöcke definieren auch Klassen ihren eigenen Geltungsbereich. `size_type` ist der in `std::string` definierte geeignete Typ, um die Anzahl der Zeichen in einem `string` zu speichern. Immer wenn Sie eine Variable benötigen, um die Anzahl Zeichen eines `string` zu speichern, können Sie den Typ `std::string::size_type` verwenden.

Durch die Verwendung dieses Typs stellen wir sicher, dass `spalten` in der Lage ist, die Anzahl der Zeichen in `gruss` zu speichern, unabhängig wie groß `gruss` tatsächlich ist.

Da es nicht möglich ist, eine negative Anzahl von Zeichen in einem `string` zu erhalten, ist `std::string::size_type` ein ***vorzeichenloser (unsigned)*** Typ. Objekte dieses Typs sind nicht in der Lage negative Zahlen zu speichern.

Wir kennen jetzt die Anzahl der Zeichen und können sie in einer weiteren `while` Anweisung ausgeben:

```
std::string::size_type s = 0;

// Invariante: s Zeichen wurden bisher in der aktuellen Zeile ausgegeben
while (s != spalten)
{
      // eines oder mehrere Zeichen ausgeben
      // den Wert von s entsprechend der Invariante anpassen
}
```

Diese `while` Anweisung verhält sich analog zu derjenigen aus Kapitel 2.3 mit dem kleinen Unterschied, dass statt *einer* Zeile *mehrere* Zeichen ausgegeben werden. Es gibt hier keinen Grund nur ein einzelnes Zeichen auszugeben, solang wir sicherstellen, dass die Anzahl der Zeichen genau `spalten` entspricht.

### *2.4.1 Die Ausgabe der Umrandung*

Als letztes Problem müssen wir noch herausfinden, welche Zeichen auszugeben sind. Befinden wir uns in der ersten oder letzten Zeile oder in der ersten oder letzten Spalte, wissen wir, dass ein `'*'` auszugeben ist. Wir können zusätzlich unsere Kenntnis über die Invarianten verwenden, um zu bestimmen ob es an der Zeit ist, einen `'*'` auszugeben.

Ist beispielsweise `r` gleich null, so wissen wir aus der Invarianten, dass wir noch keine Ausgabezeile geschrieben haben, was bedeutet, dass wir uns noch in der ersten Zeile befinden. Genauso wissen wir, dass wir uns in der letzten Zeile befinden müssen, sofern `r` gleich `zeilen - 1` ist, denn dann haben wir bereits `zeilen - 1` Zeilen ausgegeben und sind somit in der letzten Zeile.

Mit den gleichen Überlegungen können wir annehmen, dass wir uns in der ersten Spalte befinden, sofern `s` gleich null ist und dass wir die letzte Spalte schreiben sofern `s` gleich `spalten - 1` ist. Mit diesem Wissen können wir einen weiteren Teil unseres Programms schreiben:

```
// Invariante: s Zeichen wurden bisher in der aktuellen Zeile ausgegeben
while (s != spalten)
{
      if (r == 0 || r == zeilen - 1 || s == 0 || s == spalten -1)
      {
            std::cout < "*";
            c++;
      }
      else
      {
      // eines oder mehrere Zeichen ausgeben, die nicht zur Umrandung gehören
            // den Wert von s entsprechend der Invariante anpassen
      }
}
```

Die zahlreichen neuen Ideen dieser Anweisungen werden wir nun detailliert erklären.

#### *2.4.1.1 Die `if` Anweisung*

Im Rumpf der `while` Anweisung befindet sich eine `if` Anweisung, die bestimmt, ob ein `'*'` auszugeben ist. Die `if` Anweisung hat zwei mögliche Formate:

```
if (Bedingung)
      Anweisung
```

oder, wie in unserem Beispiel,

```
if (Bedingung)
      Anweisung 1
else
      Anweisung 2
```

Genau wie bei der `while` Anweisung ist *Bedingung* ein Ausdruck, der einen Wahrheitswert als Ergebnis hat. Falls die Bedingung wahr ist, wird die Bedingung direkt nach dem `if` ausgeführt. Im zweiten Format wird die Anweisung nach `else` ausgeführt, sofern die Bedingung falsch war.

#### *2.4.1.2 Logische Operatoren*

Können Sie die `if` Bedingung selbst erklären:

```
r == 0 || r == zeilen - 1 || s == 0 || s == spalten -1
```

Die Bedingung wird *wahr*, falls `r` gleich `0` oder gleich `zeilen - 1` ist, oder falls `s` gleich `0` oder gleich `spalten - 1` ist. Die Bedingung verwendet zwei neue Operatoren: == und `||`. C++ Programme testen mit dem == Symbol auf Gleichheit im Unterschied zum Zuweisungsoperator =. Also liefert `r == 0` als Ergebnis einen *bool* Wert, der anzeigt, ob der Wert von `r` gleich 0 ist. Der ***logische-oder*** Operator, `||`, ergibt *true*, sofern ***einer*** seiner Operanden *true* ist.

Die logischen Operatoren haben einen niedrigeren ***Vorrang***, als die arithmetischen Operatoren. Der Vorrang definiert in Ausdrücken, die mehr als einen Operator verwenden, die Reihenfolge, in der die Operanden zu den Operatoren gruppiert werden.

Zum Beispiel :

```
r == zeilen - 1
```

bedeutet

```
r == (zeilen - 1)
```

und nicht

```
(r == zeilen) - 1
```

weil der arithmetische − Operator einen höheren Vorrang hat, als der logische Operator ==. Sie können den Vorrang durch Verwendung von Klammern ( '(' und ')' ) verändern.

Der logische-oder Operator testet ob irgendeiner seiner Operanden *wahr* ist. Das Format ist

Bedingung 1 `||` Bedingung 2

wobei Bedingung 1 und Bedingung 2 *Bedingungen* sind – Ausdrücke, die einen Wahrheitswert als Ergebnis haben. Der `||` Ausdruck selbst, hat ein Ergebnis vom Typ *bool*, das entweder wahr (*true*) oder falsch (*false*) ist.

Betrachten Sie nun die vier Abfragen auf Gleichheit in unserem Programm, so stellen Sie fest, dass die vier Abfragen prüfen, ob wir in der ersten Zeile, letzten Zeile, ersten Spalte oder letzten Spalte sind. Liefert eine der vier Abfragen *true* als Ergebnis, schreibt die `if` Abfrage einen `'*'`, andernfalls wird etwas anderes ausgegeben, das wir erst noch definieren müssen.

### *2.4.2 Die Ausgabe von Zeichen innerhalb der Umrandung*

Nun können die Anweisungen geschrieben werden, die zu folgenden Kommentaren korrespondieren:

```
// eines oder mehrere Zeichen ausgeben, die nicht zur Umrandung gehören
// den Wert von s entsprechend der Invariante anpassen
```

Diese Anweisungen behandeln alle Zeichen, die nicht Teil der Umrandung sind, also entweder Leerzeichen oder den Grußtext. Das Problem besteht also darin, herauszufinden ob Leerzeichen oder der Gruß ausgegeben werden sollen.

Wie überprüfen zuerst, ob wir dabei sind das erste Zeichen des Grußes zu schreiben, indem wir testen ob wir in der richtigen Zeile und Spalte sind. Die Zeile die wir suchen ist diejenige, nach der ersten Zeile (mit den '*'), gefolgt von `lzeichen` zusätzlichen Zeilen. Die richtige Spalte finden wir, nachdem der `'*'` am Zeilenanfang und `lzeichen` Leerzeichen geschrieben wurden. Mit anderen Worten, wir müssen überprüfen, ob `r` und `s` gleich `lzeichen + 1` sind. Ist dies richtig, wird der Gruß ausgegeben, sonst einfach ein Leerzeichen. In beiden Fällen müssen wir daran denken, r und s zu aktualisieren:

```
if (r == lzeichen + 1 && s == pad + 1)
{
      std::cout << gruss;
      s += gruss.size();
}
```

```
else
{
      std::cout << " ";
      s++;
}
```

Die Bedingung innerhalb der `if` Anweisung verwendet den ***logischen – und*** Operator. Der `&&` Operator bewertet zwei Bedingungen und ergibt einen Wahrheitswert als Ergebnis, genau wie der `||` und der == Operator. Der `&&` Operator liefert *true*, wenn beide Bedingungen wahr sind und *false*, falls ***eine*** der Bedingungen falsch ist.

War der Test erfolgreich, so können wir den Gruß ausgeben. Dadurch wird die Invariante *false*, da `s` nun nicht mehr gleich der Anzahl ausgegebener Zeichen ist. Wir können die Invariante wieder *wahr* machen, indem wir den Wert von `s` an die Anzahl der ausgegebenen Zeichen anpassen. Die Anweisung, die `s` aktualisiert verwendet einen weiteren neuen Operator: +=, genannt ***compound-assignment*** Operator. Dieser Operator ist eine abkürzende Schreibweise für die Anweisung die rechte **und** linke Seite zu addieren und anschließend das Ergebnis in der linken Seite zu speichern. Die Anweisung `s += gruss.size();` ist also äquivalent zu `s = s + gruss.size();`.

Die letzte noch nicht bearbeitete Möglichkeit besteht darin, dass wir nicht auf dem Rand sind **und** nicht dabei sind den Gruß zu schreiben. In diesem Fall wird ein Leerzeichen ausgegeben und `s` inkrementiert, wie Sie in dem `else` Block der `if` Anweisung sehen können.

### *2.5 Das komplette Programm*

Bevor Sie gleich das ganze Programm sehen, werden wir es auf drei Arten verkürzen.

Zuerst werden Sie eine Deklaration kennen lernen, die es ermöglicht auf `std::` zu verzichten. Die zweite Abkürzung betrifft die `while` Anweisung. Drittens kann das Programm etwas verkürzt werden, indem Sie `s` an einer Stelle inkrementieren, statt an zwei Stellen.

#### *2.5.1 Einmalige Verwendung von* `std::`

Sie verwenden `std::` um dem Compiler mitzuteilen, dass ein Name aus der Standardbibliothek verwendet werden soll. C++ bietet Ihnen die folgende Möglichkeit um festzulegen, dass ein Name immer aus der Standardbibliothek verwendet werden soll:

```
using std::cout;
```

Mit dieser Anweisung legen Sie fest, dass die Verwendung von `cout` immer `std::cout` bedeutet. Nach dieser Anweisung können Sie also `cout` schreiben und müssen nicht mehr `std::cout` verwenden.

#### *2.5.2 Die Verwendung der* `for` *Anweisung*

Betrachten Sie den folgenden Teil des Programms:

```
int r = 0;

while (r != zeilen)
{
      // Anweisungen, die r nicht verändern
      r++;
}
```

Diese Form der `while` Anweisung erscheint häufig. Vor der Anweisung wird eine lokale Variable definiert und initialisiert, die dann in der Bedingung überprüft wird. Im Rumpf der `while` Anweisung wird die Variable dann so verändert, dass irgendwann die Bedingung falsch wird. Für diese Kontrollstruktur gibt es folgende kürzere Schreibweise:

```
for (int r = 0; r != zeilen; r++)
{
      // Anweisungen, die r nicht verändern
}
```

`r` nimmt im Rumpf der Anweisung eine Sequenz von Werten an und zwar von 0 bis `zeilen - 1`. Die `for` Anweisung hat folgende allgemeine Form:

```
for (Initialisierungsausdruck; Bedingung; Ausdruck)
      Anweisung
```

Die `for` Anweisung führt zuerst ***einmalig*** die Initialisierungsanweisung aus. Bei jedem Schleifendurchlauf, auch beim ersten, wird die Bedingung bewertet. Ist das Ergebnis *true*, so wird der Rumpf ausgeführt. Danach wird A*usdruck* ausgeführt. Anschließend wird der Test wiederholt und der Rumpf ausgeführt und so weiter bis schließlich der Test irgendwann *false* ergibt.

### *2.5.3 Zusammenfassung von zwei Inkrementierungen*

Es ist oft möglich die Reihenfolge von Test innerhalb eines Programms zu vertauschen und dabei zwei oder mehr identische Anweisungen in einer einzigen Anweisung zu komprimieren.

Da sich unsere drei Testfälle gegenseitig ausschließen, können sie in beliebiger Reihenfolge ausgeführt werden. Wenn wir zuerst testen, ob wir gerade den Gruß schreiben, dann wissen wir dass es in den beiden anderen Fällen ausreichend ist `s` zu inkrementieren, um die Invariante zu erhalten. Folglich können zwei Inkrement Anweisung in einer zusammengefasst werden:

```
if (wir sind dabei den Gruß zu schreiben)
{
      cout << gruss;
      c += gruss.size();
}
else
{
      if (wir sind beim Rand)
            cout << "*";
      else
            cout << " ";
      c++;
}
```

Bitte beachten Sie, wie die Einrückungen die Übersichtlichkeit erhöhen.

### *2.5.4 Das ganze Programm*

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

// Definitionen aus der Standardbibliothek

using std::cin;
using std::cout;
using std::endl;
using std::string;
```

```
int main()
{
	cout << "Bitte geben Sie Ihren Vornamen ein: ";
	string name;
	cin >> name;

	// Begrüßungstext zusammenbauen
	const string gruss = "Hallo, " + name + "!";

	// Anzahl Leerzeilen die den Gruß umgeben
	const int lzeichen = 1;

	// Gesamtzahl der Ausgabezeilen und Spalten
	const int zeilen = lzeichen * 2 + 3;
	const string::size_type spalten = gruss.size() + lzeichen * 2 + 2;

	// Ausgabe von der Eingabe absetzen -> Leerzeile
	cout << endl;


	// Anzahl 'zeilen' Zeilen ausgeben
	// Invariante: r Zeilen wurden bisher geschrieben


	for (int r = 0; r != zeilen; r++)
	{
		string::size_type s = 0;

		// Invariante: s Zeichen wurden bisher in der Zeile ausgegeben
		while (s != spalten)
		{
			// Ist der Gruß auszugeben?
			if (r == lzeichen + 1 && s == lzeichen + 1)
			{
				cout << gruss;
				s += gruss.size();
			}
			else
			{
				// sind wir auf dem Rand?
				if (r == 0 || r == zeilen - 1 ||
				    s == 0 || s == spalten -1)
				  cout << "*";
				else
					cout << " ";
				s++;
			}

		}
		cout << endl;
	}

	return 0;
}
```

### *2.6 Zählen mit C++*

Viele erfahrene C++ Programmierer besitzen eine Angewohnheit, die auf den ersten Blick etwas seltsam erscheinen mag: Ihre Programme beginnen beständig mit `0` zu zählen und nicht mit `1`. Betrachten Sie zum Beispiel die äußere `for` Schleife des Programms:

```
for (int r = 0; r != zeilen; r++)
{
    // eine Zeile ausgeben
}
```

Diese Zeile hätten wir auch so schreiben können:

```
for (int r = 1; r <= zeilen; r++)
{
    // eine Zeile ausgeben
}
```

Eine Version zählt von `0` und verwendet `!=` für ihren Vergleich, die andere zählt von `1` und verwendet `<=` für ihren Vergleich. Die Anzahl der Iterationen ist in jedem Fall die gleiche. Gibt es irgend einen Grund eine Version der anderen vorzuziehen?

Ein Grund mit dem Zählen bei `0` zu beginnen ist, dass dadurch die Invariante leichter auszudrücken ist. In unserem Beispiel ist die Invariante: *r Zeilen wurden bisher geschrieben*. Wie würde die Invariante aussehen, wenn wir bei `1` mit dem Zählen beginnen würden?

Man könnte glauben die Invariante wäre, dass wir *dabei* sind, die r-te Zeile zu schreiben, aber diese Aussage eignet sich nicht als Invariante. Der Grund dafür ist, dass beim letzten mal, wenn die `while` Schleife ihre Bedingung testet, `r` gleich `zeilen + 1` ist, und wir die Absicht haben `zeilen` Zeilen auszugeben. Wir sind also *nicht dabei* die r-te Zeile auszugeben und die Invariante ist falsch!

Die Invariante könnte auch sein, dass wir bisher `r - 1` Zeilen geschrieben haben, aber dies können wir vereinfachen indem wir bei `0` mit dem Zählen beginnen.

Ein weiterer Grund bei `0` mit dem Zählen zu beginnen, ist dass wir damit die Möglichkeit haben `!=` anstatt `<=` als Vergleichsoperator zu verwenden. Dieser Unterschied mag Ihnen trivial erscheinen, aber er beeinflusst unsere Kenntnis über den Zustand des Programms, wenn die Schleife beendet ist.

Ist beispielsweise die Bedingung `r != zeilen`, so wissen wir, dass nach Ende der Schleife `r == zeilen` gilt. Weil die Invariante lautet, dass wir `r` Zeilen geschrieben haben, so wissen wir, dass genau `zeilen` Zeilen ausgegeben wurden. Andererseits, wenn die Bedingung wäre `r <= zeilen`, dann könnten wir nur sicher sein *mindestens* `r` Zeilen geschrieben zu haben, es könnten aber jederzeit auch mehr sein!

### *2.7 Details*

***Ausdrücke:*** C++ stellt eine Reihe von Operatoren zur Verfügung, einige davon haben sie bereits kennengelernt. Zusätzlich, wie Sie bereits mit den Ein- und Ausgabe Operatoren sehen konnten, können C++ Programme die Kernsprache erweitern, indem sie Operatoren für bestimmte Datentypen definieren. Sofern Sie effektiv mit C++ programmieren möchten, ist es eine fundamentale Voraussetzung, dass Sie komplexe Ausdrücke korrekt verstehen. Solche Ausdrücke verstehen erfordert zu verstehen

- wie Operanden gruppiert werden, was durch Vorrang und Assoziativität von Operatoren definiert wird
- wie Operanden in andere Typen konvertiert werden, falls überhaupt
- die Reihenfolge in der Operanden ausgewertet werden

Verschiedene Operatoren haben unterschiedlichen Vorrang. Die meisten Operatoren sind links-assoziativ, aber Zuweisungsoperatoren und Operatoren mit einem einzigen Argument sind rechts-assoziativ. Es folgt eine Liste der am häufigsten verwendeten Operatoren, unabhängig davon, ob wir Sie bereits verwendet haben. Die Operatoren sind nach Vorrang geordnet, beginnend mit dem höchsten Vorrang. Eine Doppellinie trennt Gruppen mit gleichem Vorrang.

| | |
|---|---|
| `x.y` | The member `y` of object `x` |
| `x[y]` | The element in object `x` indexed by `y` |
| `x++` | Increments `x`, returning the original value of `x` |
| `x--` | Decrements `x`, returning the original value of `x` |
| `++x` | Increments `x`, returning the incremented value |
| `--x` | Decrements `x`, returning the decremented value |
| `!x` | Logical negation. If `x` is `true` then `!x` is `false`. |
| `x * y` | Product of `x` and `y` |
| `x / y` | Quotient of `x` and `y`. If both operands are integers, the implementation chooses whether to round toward zero or $-\infty$. |
| `x % y` | Remainder of `x` divided by `y`, equivalent to `x - ((x / y) * y)` |
| `x + y` | Sum of `x` and `y` |
| `x - y` | Result of subtracting `y` from `x` |
| `x >> y` | For integral `x` and `y`, `x` shifted right by `y` bits; `y` must be non-negative. If `x` is an `istream`, reads from `x` into `y`. |
| `x << y` | For integral `x` and `y`, `x` shifted left by `y` bits; `y` must be non-negative. If `x` is an `ostream`, writes `y` onto `x`. |
| `x` *relop* `y` | Relational operators yield a `bool` indicating the truth of the relation. The operators (<, >, <=, and >=) have their obvious meanings. |
| `x == y` | Yields a `bool` indicating whether `x` equals `y` |
| `x != y` | Yields a `bool` indicating whether `x` is not equal to `y` |
| `x && y` | Yields a `bool` indicating whether both `x` and `y` are `true`. Evaluates `y` only if `x` is `true`. |
| `x \|\| y` | Yields a `bool` indicating whether either `x` or `y` is `true`. Evaluates `y` only if `x` is `false`. |
| `x = y` | Assign the value of `y` to `x`, yielding `x` as its result. |
| `x` *op*`= y` | Compound assignment operators; equivalent to `x = x` *op* `y`, where *op* is an arithmetic or shift operator. |
| `x ? y : z` | Yields `y` if `x` is `true`; `z` otherwise. Evaluates only one of `y` and `z`. |

Sofern möglich, werden Operatoren in einen angemessenen Typ umgewandelt. Numerische Operanden werden nach den ***Regeln für arithmetische Umwandlung*** konvertiert, die Sie im Detail erst später kennenlernen werden. Grundsätzlich wird bei diesen Umwandlung versucht, die Genauigkeit zu erhalten. Kleinere Typen werden in größere umgewandelt, Typen mit Vorzeichen werden in vorzeichenlose Typen umgewandelt. Arithmetische Werte können in `bool` umgewandelt werden: der Wert `0` wird zu `false`, alle anderen Werte werden zu `true`.

### *Typen:*

- `bool` eingebauter Typ für Wahrheitswerte; kann `true` oder `false` sein
- `unsigned` Ganzzahltyp, der nur positive Werte enthält
- `short` Ganzzahltyp, der mindestens 16 Bit groß ist
- `long` Ganzzahltyp, der mindestens 32 Bit groß ist
- `size_t` Ganzzahltyp ohne Vorzeichen (aus `<cstddef>`, der die Größe eines beliebigen Objektes aufnehmen kann
- `string::size_type` Ganzzahltyp ohne Vorzeichen, der die Länge eines beliebigen `string` aufnehmen kann.

### *Bedingung:*

Ein Ausdruck, der einen Wahrheitswert als Ergebnis hat. Arithmetische Werte in Bedingungen werden in den Typ `bool` umgewandelt: `0` wird nach `false` konvertiert, alle anderen Werte nach `true`.

### *Anweisungen:*

`using` *Namensraum-Name* `::` N*ame;*

Definiert *Name* als Synonym für *Namensraum-Name* `::` *Name.*

*Typ-Name Name;*

Definiert *Name* mit dem Typ *Typ-Name.*

*Typ-Name Name = Wert;*

Definiert *Name* mit dem Typ *Typ-Name* und initialisiert mit einer Kopie von *Wert*.

*Typ-Name Name(Argumente);*

Definiert *Name* mit dem Typ *Typ-Name* und konstruiert *Name* so, wie es sich aus den in *Argumente* übergebenen Argumenten ergibt.

*Ausdruck;*

Berechnet *Ausdruck* incl. der Seiteneffekte.

`{` *Anweisung(en)* `}`

Wird Block genannt. Führt eine Folge von null oder mehreren Anweisung in der angegebenen Reihenfolge aus und kann an allen Stellen verwendet werden an denen *Anweisung* erwartet wird. Der Geltungsbereich von Variablen, die innerhalb eines Blocks definiert werden, ist auf den Block begrenzt.

`while` (*Bedingung*) *Anweisung*

Falls *Bedingun*g gleich `false` ist, tue nichts; sonst führe *Anweisung* aus und dann wiederhole `while`.

`for` (*Initialisierungsausdruck; Bedingung; Ausdruck*)

Äquivalent zu `{` *Initialisierungsausdruck;* `while` (*Bedingung*) `{` *Ausdruck* `} }`

`if` (*Bedingung*) *Anweisung*

Führt *Anweisung* aus, falls *Bedingung* gleich `true` ist.

`if` (*Bedingung*) *Anweisung1* `else`

Führt *Anweisung* aus, falls *Bedingung* gleich `true` ist, andernfalls wird *Anweisung2* ausgeführt.

`return` *Wert;*

Beendet die Funktion und gibt *Wert* an den Aufrufer der Funktion zurück.

### *Aufgaben*

2.1 Übersetzen Sie das Programm aus diesem Kapitel und starten Sie es.

2.2 Ändern Sie das Programm so, dass es den Gruß ohne Zwischenraum zum Rahmen ausgibt.

2.3 Ändern Sie das Programm so, dass es eine andere Anzahl Zwischenräume zur Trennung der Seiten vom Gruß verwendet, als es Zeilen verwendet, um die erste und letzte Zeile vom Gruß zu separieren.

2.4 Ändern Sie das Programm so, dass es den Anwender auffordert, die Anzahl der Zwischenräume einzugeben.

2.5 Das Programm schreibt die fast leeren Zeilen, die den Gruß von der ersten und letzten Zeile trennen, Zeichen für Zeichen. Ändern Sie das Programm so, dass es alle Leerzeichen mit einer einzigen Ausgabeanweisung erzeugt.

2.6 Schreiben Sie eine Anzahl von "*" Zeichen, so dass sie ein Quadrat, ein Rechteck und ein Dreieck ergeben.

2.7 Was macht das folgende Programm:

```
int i = 0;
while (i < 10)
{
    i += 1;
    std::cout << i  << std:endl;
}
```

2.8 Schreiben Sie ein Programm das von `10` abwärts nach `-5` zählt.

2.9 Erklären Sie jede Verwendung von `std::` im folgenden Programm:

```
int main()
{
    int k = 0;
    while (k != 10)  // Invariante: Wir haben bisher k '*' geschrieben
    {
        using std::cout;
        cout << "*";
        k++;
    }
    std::cout << std::endl;
    return 0;
}
```

# 3 Die Arbeit mit Datenmengen

Die Programme aus Kapitel 1 und 2 konnten nicht viel mehr, als einen Text einzulesen und ihn wieder auszugeben, manchmal mit etwas Dekoration. Fast alle technischen und wissenschaftlichen Fragestellungen sind komplizierter und mit solch einfachen Programmen nicht lösbar. Eine der häufigsten Quellen von Komplexität in Programmen ist die Anforderung, eine Vielzahl von ähnlichen Daten zu bearbeiten.

Wenn Sie eine Zeichenkette (string) als eine Vielzahl von einzelnen Zeichen verstehen, sind unsere Programm schon in der Lage auf diese Anforderung zu reagieren. Tatsächlich waren die Programme deshalb so einfach zu schreiben, weil die Möglichkeit bestand, eine unbekannte Anzahl von Zeichen in einem Objekt – einem `string` – zu behandeln.

In diesem Kapitel werden Sie weitere Möglichkeiten kennenlernen eine Vielzahl von Daten zu bearbeiten, indem wir ein Programm entwickeln, das die Noten eines Studenten aus Vordiplom, Diplom und WP-Fächern einliest und daraus die Gesamtnote berechnet. Dabei werden Sie lernen, wie man alle Noten speichert, sogar wenn vorab nicht bekannt ist, wie viele Noten vorhanden sind.

## *3.1 Berechnung der Gesamtnote*

Stellen Sie sich einen Studiengang vor, in dem sich die Gesamtnote zu 40% aus der Diplom-Note, zu 20% aus der Vordiplom-Note und zu 40% aus dem Mittelwert aller WP-Noten errechnet. Hier ist die erste Version eines Programms, das die Gesamtnote nach dieser Vorschrift berechnet:

```
#include <iomanip>
#include <ios>
#include <iostream>
#include <string>

using std::cin;         using std::setprecision;
using std::cout;        using std::string;
using std::endl;        using std::streamsize;

int main()
{
      // den Namen des Studenten erfragen
      cout << "Bitte gib Deinen Vorname ein: ";

      string name;
      cin >> name;

      cout << "Tach auch, " << name << "!?!" << endl;

      // Noten aus Vordiplom- und Diplomprüfung erfragen
      cout << "Bitte gib deine Vordiplom- und Diplomnote ein: ";

      double  vordiplom,
              diplom;

      cin >> vordiplom >> diplom;

      // Noten der WP-Fächer erfragen
      cout << "Bitte die Noten aller WP-Fächer eingeben "
                  "und mit end-of-file beenden: ";

      // Anzahl und Summe der Noten
      int    anzahl = 0;
      double summe  = 0;

      // Variable zum Einlesen der Noten
      double x;
```

```
        // WP-Noten einlesen
        while (cin >> x)
        {
              anzahl++;
              summe += x;
        }

        streamsize prec = cout.precision();

        cout << "Deine Gesamtnote ist: " << setprecision(2)
               << 0.2 * vordiplom + 0.4 * diplom + 0.4 * summe / anzahl
               << setprecision(prec) << endl;

        return 0;
}
```

Zu Beginn stehen wie immer die `#include`-Anweisungen und `using`-Deklarationen für die verwendeten Funktionalitäten aus der Standardbibliothek. Neu sind der Header `<ios>`, der `streamsize` definiert, ein Typ, der von der IO-Bibliothek für Größenangaben verwendet wird und der Header `<iomanip>`, der den Manipulator `setprecision` definiert, der uns ermöglicht die Anzahl signifikanter Stellen eines Ausgabestroms einzustellen.

Das Programm beginnt mit der Frage nach dem Vornamen des Studenten und liest anschließend den Vornamen sowie die Noten von Vordiplom, Diplom und einer unbekannten Anzahl von WP-Fächern bis *end-of-file* signalisiert wird. Es gibt verschiedene Möglichkeiten ein *end-of-file* Signal zu erzeugen, die einfachste Möglichkeit besteht darin, eine neue Eingabezeile zu beginnen, die *strg*-Taste (*control-key*) gedrückt zu halten und dann den Buchstaben *d* einzugeben.

Während die WP-Noten eingelesen werden, speichert das Programm die Anzahl der WP-Noten in der Variablen `anzahl` und die Summe aller WP-Noten in der Variablen `summe`. Nach Eingabe aller WP-Noten gibt das Programm eine Höflichkeitsfloskel und die Gesamtnote aus. Zur Berechnung der Gesamtnote werden `anzahl` und `summe` verwendet.

Vieles in diesem Programm ist bereits bekannt, aber es gibt einige neue Dinge, die nun erklärt werden.

Die erste neue Idee ist in dem Programmteil enthalten, der die Prüfungsnoten einliest:

```
cout << "Bitte gib deine Vordiplom- und Diplomnote ein: ";

double  vordiplom,
        diplom;

cin >> vordiplom >> diplom;
```

Die erste Anweisung ist bekannt: sie schreibt eine Meldung, die zur Eingabe der Vordiplom- und Diplomnoten auffordert. Die nächste Anweisung definiert `vordiplom` und `diplom` als Variable vom Typ `double`, dem eingebauten Typ für *doppelt-genaue* Fließkomma-Zahlen. Es gibt auch einen Typ für *einfach-genaue* Fließkomma-Zahlen, genannt `float`. Auch wenn Ihnen manchmal `float` als der angemessene Typ erscheint, ist es fast immer richtig `double` zu verwenden.

Die Namen dieser Typen stammen aus einer Zeit, als Speicherplatz *sehr* viel teurer war als heute. Der kleinere Fließkomma-Typ, `float`, kann sechs signifikante Dezimalstellen aufnehmen, zu wenig um den Preis eines Mittelklassewagens auf den Cent genau anzugeben. Der Typ `double` garantiert *mindestens* zehn signifikante Dezimalstellen, alle gängigen C++-Compiler verwenden mindestens fünfzehn Dezimalstellen. Auf modernen Computern ist `double` also sehr viel genauer als `float` und nur wenig langsamer, auf manchen Prozessoren sogar schneller.

Nachdem die Variablen `vordiplom` und `diplom` definiert sind, können wir Werte in sie einlesen. Wie der Ausgabeoperator liefert auch der Eingabeoperator seinen linken Operanden als Ergebnis. Daher können Eingabeoperationen genauso wie Ausgabeoperationen verkettet werden:

```
cin >> vordiplom >> diplom;
```

hat somit den gleichen Effekt, wie

```
cin >> vordiplom;
cin >> diplom;
```

Die nächste Anweisung fordert den Studenten auf, die WP-Noten einzugeben:

```
cout << "Bitte die Noten aller WP-Fächer eingeben "
            "und mit end-of-file beenden: ";
```

Es werden offensichtlich zwei konstante Zeichenketten mit einem Eingabeoperator (<<) ausgegeben. Dies funktioniert, weil zwei oder mehr konstante Zeichenketten, die nur durch Leerzeichen getrennt sind, automatisch verkettet werden.

Das nächste Code-Fragment definiert die Variablen, die benötigt werden, um die Eingabe einzulesen:

```
int    anzahl = 0;
double summe  = 0;
```

Beachten Sie, dass beide Variablen den Anfangswert `0` erhalten. Der Wert `0` ist vom Typ `int`, dies bedeutet, dass der Wert implizit in den Typ `double` umgewandelt werden muss, um in der Initialisierung von `summe` verwendet werden zu können. Wir hätten `summe` natürlich auch mit `0.0` initialisieren können.

Es ist sehr wichtig, dass diese Variablen einen Anfangswert erhalten. Wird kein Anfangswert angegeben, so verlässt man sich automatisch auf die ***Standard-Initialisierung***. Die Standard-Initialisierung hängt vom Typ ab. Für Objekte vom Typ *Klasse* wird in der Klasse beschrieben, welcher Initialisierungswert verwendet werden soll, sofern bei der Variablendefinition kein Anfangswert angegeben wird. Beispielsweise werden Variable vom Typ `string` implizit mit einem Leerstring initialisiert.

Der Anfangswert von lokalen Variablen von eingebauten Typen (`int, double` etc.), die nicht *explizit* initialisiert werden, ist ***nicht*** definiert, d.h. der Wert der Variable besteht aus dem Wert, der sich zufällig an dem Speicherplatz befindet, an dem die Variable angelegt wird.

Ein weiterer neuer Aspekt ist die Form der Bedingung in der `while` – Anweisung:

```
// WP-Noten einlesen
while (cin >> x)
{
      anzahl++;
      summe += x;
}
```

Wir wissen bereits, dass die `while` – Schleife solange ausgeführt wird, wie die Bedingung `cin >> x` *wahr* ist. Was es bedeutet `cin >> x` als Bedingung zu verwenden, werden wir im nächsten Absatz ausführlich erläutern. Hier genügt es, sich zu merken, dass die Bedingung wahr ist, sofern die letzte Eingabeaufforderung (`cin >> x`) erfolgreich bearbeitet wurde.

Innerhalb der `while` – Schleife werden der Inkrement- und der Zuweisungsoperator, beide bereits aus Kapitel 2 bekannt, verwendet.

Jetzt ist nur noch zu erklären, wie das Programm die Ausgabe durchführt:

```
streamsize prec = cout.precision();

cout << "Deine Gesamtnote ist: " << setprecision(2)
     << 0.2 * vordiplom + 0.4 * diplom + 0.4 * summe / anzahl
     << setprecision(prec) << endl;
```

Die Ausgabe soll mit zwei signifikanten Ziffern erfolgen. Dies erreichen wir durch die Anweisung `setprecision(2)`. Genau wie `endl` ist auch `setprecision` ein Manipulator. Er manipuliert den Ausgabestrom so, dass alle folgenden Ausgaben mit der als Argument angegebenen Genauigkeit erfolgen.

Es ist grundsätzlich sinnvoll die Genauigkeit des Ausgabestroms `cout` wieder auf den vor der Änderung verwendeten Wert zurückzusetzen. Dies erreichen wir mit der zu `cout` gehörenden Elementfunktion `precision`. Diese Funktion liefert uns die Genauigkeit, die ein Stream für die Ausgabe von Fließkommazahlen verwendet. Wir verwenden `setprecision(2)`, um die Genauigkeit auf 2 zu setzen, geben die Gesamtnote aus und setzen dann die Genauigkeit zurück auf den ursprünglich Wert, den wir von `precision` erhalten hatten.

### *3.1.1 Test auf das Ende der Eingabe*

Konzeptionell ist die Bedingung der `while` – Anweisung der einzig wirklich neue Aspekt des Programms. Die Bedingung verwendet implizit ein `istream` Objekt als Subjekt der `while` – Anweisung:

```
while (cin >> x)
```

Diese Anweisung versucht von der Standardeingabe `cin` zu lesen. War das Lesen erfolgreich, so enthält die Variable `x` den gelesenen Wert und das Ergebnis der `while` – Bedingung ist `true`. War das Lesen *nicht* erfolgreich (weil das Eingabeende erreicht war, oder weil die Eingabe ungültig für den Typ der Variablen `x` war), dann ist das Ergebnis der `while` – Bedingung `false` und `x` enthält keinen gültigen Wert.

Diesen Teil des Programms zu verstehen, ist etwas schwierig. Erinnern wir uns, dass der >> Operator seinen linken Operanden als Rückgabewert hat, d.h. wir erhalten das Ergebnis von `cin >> x` indem wir `cin >> x` ausführen und *anschließend* nach dem Wert von `cin` fragen. Beispielsweise können wir mit folgendem Code einen einzelnen Wert nach `x` einlesen und überprüfen, ob diese Anweisung erfolgreich war:

```
if (cin >> x)
```

Diese Anweisung hat exakt die gleiche Bedeutung wie

```
cin >> x;
if (cin)
```

oder wie

```
cin >> x;
if (cin == true)
```

Wird `cin >> x` innerhalb einer Bedingung verwendet, so wird nicht nur die Bedingung bewertet, sondern als Seiteneffekt ein Wert in die Variable `x` eingelesen. Wir müssen jetzt also noch verstehen, was es bedeutet `cin` als Bedingung zu verwenden.

Die Bedingungen aus Kapitel 2 verwendeten Vergleichsoperatoren, die als Ergebnis den Typ `bool` hatten. Zusätzlich können wir arithmetische Ausdrücke in Bedingungen verwenden. Der arithmetische Wert wird innerhalb einer Bedingung automatisch in den Typ `bool` umgewandelt: Werte ungleich null werden in `true` umgewandelt, null wird in `false` umgewandelt. `cin` ist vom Typ `istream`. Ohne den Typ `istream` im Detail zu kennen, genügt es zu wissen, dass Werte vom Typ `istream` ebenfalls nach `bool` umgewandelt werden können. Ohne den Wert von `cin` genau zu kennen, wissen wir also, dass `cin` in einer Bedingung verwendet werden kann. Der Wert der Umwandlung hängt vom internen Status des `istream` Objektes, also von `cin`, ab. Der interne Status von `cin` berücksichtigt, ob der letzte Lesevorgang erfolgreich war. Die Verwendung von `cin` innerhalb einer Bedingung ist somit äquivalent zu einem Test, ob die letzte Leseoperation von `cin` erfolgreich durchgeführt werden konnte.

Es gibt mehrere Gründe, warum der Versuch von einem Stream zu lesen, scheitern kann:

- Das Ende der Eingabe ist erreicht.
- Der Typ der Eingabe stimmt nicht mit dem Typ der Variablen überein, in die eingelesen werden soll, z.B. wenn ein Buchstabe in eine Variable vom Typ `int` eingelesen werden soll.
- Das System hat einen Hardwarefehler entdeckt.

In jedem dieser Fälle ist das Ergebnis das gleiche: Die Verwendung des Eingabestroms `cin` in einer Bedingung wird anzeigen, dass die Bedingung falsch ist. Zudem werden alle weiteren Versuche von dem Eingabestrom zu lesen solange fehlschlagen, bis der Strom neu initialisiert wird. In Kapitel 4 werden Sie lernen wie das zu bewerkstelligen ist.

## *3.2 Die Verwendung des Medians anstelle des Durchschnitts*

Das bisher verwendete Programm hat eine Schwachstelle: Jede einzelne Note eines WP-Faches wird sofort nach dem Einlesen wieder gelöscht bzw. überschrieben. Sofern Durchschnittswerte berechnet werden sollen ist diese Vorgehensweise ausreichend, aber was wäre wenn wir z.B. den Median der WP-Noten berechnen wollten und nicht die Durchschnittsnote?

Die einfachste Möglichkeit den Median einer Menge von Werten zu finden, besteht darin, die Werte aufsteigend (oder absteigend) zu sortieren und dann den *mittleren* Wert auszuwählen, oder, falls die Anzahl der Werte ein gerade Zahl ist, den Mittelwert der *beiden mittleren* Werte zu bilden.

Um den Median zu berechnen muss das Programm fundamental geändert werden. Den Median einer unbekannten Anzahl von Werten zu finden, erfordert, dass jeder Wert gespeichert wird, bis alle Werte eingelesen wurden.

### *3.2.1 Speichern von Datenmengen in einem `vector`*

Zur Medianberechnung müssen alle WP-Noten eingelesen, gespeichert und sortiert werden und dann der mittlere Wert (oder die beiden mittleren Werte) ausgewählt werden. Für diese Berechnung benötigen wir eine Möglichkeit

- einzelne, eingelesene Werte zu speichern, ohne vorab die Anzahl der Werte zu kennen
- die Werte nach dem Einlesen zu sortieren
- den (die) mittleren Wert(e) auszuwählen

Die Standardbibliothek stellt uns den Typ `vector` zur Verfügung, der alle diese Aufgaben einfach lösen kann. Ein `vector` beinhaltet eine Folge von Werten eines bestimmten Typs, wird bei Bedarf erweitert, um zusätzliche Werte aufzunehmen und ermöglicht den effizienten Zugriff auf einzelne Werte.

Beginnen wir damit unser Programm neu zu schreiben, in dem wir die WP-Noten in einem `vector` speichern, anstatt aus `summe` und `anzahl` die Durchschnittsnote zu berechnen. Das Originalprogramm sah wie folgt aus:

```
// Originalprogramm (Ausschnitt)
int    anzahl = 0;
double summe  = 0;

// Variable zum Einlesen der Noten
double x;

// WP-Noten einlesen
while (cin >> x)
{
      anzahl++;
      summe += x;
}
```

Diese Schleife merkte sich die Anzahl der eingelesenen WP-Noten und die Summe aller WP-Noten. Das Programm zum Einlesen der Noten in einen `vector` ist etwas einfacher:

```
// überarbeitete Version des Ausschnitts
double x;
vector <double> wp;

while (cin >> x)
      wp.push_back(x);
```

Die grundsätzliche Struktur des Codes wurde nicht geändert: nach wie vor werden einzelne Werte nach `x` eingelesen bis *end-of-file* erreicht ist. Der Unterschied besteht in der Behandlung der einzelnen Werte.

Beginnen wir mit `wp`, einer Variable die vom Typ `vector<double>` ist. Ein `vector` ist ein ***Container***, der eine Menge von Werten enthält. Alle Werte eines bestimmten `vector`s sind vom gleichen Typ, aber verschiedene `vector`en können unterschiedliche Typen beinhalten. Bei jeder Definition eines `vector`s müssen wir den Typ der Werte angeben, die in dem `vector` gespeichert werden sollen. Unsere Definition bedeutet, dass `wp` ein `vector` ist, dessen Inhalt aus Werten vom Typ `double` besteht.

Der Typ `vector` verwendet ein Sprachelement mit dem Namen ***Klassentemplate***. Die Definition von ***Klassentemplates*** wird hier nicht behandelt. Hier ist wichtig, dass wir unterscheiden zwischen den Eigenschaften eines `vector` und dem speziellen Typ der Objekte, die ein `vector` speichert. Der Typ der Objekte wird innerhalb von `< >` angegeben. Beispielsweise sind Objekte des Typs `vector<double>` `vector`en, die Objekte des Typs `double` beinhalten, Objekte des Typs `vector<string>` beinhalten `string`s etc.

Die `while` – Schleife liest Werte von der Standardeingabe und speichert sie in einem `vector`. Wie zuvor lesen wir bis *end-of-file* erreicht ist, oder ein ungültiger Wert eingegeben wird. Neu ist die Anweisung

```
wp.push_back(x);
```

Genau wie `gruss.size()` aus Kapitel 1.2 ist `push_back` eine *Elementfunktion*, die als Teil des Typs `vector` definiert ist und wir führen die Funktion als Teil des Objektes `wp` aus. Die Funktion wird mit der Variablen `x` als Argument aufgerufen. `push_back` erweitert nun den `vector`, indem ein neues Element an das Ende des `vector`s angefügt wird. Diese neue Element bekommt als Inhalt den Wert der Variablen `x`, die als Argument übergeben wurde. `push_back` *pushed* also sein Argument an das Ende (*back*) des `vector`s `wp`. Als Seiteneffekt wird die Größe (*size*) des `vector`s `wp` um eins erhöht.

Als nächstes müssen wir uns mit der Programmausgabe beschäftigen.

### *3.2.2 Die Programmausgabe*

In der Originalversion unseres Programms aus Kapitel 3.1 wurde die Gesamtnote innerhalb der Ausgabeanweisung erzeugt:

```
streamsize prec = cout.precision();

cout << "Deine Gesamtnote ist: " << setprecision(2)
       << 0.2 * vordiplom + 0.4 * diplom + 0.4 * summe / anzahl
       << setprecision(prec) << endl;
```

wobei `vordiplom` und `diplom` die Prüfungsnoten und `summe` und `anzahl` die Gesamtsumme und Anzahl aller WP-Noten beinhalteten.

Wie bereits in Kapitel 3.2.1 beschrieben, besteht die einfachste Möglichkeit zur Berechnung des Median darin, die Werte zu sortieren und anschließend den mittleren Wert oder den Mittelwert der beiden mittleren Werte zu bestimmen. Wir werden die Berechnung einfacher gestalten, indem wir die Berechnung von der Ausgabe trennen.

Zur Berechnung des Medians benötigen wir die Größe (d.h. die Anzahl der in `wp` gespeicherten Objekte) des `wp vectors` mindestens zweimal: einmal um zu überprüfen, ob die Größe null ist, und ein weiteres mal um das mittlere Element zu bestimmen. Damit wir die Größe nur einmal bestimmen müssen, speichern wir sie in einer lokalen Variablen:

```
typedef vector<double>::size_type vec_sz;
vec_sz size = wp.size();
```

Der Typ `vector` definiert einen Typ mit dem Namen `vector<double>::size_type` und eine Funktion `size()`. Diese Elemente funktionieren analog zu denen von `string`: Der durch `size_type` definierte Typ, ist ein `unsigned` Typ, der groß genug ist, die Größe des längsten möglichen `vectors` aufzunehmen und `size()` ist die Elementfunktion, die die Anzahl der Elemente des vectors als Typ `size_type` zurückgibt.

Da wir die Größe an zwei Stelle benötigen, merken wir uns den Wert in einer lokalen Variablen. Verschiedene C++-Implementierungen verwenden verschiedene Typen, um Größenangaben zu definieren, daher können wir den entsprechenden Typ nicht direkt angeben und trotzdem implementierungsunabhängig bleiben. Aus diesem Grund ist es eine gute Angewohnheit `size_type` zu verwenden, das in der C++ - Bibliothek definiert wird und nicht etwa `int` oder `long`.

In diesem Beispiel ist der Typ `vector<double>::size_type` umständlich zu tippen und zu lesen. Zur Vereinfachung des Programms verwenden wir deshalb ein neues Sprachelement, genannt **`typedef`**. Benutzen wir das Wort `typedef` als Teil einer Definition, so sagen wir, dass wir den Namen der definiert wird, künftig als ***Synonym*** (oder eine Abkürzung) für den Typ selbst verwenden möchten (und nicht als Variable des angegeben Typs!). Da unsere Definition `typedef` enthält, definiert sie `vec_sz` als Synonym für `vector<double>::size_type`. Namen, die mit `typedef` definiert werden, haben den gleichen Geltungsbereich, wie alle anderen Namen auch. Wir können `vec_sz` also als Synonym für `vector<double>::size_type` bis zum Ende des aktuellen Blocks verwenden.

Da wir jetzt wissen, wie der Typ des Wertes lautet, den `wp.size()` zurückgibt, können wir diesen Wert in einer lokalen Variablen mit dem Namen `size` abspeichern.

Da es sinnlos ist, den Median eines leeren Datensatzes zu berechnen, überprüfen wir als nächstes, ob Daten vorhanden sind:

```
if (size == 0)
{
    cout << endl << "Du Schlafmütze, ohne WP kein Diplom!" << endl;
    return 1;
}
```

Sofern keine Daten vorhanden sind (`size == 0`), wird eine Fehlermeldung erzeugt und das Programm beendet. Rückgabewert ist 1, um dem Betriebssystem anzuzeigen, dass das Programm fehlerhaft beendet wurde.

Nachdem verifiziert wurde, dass Daten vorhanden sind, kann jetzt der Median berechnet werden. Der erste Schritt besteht darin, die Daten zu sortieren. Wir tun dies, durch den Aufruf einer Bibliotheksfunktion:

```
sort(wp.begin(), wp.end());
```

Die Funktion `sort`, definiert im Header `<algorithm>`, ordnet die Elemente in einem Container (z.B. einem `vector`) in aufsteigender Reihenfolge an.

Die Argumente der Funktion `sort` spezifizieren die Elemente, die sortiert werden sollen. Zu diesem Zweck hat die Klasse `vector` Elementfunktionen mit den Namen `begin` und `end`, so dass `wp.begin()` das erste Element des `vectors` angibt und `wp.end()` das letzte Element (genauer: „eins hinter dem letzten Element“) des `vectors` angibt.

Die Funktion `sort` verrichtet ihre Arbeit im Originalvector: Die Elemente werden im Originalcontainer umgeordnet und ***nicht*** in einen neuen Container kopiert.

Nachdem wp sortiert ist, können wir nun das (die) mittlere(n) Element(e) finden:

```
vec_sz mitte = size / 2;
double median;
median = size % 2 == 0 ? (wp[mitte] + wp[mitte - 1]) / 2
                       : wp[mitte];
```

Im ersten Schritt dividieren wir size durch zwei, um die Mitte des vectors zu bestimmen. Ist die Anzahl der Elemente eine gerade Zahl, ist das Ergebnis richtig, ist die Anzahl der Elemente eine ungerade Zahl, dann ist das Ergebnis die nächst kleinere Zahl.

Zur Erinnerung: bei gerader Anzahl ist der Median der Mittelwert der beiden mittleren Elemente, bei ungerader Anzahl ist der Median das mittlere Element

Der Ausdruck, der den Median berechnet, verwendet zwei neue Operatoren: den Operator %, und den Operator ?:, auch *bedingte Bewertung* oder *bedingter Ausdruck* genannt.

Der Operator % liefert den Divisionsrest nach einer ganzzahligen Division. Ist der Rest nach einer Division durch 2 gleich 0, dann hat das Programm eine gerade Anzahl von Elementen im wp vector.

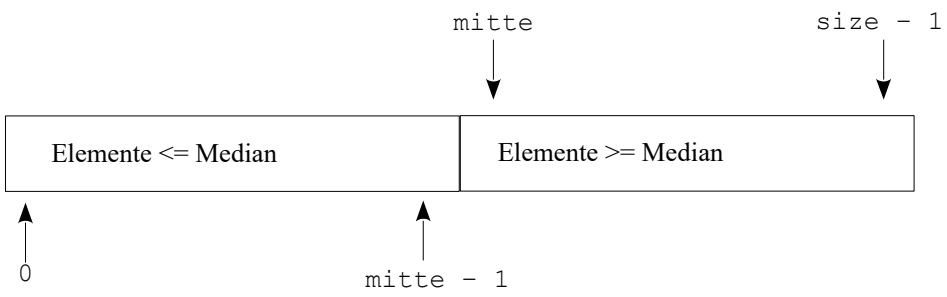
Die *bedingte Bewertung* ist eine Abkürzung für eine einfache if-then-else Anweisung. Zuerst wird der Ausdruck size % 2 == 0 bewertet, der als Ergebnis einen Wert vom Typ bool liefert. Falls diese Bedingung true ergibt, dann erhält man als Ergebnis der bedingten Bewertung den Wert des Ausdrucks zwischen ? und :, andernfalls erhält man als Ergebnis den Wert des Ausdrucks nach dem :. Sofern wir also ein gerade Anzahl von Elementen haben, weisen wir median den Mittelwert der beiden mittleren Elemente zu, haben wir eine ungerade Anzahl von Elementen weisen wir median den Wert wp[mitte], also den Wert des mittleren Elementes zu. Analog zu && und || bewertet der Operator ?: zuerst seinen linken Operanden. Abhängig von diesem Ergebnis, wird dann ***genau einer*** der beiden anderen Operanden ausgewertet.

wp[mitte] und wp[mitte - 1] zeigen ein Möglichkeit, auf die Elemente eines vectors zuzugreifen.

Mit jedem Element eines vectors ist ein Integerwert, genannt sein ***Index***, verknüpft. So ist zum Beispiel wp[mitte] das Element von wp, das mit dem Index mitte verknüpft ist. Wie Sie bestimmt schon vermuten erhält man mit wp[0] das erste Element von wp und mit wp[size - 1] das letzte Element von wp.

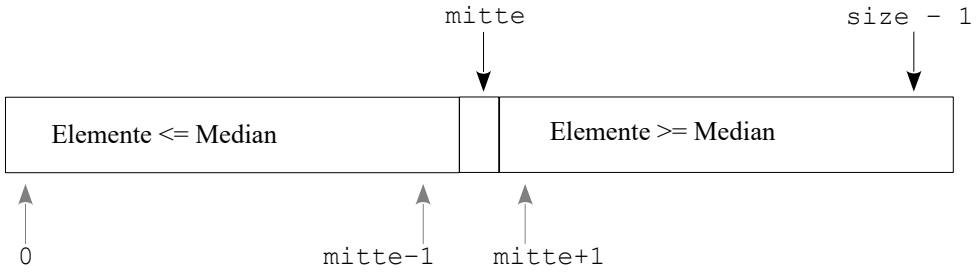
Jedes vector Element selbst ist ein Objekt (*ohne* Namen) des in einem Container gespeicherten Typs. wp[mitte] ist also ein Objekt vom Typ double, auf das wir alle für double definierten Operationen ausführen können. Insbesondere können wir zwei Elemente addieren und das Ergebnis durch 2 dividieren, um den Mittelwert zu erhalten.

Wir wissen jetzt wie der Elementzugriff funktioniert und können nun die Berechnung des Median im Detail betrachten. Nehmen wir an, dass size *gerade* ist, dann ist mitte genau size / 2. Es gibt somit genau mitte Elemente von wp auf jeder Seite der Mitte:



Wir wissen, dass jede Hälfte von `wp` genau `mitte` Elemente besitzt, somit ist leicht zu erkennen, dass die Indizes der mittleren Elemente `mitte - 1` und `mitte` sind; der Median ist der Mittelwert dieser Elemente.

Falls die Anzahl der Elemente *ungerade* ist, dann ist `mitte` tatsächlich `(size - 1) / 2`, da bei der Ganzzahldivision der Divisionsrest abgeschnitten wird. In diesem Fall können uns vorstellen, dass der sortierte `wp vector` aus zwei Segmenten mit `mitte` Elementen besteht, die durch ein einzelnes Element in der Mitte getrennt werden. Dieses Element ist der Median:




In beiden Fällen hängt die Berechnung davon ab, auf ein Element eines `vectors` über einen Index zugreifen zu können.

Nach der Berechnung des Median können wir die Gesamtnote berechnen und ausgeben:

```
streamsize prec = cout.precision();

cout << "Deine Gesamtnote ist: " << setprecision(2)
       << 0.2 * vordiplom + 0.4 * diplom + 0.4 * median
       << setprecision(prec) << endl;
```

Das komplette Programm ist nicht sehr viel komplizierter als das aus Kapitel 3.1, obwohl es sehr viel mehr Arbeit verrichtet. Insbesondere muss sich das Programm nicht damit beschäftigen Speicherplatz für den dynamisch wachsenden Inhalt des `wp vectors` bereitzustellen. Die Standardbibliothek erledigt diese Aufgabe für uns.

Hier ist nun das komplette Programm. Die einzigen nicht besprochenen Teile sind einige `#include` Anweisungen, einige `using` Deklarationen und einige Kommentare:

```
#include <algorithm>
#include <iomanip>
#include <ios>
#include <iostream>
#include <string>
#include <vector>


using std::cin;              using std::sort;
using std::cout;             using std::string;
using std::endl;             using std::streamsize;
using std::setprecision;     using std::vector;

int main()
{
      // den Namen des Studenten erfragen
      cout << "Bitte gib Deinen Vorname ein: ";

      string name;
      cin >> name;

      cout << "Tach auch, " << name << "!?!" << endl;
```

```
        // Noten aus Vordiplom- und Diplomprüfung erfragen
        cout << "Bitte gib deine Vordiplom- und Diplomnote ein: ";


        double  vordiplom,
                diplom;

        cin >> vordiplom >> diplom;

        // Noten der WP-Fächer erfragen
        cout << "Bitte die Noten aller WP-Fächer eingeben "
                    "und mit end-of-file beenden: ";

        // Variable zum Einlesen der Noten
        double x;
        vector<double> wp;

        // Noten einlesen
        while (cin >> x)
        {
              wp.push_back(x);
        }

        typedef vector<double>::size_type vec_sz;
        vec_sz size = wp.size();

        if (size == 0)
        {
              cout << "Du Schlafmütze, ohne WP kein Diplom!" << endl;
              return 1;
        }

        sort(wp.begin(), wp.end());

        vec_sz mitte = size / 2;
        double median;
        median = size % 2 == 0 ? (wp[mitte] + wp[mitte - 1]) / 2
                               : wp[mitte];

        streamsize prec = cout.precision();

        cout << "Deine Gesamtnote ist: " << setprecision(2)
               << 0.2 * vordiplom + 0.4 * diplom + 0.4 * median
               << setprecision(prec) << endl;

        return 0;
    }
```

### *3.3 Details*

***Lokale Variable*** werden standardmäßig initialisiert, sofern keine explizite Initialisierung angegeben ist. Die Standardinitialisierung von eingebauten Datentypen (z.B. `int, double` etc.) ist nicht definiert, d.h. der Wert der entsprechenden Variablen ist ebenfalls ***undefiniert***.

***Typdefinitionen:***

`typedef` *type name;* Definiert *name* als Synonym für *type*.

***Der Typ `vector`***, definiert in `<vector>`, ist ein in der Standardbibliothek definierter Typ, der einen Container darstellt, der eine Folge von Werten des angegeben Typs beinhaltet. `Vector`en wachsen dynamisch. Einige wichtige Operationen sind:

| | |
|---|---|
| `vector<T>::size_type` | Ein Ganzzahltyp der garantiert groß genug ist, die Anzahl der Elemente des größtmöglichen vectors aufzunehmen. |
| `v.begin()` | Gibt einen Wert zurück, der auf das erste Element von `v` verweist. |
| `v.end()` | Gibt einen Wert zurück, der auf „eins hinter das das letzte Element“ von `v` verweist. |
| `vector<T> v` | Erzeugt einen `vector`, der Elemente des Typs `T` aufnehmen kann. |
| `v.push_back(e)` | Erweitert `v` um ein mit `e` initialisiertes Element. |
| `v[i]` | Liefert den an der Position `i` abgelegten Wert. |
| `v.size()` | Liefert die Anzahl der Elemente von `v`. |

***Weitere Bibliotheksoperationen***

| | |
|---|---|
| `sort(b, e)` | Ordnet die Elemente des durch `[b, e)` definierten Bereichs in aufsteigender Reihenfolge. Ist in `<algorithm>` definiert. |
| `while (cin >> x)` | Liest einen Wert des passenden Typs nach `x` und überprüft den Status des Eingabestroms. Ist der Status fehlerhaft, so ist das Ergebnis `false`, andernfalls ist das Ergebnis des Tests `true` und der Rumpf der `while` – Anweisung wird ausgeführt. |
| `s.precision(n)` | Legt die Genauigkeit der Ausgabe des Stroms `s` für künftige Ausgaben auf `n` signifikante Stellen fest (oder ändert die Genauigkeit nicht, falls `n` nicht angegeben wird). Gibt die vorherige Genauigkeit zurück. |
| `setprecision(n)` | Gibt einen Wert zurück, der, sofern er auf einen Ausgabestrom geschrieben wird, die gleiche Bedeutung hat wie `s.precision(n)`. Definiert in `<iomanip>`. |
| `streamsize` | Der Typ des Wertes, der von `setprecision` erwartet wird und von `precision` zurückgegeben wird. Definiert in `<ios>`. |

## *Aufgaben*

3.1 Schreiben Sie ein Programm, das nach Eingabe der Vordiplomnote, der Diplomnote und der unbekannten Anzahl an Noten der WP-Fächer die Gesamtnote nach folgender Gewichtung ermittelt: 20% Vordiplom, 40% Diplom, 40% Summe aller WP-Noten. Tipp: Die Bedingung `while(cin >> x)` liefert `false`, sofern `ctrl-d` eingegeben wird.

3.2 In dem Typ `vector<T>` können Sie eine Anzahl von Elementen des Typs `T` abspeichern. `vector` ist in `<vector>` definiert. Beispiel: `vector<int> iVector;` `iVector` ist eine Variable, die eine beliebige Anzahl von `int` Werten aufnehmen kann. Mit `iVector.push_back(i)` können Sie das Element i zu dem Vektor hinzufügen, mit `iVector[k]` greifen Sie auf das k-te Element des Vektors zu. Schreiben Sie ein Programm, das die Zahlen von 1 bis 42 in einer Variablen des Typs `vector` abspeichert und geben sie zur Kontrolle alle Werte aus.

3.3 Mit sort(Anfang, Ende) können Sie die Elemente eines vectors sortieren. Anfang und Ende definieren den Bereich des vectors, der sortiert werden soll. Auf das erste Element des vectors iVector können Sie mit iVector.begin() zugreifen, auf das letzte Element mit iVector.end(). Die Funktion `iVector.size()` liefert die Größe (d.h. Anzahl Elemente) des vectors iVector. sort() ist in <algorithm> definiert.

Ändern Sie das Programm aus Aufgabe 3.1 so, dass es statt des Durchschnitts der WP-Noten, den *Median* verwendet. Der *Median* ist der *mittlere* Wert eine Menge von Zahlen. (Tipp: Zuerst den Ablauf überlegen, also Sortieren, Größe ermitteln, *Median* finden etc.).

3.4 Das Programm aus Aufgabe 3.1 würde durch 0 dividieren müssen, sofern Sie keine einzige WP-Note eingeben. Division durch 0 ist in C++ nicht definiert, d.h. Der Compiler darf darauf reagieren, wie es ihm gefällt. Überprüfen Sie wie sich Visual Studio verhält und ändern Sie das Programm so, dass eine Division durch 0 erkannt wird.

# 4 Die Organisation von Programmen und Daten

In den bisher entwickelten Programmen nutzen Sie Elemente aus der Standardbibliothek wie z.B. `vector` `string` und `sort`. Diese Elemente haben gemeinsame Eigenschaften: Sie

- lösen ein konkretes Problem
- sind (weitgehend) unabhängig von anderen Elementen
- verfügen über einen Namen

Unsere eigenen Programme verfügen nur über die erste Eigenschaft, die anderen fehlen. Bei kleinen Programmen ist dies bedeutungslos, aber mit zunehmender Programmgröße wird es immer wichtiger, dass Sie ihre Programme in unabhängige Elemente mit eigenem Namen aufteilen können.

In C++ gibt es grundsätzlich zwei Möglichkeiten große Programme zu organisieren: ***Funktionen*** und ***Datenstrukturen***. Zusätzlich können Funktionen und Datenstrukturen zu ***Klassen*** kombiniert werden, wie Sie ab Kapitel 9 sehen werden.

## *4.1 Funktionen*

Wir beginnen mit einer Funktion, die die Gesamtnote aus Vordiplomnote, Diplomnote und dem Mittelwert aller WP-Noten berechnet. Wir setzen voraus, dass der Mittelwert der WP-Noten bereits berechnet wurde.

Immer wenn eine Berechnung mehrfach benötigt wird (oder benötigt werden könnte!), sollten Sie daran denken eine Funktion zu schreiben, die die Berechnung durchführt. Die Verwendung von Funktionen verkleinert damit ihr Programm und erleichtert Programmänderungen. Ein weiterer Vorteil ist die Verwendung von *Namen* für Funktionen. Sie können dadurch Berechnungen abstrakter behandeln, also mehr darüber nachdenken *was* Berechnungen tun und nicht *wie* sie es tun. Wenn Sie wichtige Teile Ihrer Problemstellung erkennen und ihnen Namen geben können, so werden diese Probleme einfacher zu lösen und Ihre Programme einfacher zu verstehen sein.

Hier ist die Funktion, die die Gesamtnote berechnet:

```
double gesamt(double vd, double diplom, double wp)
{
    return 0.2 * vd + 0.4 * diplom + 0.4 * wp;
}
```

Alle Funktionen, die sie bisher verwendet hatten, hatten den Namen `main`. Die meisten anderen Funktionen werden ähnlich definiert, indem man den *Rückgabetyp* angibt, gefolgt von dem *Funktionsnamen*, einer von ( ) umgebenen *Parameterliste* und schließlich dem *Funktionsrumpf*, der von { } umgeben wird.

In diesem Beispiel sind die Parameter `vd`, `diplom` und `wp`, jeweils vom Typ `double`. Sie verhalten sich, wie innerhalb der Funktion definierte ***lokale Variable***, d.h. sie werden beim Funktionsaufruf erzeugt und beim Verlassen der Funktion zerstört. Wie andere Variable auch, müssen Parameter vor ihrer Verwendung definiert werden. Dabei werden sie jedoch nicht unmittelbar erzeugt. Dies geschieht erst beim Funktionsaufruf, bei dem korrespondierende ***Argumente*** übergeben werden müssen, die zur Initialisierung der Parameter verwendet werden.

Sie könnten beispielsweise die Funktion `gesamt` folgendermaßen verwenden:

```
cout << "Gesamtnote: " << gesamt(vordiplom, diplom, summe / anzahl) << endl;
```

Wir müssen nicht nur Argumente zur Verfügung stellen, die mit den Parametern der aufgerufenen Funktion übereinstimmen, wir müssen sie auch in genau der richtigen Reihenfolge übergeben. Beim Aufruf der Funktion `gesamt` müssen dementsprechend das erste Argument die Vordiplomnote, das zweite Argument die Diplomnote und das dritte Argument die WP-Note sein.

Anstelle von Variablen können auch Ausdrücke wie z.B. `summe / anzahl` als Argument verwendet werden. Grundsätzlich wird jedes Argument dazu verwendet den entsprechenden Parameter zu initialisieren, der sich anschließend wir eine normale lokale Variable innerhalb der Funktion verhält. Wenn Sie beispielsweise `gesamt(vordiplom, diplom, summe / anzahl)` aufrufen, werden die Parameter der Funktion `gesamt` mit *Kopien* der Argumentwerte initialisiert und beziehen sich nicht auf die Argumente selbst. Dieses Verhalten wird ***call by value*** genannt, da die Parameter eine Kopie der Werte (*values*) der Argumente erhalten.

### *4.1.1 Berechnung des Medians*

Die Berechnung des Medians eines Vektors mit Elementen des Typs `<double>` ist eine weitere Berechnung, die sinnvoll innerhalb einer Funktion ausgeführt wird. Um die Funktion zu schreiben, verwenden wir den Teil des Programms aus Kapitel 3, der den Median berechnet und führen einige Änderungen durch:

```
double median(vector<double> vec)
{
    typedef vector<double>::size_type vec_sz;

    vec_sz size = vec.size();
    if (size == 0)
    {
        throw domain_error("Fehler: Median eines leeren Vektors");
    }

    sort(vec.begin(), vec.end);

    vec_sz mitte = size / 2;

    return size % 2 == 0 ? (vec[mid] + vec[mid - 1]) / 2 : vec[mid];
}
```

Die Variable `median` wird nicht mehr benötigt: der Median wird als Rückgabewert verwendet, sobald er berechnet wurde. Die Variablen `size` und `mitte` werden weiterhin verwendet, jetzt allerdings als *lokale Variable* innerhalb der Funktion `median`. Sie können damit nur innerhalb der Funktion verwendet werden. Der Aufruf der Funktion `median` erzeugt diese Variablen, beim Verlassen der Funktion werden sie zerstört. `vec_sz` wird als *lokaler* Typname innerhalb der Funktion definiert und kann daher nicht mit einer gleichnamigen Definition an einer anderen Stelle im Programm in Konflikt geraten.

Die auffälligste Änderung betrifft die Behandlung eines *leeren* Vektors. Die Eingabe der Vektorelemente erfolgt außerhalb der Funktion, d.h. die Funktion kann sich nicht darauf verlassen, dass der Vektor tatsächlich mit WP-Noten gefüllt wurde. Die allgemeine Art der Fehlerbehandlung erfolgt in C++ durch ***werfen (throw)*** einer ***Ausnahme (exception),*** falls der `vector` leer ist.

Wirft ein Programm eine Ausnahme, so endet die Programmausführung des Programmteils, der die Ausnahme wirft und das Programm wird an einer anderen Stelle fortgesetzt und erhält dort als Fehlerinformation ein ***Ausnahmeobjekt (exception object)***, das Informationen über die Fehlerursache enthält.

In unserem Beispiel wird eine `domain_error` Ausnahme geworfen, ein Typ, der in der Standardbibliothek in `<stdexcept>` definiert ist und der anzeigt, dass ein Funktionsargument außerhalb des gültigen Wertebereichs verwendet wurde. Beim Erzeugen des `domain_error` Objektes wird ein `string` übergeben, das die Fehlerursache beschreibt. Wie dieses `string` ausgewertet werden kann, erfahren Sie in Kapitel 4.2.

Es gibt noch ein wichtiges Detail, dass Sie beachten sollten. Wenn eine Funktion aufgerufen wird, können Sie die Parameter als lokale Variable mit den Argumenten als Anfangswerten betrachten. Also werden beim Funktionsaufruf, die Parameter in die Argumente kopiert. Insbesondere wird beim Aufruf der Funktion `median` der als Argument verwendete Vektor nach `vec` kopiert.

Dieses Verhalten ist sinnvoll, da die `median` Funktion den Inhalt des Parameters `vec` mit `sort` verändert. Die Berechnung des Median soll aber nicht den Originalvektor verändern.

### *4.1.2 Neuimplementierung der Strategie der Gesamtnotenberechnung*

Die Funktion `gesamt` aus Kapitel 4.1 erwartete, dass die Gesamtnote der WP-Fächer bekannt ist und nicht nur die einzelnen WP-Noten. Wie wir die Gesamtnote erhalten ist Teil unserer Strategie: in Kapitel 3.1 verwendeten wir den Notendurchschnitt, in Kapitel 3.2.2 den Median. Diesen Teil der Berechnungsstrategie können wir auch durch eine Funktion analog zur Funktion `gesamt` aus Kapitel 4.1 ausdrücken:

```
// Gesamtnote aus Vordiplomnote, Diplomnote und WP-Noten berechnen
// Diese Funktion kopiert das Argument wp nicht, da dies bereits die
// Funktion median macht

double gesamt(double vd, double diplom, const vector<double>& wp)
{
      if (wp.size() == 0)
                  throw domain_error("Fehler: Keine WP-Noten!");

      return gesamt(vd, diplom, median(wp));
}
```

Diese Funktion hat drei besonders interessante Stellen.

Die erste Stelle ist der Typ `const vector<double>& wp`, der für das dritte Argument angegeben wird. Dieser Typ wird „Referenz auf einen `const vector` von `double` Werten" genannt. Die Definition eines Namens als *Referenz* auf ein Objekt bedeutet, dass der Name ein Synonym (ein anderer Name) für das Objekt ist. Schreiben wir beispielsweise

```
vector<double> wpnoten;
vector<double>& wp = wpnoten;
```

so sagen wir, dass `wp` ein anderer Name für `wpnoten` ist. Von diesem Moment an sind alle Operationen, die wir mit `wp` durchführen, äquivalent dazu, die gleichen Operationen mit `wpnoten` durchzuführen (dies gilt natürlich auch umgekehrt!). Fügt man `const` hinzu, wie in

```
// cwp ist ein read-only Synonym für wpnoten
const vector<double>& cwp = wpnoten;
```

bedeutet das nach wie vor, dass `cwp` ein anderer Name für `wpnoten` ist, aber diesmal versprechen wir den Inhalt von `cwp` nicht zu ändern.

Da eine Referenz ein anderer Name für das Original ist, gibt es keine Referenz auf eine Referenz. Die Definition einer Referenz auf eine Referenz ist nichts anderes als eine Referenz auf das Original. Schreiben wir beispielsweise

```
// wp1 und cwp1 sind Synonyme für wp; cwp1 ist read-only
vector<double>& wp1 = wp;
const vector<double>& cwp1 = cwp;
```

dann ist `wp1` ein anderer Name für `wpnoten`, genau wie `wp` und `cwp1`, genau wie `cwp`, ist ein anderer Name für `wpnoten`, der Schreibzugriff verbietet.

Definieren wir eine nicht`const` Referenz – eine Referenz, die Schreibzugriff erlaubt – so kann sie sich nicht auf ein `const` Objekt beziehen, denn dies würde eine Schreibberechtigung erfordern, die `const` verbietet. Deshalb können wir nicht schreiben

```
vector<double>& wp2 = cwp;    // Fehler: benötigt Schreibzugriff auf cwp
```

da `cwp` read-only definiert ist.

Ist ein Funktionsparameter mit dem Typ `const vector<double>&` definiert, so erhalten wir ganz analog direkten Zugriff auf das zugeordnete Argument, ohne dass es kopiert wird, und wir versprechen gleichzeitig den Wert des Parameters nicht zu ändern (wobei der Wert des Argumentes auch geändert würde). Da der Parameter eine Referenz auf `const` ist, können wir die Funktion sowohl mit `const vector`en, als auch mit nicht`const vector`en aufrufen. Da der Parameter eine Referenz ist, vermeiden wir den zusätzlichen Aufwand das Argument zu kopieren.

Die zweite Stelle von besonderem Interesse in der Funktion `gesamt`, ist der Name der Funktion selbst. Die Funktion heißt genau wie die Funktion aus Kapitel 4.1 `gesamt`, obwohl eine andere Funktion `gesamt` aufgerufen wird. Die Notation, dass mehrere Funktionen den gleichen Namen haben können wird ***überladen (overloading)*** genannt. Obwohl zwei Funktionen den gleichen Namen haben, gibt es *keine* Mehrdeutigkeit, da die Funktionen beim Aufruf anhand der Argumentenliste unterschieden werden können. In unserem Programm unterschieden sich die beiden Funktionen mit dem Namen `gesamt` durch den Typ des dritten Argumentes.

Die dritte bedeutsame Stelle ist der Test, ob `wp.size()` gleich null ist, obwohl wir ja wissen, dass `median` dies für uns erledigt. Da Grund dafür ist, dass `median`, falls es diesen Fehler bemerkt, eine *Ausnahme (exception)* mit der Meldung „Fehler: Median eines leeren Vektors!", wirft. Diese Meldung ist nicht sehr hilfreich für die Person, die die Gesamtnote berechnen möchte. Daher werfen wir eine eigene Ausnahme, von der wir annehmen, dass sie dem Anwender einen besseren Hinweis auf die Fehlerursache gibt.

### *4.1.3 Einlesen der WP-Noten*

Ein Problem das wir schon mehrfach zu lösen hatten, ist das Einlesen einer unbekannten Anzahl von WP-Noten in einen `vector`.

Es gibt ein grundsätzliches Problem, wenn wir eine solche Funktion konstruieren: Sie muss gleichzeitig zwei Werte zurückgeben. Ein Wert ist natürlich die gelesene WP-Note, der andere Wert ein Hinweis, ob der Lesevorgang erfolgreich war, oder nicht.

Es gibt keine direkte Möglichkeit mehr als einen Wert von einem Funktionsaufruf zurückzugeben. Eine indirekte Möglichkeit besteht darin, einen Funktionsparameter als Referenz auf ein Objekt zu definieren, in den dann eines der Ergebnisse geschrieben wird. Dies ist eine übliche Vorgehensweise für Funktionen, die Eingabe bearbeiten, deshalb werden wir sie verwenden. Unsere Funktion sieht mit dieser Idee wie folgt aus:

```
// WP-Noten von einem Eingabestrom in einen vector<double> einlesen
istream& read_wp(istream& in, vector<double>& wp)
{
    // dies ist noch zu programmieren
    return in;
}
```

In Kapitel 4.1.2 hatten wir eine Parameterdefinition mit dem Typ `const vector<double>&`, diesmal haben wir die gleiche Definition ohne `const`. Ein Referenzparameter ohne `const` signalisiert normalerweise die Absicht, den Inhalt des Parameters verändern zu wollen. Betrachten wir folgenden Code

```
vector<double> wp;
read_wp(cin, wp);
```

Die Tatsache, dass der zweite Parameter eine Referenz ist, sollte uns erwarten lassen, dass der Aufruf von `read_wp` den Inhalt von `wp` ändert.

Da wir erwarten, dass die Funktion ihre Argumente ändert, ist es nicht möglich die Funktion mit jedem beliebigen Ausdruck aufzurufen. Stattdessen müssen wir ein ***lvalue (l-Wert)*** Argument an einen Referenzparameter übergeben. Ein *lvalue* ist ein Wert (*value*), der auf ein nicht-temporäres Objekt verweist. Beispielsweise ist eine Variable ein *lvalue*, genau wie eine Referenz oder wie eine Funktion, die eine Referenz zurückgibt. Ein Ausdruck,der ein arithmetisches Ergebnis hat, wie z.B. `summe/anzahl` ist kein *lvalue*.

Beide Parameter von `read_wp` sind Referenzen, da wir erwarten, dass beide von der Funktion geändert werden. Wir kennen noch nicht die Details der Arbeitsweise von `cin`, aber sicherlich ist es innerhalb der Standardbibliothek als Datenstruktur definiert, die alles für die Bibliothek notwendige über den Status des Eingabestroms beinhaltet. Einlesen von der Standardeingabe ändert den Status der Standardeingabe und somit logischerweise auch den Wert von `cin`.

Beachten Sie, dass `in`, der Rückgabewert von `read_wp`, eine Referenz auf `istream` ist (`istream` ist der Typ der Standardeingabe `cin`). Tatsächlich übergeben wir ein Objekt (`cin`) an die Funktion, das wir nicht kopieren und geben das gleiche Objekt als Funktionsergebnis zurück, wiederum ohne es zu kopieren. Ein Stream als Rückgabewert ermöglicht uns somit folgenden Code

```
if (read_wp(cin, wp)
{
      // tue irgendetwas sinnvolles ...
}
```

als Abkürzung für

```
read_wp(cin, wp);
if (cin)
{
      // tue irgendetwas sinnvolles
}
```

zu schreiben.

Jetzt können wir uns damit beschäftigen, die WP-Noten einzulesen. Offensichtlich müssen wir genau so viele Noten einlesen, wie vorhanden sind, wie können also scheinbar schreiben

```
// erster Versuch, nicht ganz richtig !
double x;
while (in >> x)
    wp.push_back(x);
```

Diese Strategie scheitert aus zwei Gründen.

Der erste Grund ist, dass nicht wir selbst `wp` definiert haben, sondern der Aufrufer der Funktion. Da wir es nicht selbst definiert haben, wissen wir nicht, ob `wp` bereits Daten enthält. Wir können uns aber vorstellen, dass der Aufrufer unserer Funktion die WP-Fächer einer Menge von Studenten bearbeitet. In diesem Fall beinhaltet `wp` die Noten des vorherigen Studenten. Wir lösen dieses Problem, indem wir `wp.clear()` am Anfang unserer Funktion aufrufen.

Der zweite Grund ist, dass wir nicht genau wissen, wann wir das Einlesen beenden sollen. Wir können Noten einlesen, bis es nicht mehr länger möglich ist, aber an diesem Punkt bekommen wir ein Problem. Es gibt nämlich zwei Gründe, warum die Eingabe nicht mehr möglich ist: wir haben end-of-file erreicht, oder es wurde etwas eingegeben, das nicht dem Typ einer Note (also dem Typ von `x`) entsprach.

Im ersten Fall vermutet der Aufrufer end-of-file erreicht zu haben. Dieser Gedanke ist richtig, aber irreführend, da die Anzeige von end-of-file erst erfolgt, nachdem alle Daten erfolgreich eingelesen wurden. Normalerweise zeigt end-of-file einen fehlgeschlagenen Einleseversuch an.

Im zweiten Fall, wenn wir etwas gelesen haben, dass nicht dem Typ einer WP-Note entspricht, wird die Standardbibliothek den Status des Eingabestroms auf *fehlerhaft* setzen. Dies bedeutet, dass all nachfolgenden Eingabeversuche scheitern würden, genau so, als hätten wir end-of-file erreicht. Also wird der Aufrufer vermuten, dass irgendetwas mit den Eingabedaten nicht in Ordnung ist, obwohl nur die nächste Eingabe nach der letzten WP-Note gelesen wurde.

Im beiden Fällen würden wir gerne so tun, als hätten wir die Eingabe ***nach*** der letzten WP-Note niemals gelesen. Dies ist einfach möglich: falls wir end-of-file erreicht hatten, gab es keine zusätzliche Eingabe, falls wir etwas anderes als eine Note erwischt hatten, wird es die Bibliothek für die nächste Leseoperation

ungelesen im Eingabestrom stehen lassen. Alles was wir tun müssen, ist also der Bibliothek mitzuteilen, dass sie den Fehlerstatus innerhalb des Eingabestroms `in` zurücksetzt. Dies erfolgt mit dem Aufruf von `in.clear()`, der ermöglicht, dass die Eingabe von `in` zukünftig wieder erfolgen kann.

Ein weiteres Detail ist noch zu berücksichtigen: vielleicht wurde bereits end-of-file oder eine andere Fehlerbedingung erreicht, bevor überhaupt das erste mal versucht wurde, eine WP-Note einzulesen. In diesem Fall können wir den Eingabestrom nicht verwenden.

Es folgt die komplette Funktion `read_wp`:

```
istream& read_wp(istream& in, vector <double>& wp)
{
    if (in)
    {
        // alte Inhalte löschen
        wp.clear();

        // WP-Noten einlesen
        double x;
        while (in >> x)
            wp.push_back(x);

            // Eingabestrom zurücksetzen, zukünftige Eingabe ermöglichen
            in.clear();
    }

    return in;
}
```

Beachten Sie, dass sich die Elementfunktion `clear()`, für `istream` Objekte ganz anders verhält als für `vector` Objekte. Für `istream` Objekte wird der Fehlerstatus zurückgesetzt, so dass die Eingabe fortgesetzt werden kann, bei `vector` Objekten wird der Inhalt des `vector`s gelöscht, so dass wir mit einem vollständig leeren `vector` arbeiten können.

### *4.1.4 Drei verschiedene Arten von Funktionsparametern*

Wir haben bisher drei Funktion definiert – `median`, `gesamt` und `read_wp` – die `vector`en der WP-Noten verwenden. Jede dieser Funktionen behandelt die entsprechenden Parameter auf grundsätzlich andere Art und Weise, wie die anderen Funktionen.

Die Funktion `median` hat einen Parameter vom Typ `vector<double>`. Deshalb hat ein Aufruf dieser Funktion zur Folge, dass das Argument kopiert wird, auch wenn dieses Argument ein sehr großer `vector` ist. Trotz dieser Ineffizienz ist `vector<double>` der richtige Typ für die Funktion `median`, da dieser Typ sicherstellt, dass der Inhalt des `vector`s bei der Medianberechnung nicht geändert wird. Die Funktion `median` sortiert ihre Parameter. Würde das Argument nicht kopiert werden, würde der Aufruf `median(wp)` den Wert von `wp` ändern.

Die Funktion `gesamt` hat einen Parameter vom Typ `const vector<double>&`. In diesem Fall bedeutet das Zeichen `&`, dass das Argument nicht kopiert wird und `const` sorgt dafür, dass die Funktion den Parameter nicht verändern wird. Mit Parametern diesen Typs arbeiten Programme sehr viel effizienter. Sie sind immer dann eine gute Idee, wenn die Werte eines Parameters nicht geändert werden sollen, und der Parameter von einem Typ ist, wie z.B. `string` oder `vector`, dessen Werte evtl. zeitaufwendig zu kopieren sind. Bei eingebauten Typen wie z.B. `int` oder `double`, lohnt es sich normalerweise nicht, sich Gedanken über die Verwendung von `const` Referenzen zu machen, denn solche kleinen Objekte sind üblicherweise sehr schnell kopiert, mit wenig oder gar keinem Zusatzaufwand bei der Übergabe *by value*.

Die Funktion `read_wp` hat einen Parameter vom Typ `vector<double>&`, ohne `const`. Wiederum bedeutet das Zeichen `&`, dass der Parameter direkt mit dem Argument verknüpft wird und so vermieden wird, den Parameter zu kopieren. Hier ist der Grund für das Vermeiden des Kopierens, dass *beabsichtigt* ist, den Wert des Argumentes zu ändern.

Argumente, die zu nicht`const` Referenzparametern korrespondieren, müssen *lvalues* sein, d.h. sie müssen nicht-temporäre Objekte sein. Argumente, die *by value* übergeben werden oder mit einer `const` Referenz verknüpft sind, können jeden beliebigen Typ haben. Nehmen wir beispielsweise an, wir haben eine Funktion, die einen leeren `vector` zurückgibt:

```
vector<double> emptyvec()
{
    vector<double> v;
    return v;
}
```

Wir könnten diese Funktion aufrufen und das Ergebnis als Argument der zweiten Funktion `gesamt` verwenden:

```
gesamt(vordiplom, diplom, emptyvec());
```

Wenn wir diesen Code ausführen, wird die Funktion `gesamt` augenblicklich eine Ausnahme werfen, da ihr drittes Argument leer ist. Dennoch wäre dieser Aufruf der Funktion `gesamt` syntaktisch einwandfrei, d.h. der Compiler würde keine Fehlermeldung erzeugen.

Rufen wir `read_wp` auf, so müssen beide Argumente *lvalues* sein, denn beide Parameter sind non`const` Referenzen. Übergeben wir an `read_wp` einen Vektor, der kein *lvalue* ist

```
read_wp(cin, emptyvec());
```

sollte der Compiler eine Meldung erzeugen, da der mit dem Aufruf von `emptyvec` erzeugte, namenlose `vector` verschwindet, sobald `read_wp` beendet wird. Wäre dieser Aufruf erlaubt, könnten wir Werte in einem Objekt speichern, auf das wir ***nicht*** zugreifen können.

### *4.1.5 Die Verwendung von Funktionen zur Berechnung der Gesamtnote*

Die in den vorherigen Kapiteln geschriebenen Funktionen verwenden wir jetzt um das Programm zur Berechnung der Gesamtnote aus Kapitel 3.2.2 neu zu implementieren:

```
#include <algorithm>
#include <iomanip>
#include <ios>
#include <iostream>
#include <stdexcept>
#include <string>
#include <vector>

using namespace std;

double median (vector<double> vec)
{
      typedef vector<double>::size_type vec_sz;

      vec_sz size = vec.size();

      if (size == 0)
            throw domain_error("Fehler: Median eines leeren Vektors!");

      sort(vec.begin(), vec.end());

      vec_sz mid = size / 2;
```

```
	return size % 2 == 0 ? (vec[mid] + vec[mid-1]) / 2 : vec[mid];
}

double gesamt(double vd, double diplom, double wp)
{
	return 0.2 * vd + 0.4 * diplom + 0.4 * wp;
}

double gesamt(double vd, double diplom, const vector<double>& wp)
{
	if (wp.size() == 0)
			throw domain_error("Fehler: Keine WP-Noten!");

	return gesamt(vd, diplom, median(wp));
}

istream& read_wp(istream& in, vector <double>& wp)
{
	if (in)
	{
		// alte Inhalte löschen
		wp.clear();

		// WP-Noten einlesen
		double x;
		while (in >> x)
			wp.push_back(x);

		// Eingabestrom zurücksetzen, zukünftige Eingabe ermöglichen
		in.clear();
	}

	return in;
}

int main()
{
	// den Namen des Studenten erfragen
	cout << "Bitte gib Deinen Vorname ein: ";

	string name;
	cin >> name;

	cout << "Tach auch, " << name << "!?!" << endl;

	// Noten aus Vordiplom- und Diplomprüfung erfragen
	cout << "Bitte gib deine Vordiplom- und Diplomnote ein: ";

	double  vordiplom,
			diplom;

	cin >> vordiplom >> diplom;

	// Noten der WP-Fächer erfragen
	cout << "Bitte die Noten aller WP-Fächer eingeben "
		"und mit end-of-file beenden: ";

	// Variable zum Einlesen der Noten
	vector<double> wp;
```

```
        read_wp(cin, wp);

        try
        {
              double GesamtNote = gesamt(vordiplom, diplom, wp);
              streamsize prec = cout.precision();

              cout << "Deine Gesamtnote ist: " << setprecision(2)
                   << GesamtNote << setprecision(prec) << endl;
        }
        catch (domain_error)
        {
              cout << endl << "WP Noten eingeben!" << endl;
              return 1;
        }
        return 0;
}
```

Die Änderungen gegenüber früheren Versionen bestehen in der Art und Weise, wie WP-Noten eingelesen werden und die Gesamtnote berechnet und ausgegeben wird.

Nach der Aufforderung die WP-Noten einzugeben, rufen wir die Funktion `read_wp` auf, um die Daten einzulesen. Die `while` Anweisung innerhalb von `read_wp` liest wiederholt die WP-Noten ein, bis end-of-file oder ein Wert, der nicht dem Typ `double` entspricht, eingegeben wird.

Die wichtigste neue Idee in diesem Beispiel ist die **`try`** Anweisung. Sie versucht alle Anweisungen innerhalb des { } - Blockes auszuführen, der auf das `try` Schlüsselwort folgt. Falls innerhalb dieser Anweisungen ein `domain_error` auftritt, wird die Bearbeitung der Anweisungen abgebrochen und die Anweisungen innerhalb des anderen { } - Blockes werden ausgeführt. Diese Anweisungen sind Bestandteil einer **`catch`** Anweisung, die mit dem Wort `catch` beginnt und den Typ der Ausnahme anzeigt, die aufgefangen wird.

Können **alle** Anweisungen zwischen `try` und `catch`, ausgeführt werden, ohne dass eine Ausnahme geworfen wird, so wird der `catch` Block übersprungen und das Programm wird mit der nächsten Anweisung fortgesetzt, in unserem Beispiel mit `return 0;`.

Immer wenn wir eine `try` Anweisung verwenden, müssen wir sorgfältig die dabei auftretenden Seiteneffekte beachten. Wie müssen davon ausgehen, dass alle Anweisungen zwischen `try` und `catch` eine Ausnahme werfen könnten. Geschieht dies, so werden alle Anweisungen, die nach der Ausnahme ausgeführt werden sollten, übersprungen. Es ist sehr wichtig zu erkennen, dass die Anweisung, die direkt nach einer Ausnahme ausgeführt wird, nicht unbedingt als nächste Anweisung im Programmtext erscheint.

Nehmen Sie beispielsweise an wir hätten unsere Ausgabe (etwas knapper) wie folgt geschrieben:

```
// Dieses Beispiel funktioniert nicht!
try
{
    streamsize prec = cout.precision();
    cout << "Deine Gesamtnote ist: " << setprecision(2)
         << gesamt(vordiplom, diplom, wp) << setprecision(prec)
         << endl;
} ...
```

Das Problem bei diesem Code besteht darin, dass, obwohl die << Operatoren von links nach rechts ausgeführt werden, keine bestimmte Reihenfolge festgelegt ist, in der der Compiler die Operanden bewerten muss. Insbesondere ist es möglich, dass `gesamt` aufgerufen wird, **nachdem** `Deine Gesamtnote ist:` ausgegeben wurde. Wirft `gesamt` eine Ausnahme, so könnte die Ausgabe, diese falsche Meldung enthalten. Außerdem könnte der erste Aufruf von `setprecision` die Genauigkeit der Ausgabe auf 2 setzen, ohne dem zweiten Aufruf von `setprecision` die Möglichkeit zu geben, die Genauigkeit

zurückzusetzen. Alternativ ist es auch denkbar, dass der Compiler **zuerst** `gesamt` aufruft, bevor irgendetwas ausgegeben wurde. Ob dies geschieht ist ausschließlich von der C++ - Implementierung (dem Compiler) abhängig.

Diese Analyse erklärt, warum wir den Ausgabeblock in drei Anweisungen auftrennen: die erste Anweisung garantiert, dass `gesamt` aufgerufen wird, bevor irgendeine Ausgabe erzeugt wird.

Als Faustregel gilt, dass mehr als ein Seiteneffekt in einer einzelnen Anweisung vermieden werden soll. Das Werfen einer Ausnahme ist ein Seiteneffekt, also sollte eine Anweisung, die eine Ausnahme wirft, keine anderen Seiteneffekte verursachen, insbesondere keine Eingabe oder Ausgabe.

## *4.2 Datenstrukturen*

Die Berechnung der Gesamtnote für einen Studenten mag für diesen einen Studenten vielleicht nützlich sein, aber die Berechnung der Gesamtnote ist so einfach, dass sie mit einem Taschenrechner ebenso gut wie mit unserem Programm erledigt werden kann. Andererseits benötigen wir die Diplomnoten für ***alle*** Studenten eines Jahrgangs. Wir werden unser Programm nun so ändern, dass es diese Anforderung erfüllt.

Anstatt interaktiv die Noten jedes einzelnen Studenten einzugeben, nehmen wir an, dass die Daten wie in einer Datei vorliegen, die Namen und Noten der Studenten enthält. Auf jeden Namen folgen in einer Zeile Vordiplom- und Diplomnote, anschließend eine oder mehrere WP-Noten:

```
Schmitt 4 4 2 3 4 2 4
Müller 1 1 2 1 2 1 3
```

Das Programm soll die Gesamtnote für jeden Studenten berechnen und dabei den Median verwenden: Der median der WP-Noten zählt 40%, die Diplomnote zählt 40% und die Vordiplomnote zählt 20%.

Für diese Eingabe soll folgende Ausgabe entstehen:

```
Müller      1.4
Schmitt     3.6
```

Die Ausgabe soll alphabetisch sortiert werden und wir möchten, dass die Noten untereinander in einer Spalte erscheinen.

Diese Anforderungen implizieren, dass wir die Daten jedes Studenten ***zusammenhängend*** speichern, um sie alphabetisch sortieren zu können. Außerdem müssen wir die Länge des längsten Namens herausfinden, um die Anzahl Leerzeichen zwischen Namen und Gesamtnote berechnen zu können.

Unter der Annahme, dass wir eine Möglichkeit haben, die Daten eines Studenten zu speichern, können wir die Daten aller Studenten in einem `vector` ablegen. Wir beginnen nun, indem wir eine Datenstruktur anlegen, die die Studentendaten beinhaltet und schreiben einige Funktionen zum Lesen und Bearbeiten dieser Daten.

### *4.2.1 Datenstruktur für die Daten eines Studenten*

Wir wissen, dass wir die Daten jedes Studenten lesen und anschließend alphabetisch sortieren müssen. Dabei möchten wir Name und Noten gemeinsam bearbeiten, d.h. wir benötigen eine Möglichkeit ***alle*** Information eines Studenten an einem Platz zu speichern. Dieser Platz soll eine Datenstruktur sein, die den Namen des Studenten, seine Vordiplom- und Diplomnote, sowie alle WP-Noten beinhaltet.

In C++ wird diese Datenstruktur wie folgt definiert:

```
struct StudentInfo
{
    string name;
    double vordiplom;
    double diplom;
    vector<double> wp;
};  // Bitte beachten: Dieses Semikolon ist notwendig!!
```

Diese **`struct`** – Definition bedeutet, dass `StudentInfo` ein Typ ist, der vier Datenelemente besitzt. Da `StudentInfo` ein Typ ist, können wir Variable und Objekte dieses Typs anlegen, die jeweils eine Instanz der vier Datenelemente beinhalten.

Jedes Objekt des Typs `StudentInfo` enthält die Information zu einem Studenten. Weil `StudentInfo` ein Typ ist, können wir ein `vector<StudentInfo>` – Objekt verwenden , um die Information für eine beliebige Anzahl von Studenten aufzunehmen, genau so wie wir ein `vector<double>` – Objekt verwendeten, um die unbekannte Anzahl von WP – Noten aufzunehmen.

### *4.2.2 Verwalten der Studentendaten*

Unterteilen wir unser Problem in einzelne, überschaubare Komponenten, so stellen wir fest, dass es aus drei getrennten Schritten besteht, die jeweils von einer Funktion repräsentiert werden: Wir müssen die Daten eines Studenten in ein `StudentInfo` – Objekt einlesen, wir müssen die Gesamtnote für ein `StudentInfo` – Objekt berechnen und wir müssen einen `vector` von `StudentInfo` – Objekten sortieren.

Die Funktion, die die Daten eines Studenten einliest, sieht ganz ähnlich aus, wie die Funktion zum Einlesen der WP-Noten `read_wp` aus Kapitel 4.1.3. Tatsächlich können wir dieses Funktion zum einlesen der WP-Noten verwenden. Zusätzlich müssen wir den Namen und Vordiplom- und Diplomnote eines Studenten einlesen:

```
istream& read(istream& is, StudentInfo& s)
{
    // Name, Vordiplomnote und Diplomnote einlesen und speichern
    is >> s.name >> s.vordiplom >> s.diplom;

    read_hw(is, s.wp);  // WP-Noten einlesen und speichern
    return is;
}
```

Diese Funktion benötigt genau wie `read_wp` zwei Referenzen als Argumente: eine auf den `istream` von dem gelesen wird und eine andere auf das Objekt, in dem die eingelesenen Daten abgespeichert werden. Verwenden wir den Parameter `s` innerhalb der Funktion, so verändern wir den Inhalt des Objektes, das wir als Argument übergeben hatten.

Die nächste Funktion, die wir benötigen, berechnet die Gesamtnote für ein `StudentInfo` – Objekt. Den größten Teil dieses Problems lösten wir bereits, als wir die Funktion `gesamt` in Kapitel 4.1.2 definierten. Wir setzen die Arbeit fort, in dem die Funktion `gesamt` mit einer Version überladen, die die Gesamtnote für ein `StudentInfo` – Objekt berechnet:

```
double gesamt(const StudentInfo& s)
{
    return gesamt(s.vordiplom, s.diplom, s.wp);
}
```

Diese Funktion arbeitet mit einem Objekt des Typs `StudentInfo` und hat als Rückgabewert die Gesamtnote. Beachten sie, dass der Parameter den Typ `const StudentInfo&` hat und nicht den Typ `const StudentInfo`, so dass wir beim Aufruf den Aufwand für das Kopieren des ganzen `StudentInfo` – Objektes vermeiden.

Als letzte Aufgabe, bevor wir das ganze Programm erstellen können, müssen wir entscheiden wie wir den `vector` der `StudentInfo` – Objekte sortieren wollen. In der `median` – Funktion sortierten wir den `vector<double>` Parameter mit dem Namen `vec` mit der Bibliotheksfunktion `sort`:

```
sort(vec.begin(), vec.end());
```

Nehmen wir nun an, unsere Daten sind in einem `vector` mit dem Namen `studenten`, so können wir ***nicht*** schreiben:

```
sort(studenten.begin(), studenten.end()); // Dies ist nicht ganz richtig!!
```

Warum nicht? Zur Beantwortung dieser Frage müssen wir ein wenig abstrakt darüber nachdenken, wie die `sort` – Funktion arbeitet. Insbesondere woher `sort` weiß, wie die Werte eines `vector`s anzuordnen sind.

Die `sort` – Funktion muss zwei Elemente eines `vector`s vergleichen, um sie in die richtige Reihenfolge zu bringen. Für diesen Vergleich verwendet sie den Operator <, unabhängig davon, Elemente welchen Typs in dem `vector` enthalten sind. Wir können `sort` für einen `vector<double>` verwenden, weil der < – Operator in der Lage ist zwei `double` – Werte zu vergleichen und ein vernünftiges Ergebnis liefert. Was würde geschehen, sollte `sort` versuchen, zwei Elemente des Typs `StudentInfo` zu vergleichen? Der < – Operator hat keine vernünftige Bedeutung bei der Anwendung auf `StudentInfo` – Objekte, daher wird der Compiler eine Fehlermeldung erzeugen.

Glücklicherweise gibt es eine `sort` – Funktion, die als drittes Argument ein ***predicate*** erwartet. Ein *predicate* ist eine Funktion vom Typ `bool`. Ist das dritte Argument vorhanden, wird es von `sort` anstelle des < – Operators verwendet, um den Vergleich zweier Elemente durchzuführen. Wir müssen daher eine Funktion vom Typ `bool` definieren, die zwei Argumente vom Typ `StudentInfo` akzeptiert und als Ergebnis mitteilt, ob das erste Argument kleiner ist als das zweite. Wir möchten die Studenten alphabetisch nach dem Namen sortieren, also schreiben wir eine Vergleichsfunktion, die nur die Namen vergleicht:

```
bool compare(const StudentInfo& x, StudentInfo& y)
{
    return x.name < y.name;
}
```

Diese Funktion delegiert einfach die Vergleichsoperation an die `string` Klasse, die einen < – Operator zum Vergleich von `strings` zur Verfügung stellt.

Mit dieser Definition von `compare` können wir den `vector` sortieren, in dem wir die `compare` – Funktion als drittes Argument beim Aufruf von `sort` übergeben:

```
sort(studenten.begin(), studenten.end(), compare);
```

`sort` verwendet jetzt die Funktion `compare` für den Vergleich zweier `StudentInfo` – Objekte.

### *4.2.3 Ausgabe der Gesamtnoten*

Die Funktionen zur Bearbeitung der Studentendaten stehen zur Verfügung, sodass wir die Ausgabe erzeugen können:

```
int main()
{
    vector<StudentInfo> studenten;
    StudentInfo daten;
    string::size_type maxlen = 0;


    while (read(cin, daten))
    {
        maxlen = max(maxlen, daten.Name.size());
        studenten.push_back(daten);
    }

    sort(studenten.begin(), studenten.end(), compare);

    for (vector<StudentInfo>::size_type i = 0;
         i != studenten.size(); i++)
    {
```

```
            cout << studenten[i].Name
                 << string(maxlen + 1 - studenten[i].Name.size(), ' ');

            try
            {
                double GesamtNote = gesamt(studenten[i]);
                streamsize prec = cout.precision();
                cout << setprecision(2) << GesamtNote << setprecision(prec);
            }
            catch (domain_error e)
            {
                cout << e.what();
            }
            cout << endl;
        }
        return 0;
    }
```

Der größte Teil diese Programms ist bekannt, aber einige Stellen sind neu.

Die erste Neuigkeit ist der Aufruf der Bibliotheksfunktion `max`., die im header `<algorithm>` definiert ist. Oberflächlich betrachtet, ist das Verhalten von `max` offensichtlich. Ein Aspekt des Verhaltens ist jedoch nicht offensichtlich: Aus Gründen, die wir erst in späteren Kapiteln erklären können, müssen beide Argumente den gleichen Typ besitzen. Daher muss `maxlen` unbedingt als `string::size_type` definiert sein.

Die zweite Neuigkeit ist die der Ausdruck

```
string(maxlen + 1 - studenten[i].Name.size(), ' ');
```

Dieser Ausdruck konstruiert ein Objekt ***ohne*** Namen vom Typ `string`. Das Objekt enthält `maxlen + 1 - studenten[i].Name.size()` Leerzeichen. Der Ausdruck ist ähnlich zur Definition von `spaces` im Kapitel 1.2 / S.19., verwendet allerdings keinen Variablennamen.

Die `for` – Anweisung verwendet den Index `i`, um die einzelnen `studenten` – Elemente zu bearbeiten. Wir erhalten den Namen in dem wir `studenten` indizieren, um das richtige `StudentInfo` Objekt zu erhalten. Von diesem Objekt schreiben wir das Element `Name` und füllen dann mit genügend Leerzeichen nach rechts auf.

Danach schreiben wir die Gesamtnote für jeden Studenten. Hatte der Student keine WP-Noten, wirft die Funktion zur Berechnung der Gesamtnote ein Ausnahme. In diesem Fall fangen wir die Ausnahme und geben den Hinweis aus, der als Teil des Ausnahmeobjektes geworfen wurde. Jeder der Ausnahmen aus der Standardbibliothek, wie z.B. `domain_error`, merkt sich das optionale Argument, das zur Beschreibung der Ausnahmeursache verwendet wurde. Jeder dieser Ausnahmetypen erzeugt eine Kopie dieses Argumentes, die durch die Elementfunktion `what()` verfügbar ist. Die `catch` Anweisung diese Programms gibt dem Ausnahmeobjekt das von der Funktion `gesamt` geworfen wird, den Namen `e`, so dass es mit dem Aufruf der Funktion `e.what()` die zur Ausnahme gehörende Meldung ausgeben kann. Wurde keine Ausnahme geworfen, verwenden wir den Manipulator `setprecision` um zu spezifizieren, dass die Ausgabe mit 2 signifikanten Stellen erfolgen soll und geben anschließend die Gesamtnote aus.

### *4.3 Das komplette Programm*

Bis jetzt haben wir einige Abstraktionen definiert (Funktionen und Datenstrukturen), die hilfreich bei der Lösung unserer Probleme sind. Der einzige bisher bekannte Weg diese Abstraktionen zu nutzen, besteht darin, alle Definitionen in eine einzelne Datei zu packen und diese Datei zu kompilieren. Offensichtlich wird dieser Ansatz sehr schnell kompliziert. Um diese Komplexität zu reduzieren bietet C++, wie viele andere Programmiersprachen auch, den Mechanismus der ***getrennten Übersetzung***, die es ermöglicht das Programm in verschiedene Dateien aufzuteilen und diese unabhängig voneinander getrennt zu kompilieren.

Zuerst überlegen wir uns, wie wir die Funktion `median` anderen Funktionen in einer Datei zur Verfügung stellen können. Dazu schreiben wir die Definition der Funktion `median` in einer eigenen Datei, die wir dann separat übersetzen können. Diese Datei muss die Deklaration aller Namen beinhalten, die `median` verwendet. Aus der Standardbibliothek sind dies der Typ `vector`, die Funktion `sort` und die Ausnahme `domain_error`, so dass wir die entsprechenden Header in include-Anweisungen angeben müssen:

```
// Quellcodedatei für die Funktion median
#include <algorithm>
#include <stdexcept>
#include <vector>

using std::domain_error;
using std::sort;
using std::vector;

// Berechne den Median eines Argumentes vom Typ vector<double>
double median(vector<double> vec)
{
    //  Funktionsrumpf wie in Kapitel 4.1.1
}
```

Wie allen anderen Dateien auch, müssen wir dieser Quellcodedatei einen Namen geben. Der C++ - Standard gibt uns keine Anweisung, wie Quellcodedateien zu benennen sind, aber üblicherweise sollte der Name einer Quellcodedatei einen Hinweis auf ihren Inhalt geben. Zusätzlich erwarten die meisten C++ - Implementierungen (z.B. Microsoft Visual C++ oder Borland C++ - Builder) eine bestimmte Endung für Quellcodedateien: `.cpp`. Daher packen wir die Funktion `median` in eine Datei mit dem Namen `median.cpp`. Im nächsten Schritt müssen wir die Funktion `median` anderen Anwendern zur Verfügung stellen. Analog zur Standardbibliothek, die die Namen, die sie definiert in ***Headern*** anderen Anwendern zur Verfügung stellt, können wir eine eigene ***Headerdatei*** schreiben, die anderen Anwendern ermöglicht, die von uns definierten Namen zu verwenden. Wir können z.B. eine Datei mit dem Namen `median.h` zur Verfügung stelle, die die Deklaration der Funktion `median` enthält. Diese Datei kann dann von anderen Anwendern wie folgt verwendet werden:

```
// eine sinnvollere Verwendung von median
#include "median.h"
#include <vector>

int main()
{
    ...
}
```

Wir verwenden Anführungszeichen in der include-Anweisung für `median.h`, um dem Compiler mitzuteilen, dass der gesamte Inhalt der *Headerdatei* an Stelle der include – Anweisung in unser Programm ***kopiert*** werden soll.

Bitte beachten sie, dass wir unsere eigenen Header als Headerdateien bezeichnen, während wir die von der Implementierung zur Verfügung gestellten Header als Standard – Header bezeichnen und nicht als Standard – Headerdateien. Der Grund dafür ist, dass Header nicht zwingend als Dateien implementiert werden müssen. Obwohl mit der include – Anweisung Headerdateien und System-Header hinzugefügt werden, gibt es keine Notwendigkeit, dass beide in der gleichen Art und Weise implementiert werden.

Da wir nun wissen, wie man eine Headerdatei zur Verfügung stellt und verwendet, bleibt die offensichtliche Frage nach dem der Inhalt der Datei. Die einfache Antwort ist, dass wir eine ***Deklaration*** der Funktion `median` schreiben müssen. Dies tun wir indem wir den Rumpf der Funktion durch ein Semikolon ersetzen. Wir können, falls gewünscht, auch die Namen der Funktionsparameter weglassen, da sie ohne Funktionsrumpf bedeutungslos sind:

```
double median(vector<double>);
```

Diese Deklaration ist nicht ausreichend, denn wir müssen für alle verwendeten Namen die entsprechenden include – Anweisungen hinzufügen:

```
// median.h
#include <vector>
double median(std::vector<double>);
```

Wir benötigen den `vector` – Header weil wir `std::vector` als Parameter von `median` verwenden. Die Begründung, warum wir explizit `std::vector` anstelle einer `using` – Deklaration verwenden ist etwas subtil.

Generell sollten Headerdateien nur die Namen deklarieren, die sie auch verwenden. Durch die Begrenzung auf die in der Headerdatei verwendeten Namen sorgen wir für größtmögliche Flexibilität für die Anwender. Beispielsweise verwenden wir den qualifizierten Namen für `std::vector`, weil wir nicht wissen können wie der Anwender der Funktion `median` sich auf `std::vector` beziehen möchte. Eventuell möchten die Anwender unserer Funktion ***keine*** `using` – Deklaration für `vector` verwenden. Hätten wir eine `using` – Deklaration für `vector` in unserer Headerdatei, so bekämen alle Programme diese `using std::vector` – Deklaration, die unsere Headerdatei verwenden, unabhängig davon, ob sie diese Deklaration möchten oder nicht. Daher sollten Headerdateien generell voll qualifizierte Namen verwenden und `using` – Deklarationen vermeiden.

Es gibt noch ein letztes Detail zu beachten: Headerdateien sollten sicherstellen, dass kein Problem entsteht, sofern eine Headerdatei an mehr als einer Stelle im zu übersetzen Programm eingebunden wird. In unserem Beispiel ist die Headerdatei bereits problemlos verwendbar, da sie nur Deklarationen enthält. Es ist jedoch eine gute Angewohnheit in allen Headerdateien ein mehrfaches Einbinden zu verhindern. Wir erreichen dies durch einige ***Präprozessoranweisungen***:

```
#ifndef _median_h
#define _median_h

// median.h – endgültige Version
#include <vector>
double median(std::vector<double>);

#endif
```

Die Anweisung **`#ifndef`** überprüft, ob `median_h` bereits definiert ist. Dies ist der Name einer ***Präprozessor Variable***, die eine von mehreren Möglichkeiten darstellt, mit der man steuern kann, wie ein Programm übersetzt wird. Eine vollständige Diskussion des Präprozessors ist nicht Bestandteil diese Skriptes.

In diesem Zusammenhang beauftragt die `#ifndef` – Anweisung den Präprozessor alles zwischen dieser Anweisung und dem darauf folgenden `#endif` zu verarbeiten, sofern der angegeben Name *nicht* definiert ist. Wir müssen einen Namen wählen, der eindeutig im gesamten Programm ist, deshalb erzeugen wir den Namen aus dem Namen der Headerdatei und hoffen, dass dieser Name nirgendwo sonst im Programm verwendet wird.

Wird `median.h` das erste Mal in einem Programm eingebunden, so ist `_median_h` nicht definiert, so dass der Präprozessor den Rest der Datei verarbeitet. Das erste was er zu tun hat, ist `_median_h` zu definieren, so dass weitere Versuche `_median_h` einzubinden keine Auswirkungen haben.

### *4.4. Aufteilen des Programms zur Berechnung der Gesamtnote*

Da wir nun wissen wie die Funktion `median` getrennt übersetzt werden kann, können wir im nächsten Schritt die `StudentInfo` Struktur und die zugehörigen Funktionen in eine separate Datei packen:

```
#ifndef _StudentInfo_h
#define _StudentInfo_h
```

```
// StudentInfo.h Headerdatei
#include <iostream>
#include <string>
#include <vector>

struct StudentInfo
{
    std::string name;
    double vordiplom;
    double diplom;
    std::vector<double> wp;
};

bool compare(const StudentInfo&, const StudentInfo&);
std::istream& read(std::istream&, StudentInfo&);
std::istream& read_wp(std::istream&, std::vector<double>&);

#endif
```

Beachten sie, dass wir explizit `std::` verwenden, um Namen aus der Standardbibliothek vollständig zu qualifizieren und dass `StudentInfo` die Funktionen `compare`, `read` und `read_wp` deklariert, die eng mit der Struktur `StudentInfo` zusammenhängen. Wir werden diese Funktionen nur verwenden, sofern wir auch die Struktur verwenden, also macht es Sinn diese Funktionen in der gleichen Datei wie die Struktur zu deklarieren.

Die Funktionen sollten in einer Quellcodedatei definiert sein, die in etwa so aussieht:

```
// Quellcodedatei für mit StudentInfo assoziierte Funktionen
#include "StudentInfo.h"

using std::istream;
using std::vector;

bool compare(const StudentInfo& x, const StudentInfo& y)
{
    return x.name < y.name;
}

istream& read(istream& is, StudentInfo& s)
{
    // Definition wie in Kapitel 4.2.2, S.56
}

istream& read_wp(istream& is, vector<double>& wp)
{
    // Definition wie in Kapitel 4.1.3, S. 51
}
```

Beachten sie, dass diese Quellcodedatei die Deklarationen und Definitionen der Funktionen enthält, weil wir die Headerdatei `StudentInfo.h` einbinden. Dies ist harmlos und tatsächlich eine gute Idee. Sie gibt dem Compiler die Möglichkeit die Konsistenz von Deklarationen und Definitionen zu überprüfen.

Beachten sie weiterhin, dass es in einer Quellcodedatei keine Probleme mit `using` – Anweisungen gibt, da eine Quellcodedatei, anders als eine Headerdatei, keine Auswirkungen auf die Programme hat, die die Funktionen verwenden.

Nun fehlt noch die Headerdatei für die verschiedenen überladenen Funktionen mit dem Namen `gesamt`:

```
#ifndef _gesamt_h
#define _gesamt_h
```

```
// gesamt.h
#include <vector>
#include "StudentInfo.h"

double gesamt(double, double, double);
double gesamt(double, double, const std::vector<double>&);
double gesamt (const StudentInfo&);

#endif
```

Beachten sie, dass die Deklarationen der überladenen Funktionen an einer Stelle sehr übersichtlich ist, und es erleichtert alle Alternativen zu erkennen. Wir werden nun alle drei Funktionen in einer Datei definieren, da sie eng miteinander verknüpft sind:

```
// gesamt.cpp
#include <stdexcept>
#include <vector>
#include "gesamt.h"
#include "median.h"
#include "Student_Info.h"

using std::domain_error;
using std::vector;
// Definitionen der Funktionen gesamt aus den Kapiteln 4.1, S.47, 4.1.2,
// S.49 und 4.2.2, S. 57
```

### *4.5 Die überarbeitete Version des Programms zur Berechnung der Gesamtnote*

Wir können das komplette Programm jetzt wie folgt schreiben:

```
#include <algorithm>
#include <iomanip>
#include <ios>
#include <iostream>
#include <stdexcept>
#include <string>
#include <vector>
#include "StudentInfo.h"
#include "gesamt.h"

using namespace std;

int main()
{
      vector<StudentInfo> studenten;
      StudentInfo daten;
      string::size_type maxlen = 0;       // Länge des längsten Namens

      while (read(cin, daten))            // Einlesen und Speichern der Daten
      {                                   // der Studenten
            maxlen = max(maxlen, daten.name.size());
            studenten.push_back(daten);
      }

      // Studentendaten alphabetisch aufsteigend sortieren
      sort(studenten.begin(), studenten.end(), compare);

      // Namen und Gesamtnote ausgeben
      for (vector<StudentInfo>::size_type i = 0; i < studenten.size(); i++)
      {
            //  Name ausgeben und rechts mit Leerzeichen auffüllen
            cout << studenten[i].name
```

```
                    << string(maxlen + 1 - studenten[i].name.size(), ' ');

                // Berechnen und Ausgeben der Gesamtnote
                try
                {
                      double GesamtNote = gesamt(studenten[i]);
                      streamsize prec = cout.precision();

                      cout << setprecision(2) << GesamtNote <<
                              setprecision(prec);
                }
                catch (domain_error e)
                {
                      cout e.what();
                }

                cout << endl;
          }

          return 0;
    }
```

Dieses Programm sollte einfach zu verstehen sein. Wie gewöhnlich beginnen wir mit den notwendigen `includes` und der `using` – Anweisung. In diesem Programm müssen wir die Systemheader und unsere eigenen Headerdateien einbinden. Diese Headerdateien stellen die Definition des Typs `StudentInfo` und die Deklarationen der Funktionen zur Bearbeitung diese Typs und zur Berechnung der Gesamtnote zur Verfügung. Die Funktion `main` ist die gleiche, die wir in Kapitel 4.2.3, S. 59, präsentiert hatten.

### *4.6 Details*

***Programmstruktur:***

`#include <`S*ystem-Header*`>`

Spitze Klammern, `< >`, beinhalten System-Header. System-Header können als Dateien implementiert sein, müssen aber nicht.

`#include <"`Selbst definierter Headerdatei Name"`>`

Selbst definierte Headerdateien werden mit Anführungszeichen `include`d. Üblicherweise haben selbst definierte Header die Endung `.h`.

Headerdateien sollten vor mehrfachem Einfügen geschützt werden, indem sie eine `#ifndef` *_Headerdatei_Name* Anweisung verwenden. Header sollten vermeiden Namen zu deklarieren, die sie nicht verwenden. Insbesondere sollten sie keine `using` Anweisungen enthalten, sondern stattdessen Namen aus der Standardbibliothek explizit mit `std::` beginnen.

***Typen:***

`T&` Bezeichnet eine Referenz auf den Typ `T`. Wird am häufigsten verwendet, um einen Parameter an eine Funktion zu übergeben, der innerhalb der Funktion geändert werden soll. Argumente zu solchen Parametern müssen *lvalues* (nicht-temporäre Objekte) sein.

`const T&` Bezeichnet eine Referenz auf den Typ `T`, die nicht dazu verwendet werden darf, den Wert, mit dem die Referenz verknüpft ist, zu ändern. Wird meistens verwendet, um zu vermeiden, dass ein Parameter beim Funktionsaufruf kopiert wird.

***Strukturen:*** Eine Struktur ist ein Typ, der null oder mehrere Elemente enthält. Jedes Objekt diesen Typs, erhält eine eigene Instanz jedes seiner Elemente. Jede Struktur benötigt eine Definition:

```
struct Typname
{
    Typ Elementname;
    ...
};  // Beachten Sie das Semikolon
```

Wie alle Definitionen, darf auch eine Strukturdefinition nur einmal innerhalb einer Quellcodedatei erscheinen. Deshalb sollten Strukturen immer in entsprechend geschützten Headerdateien definiert werden.

***Funktionen:*** Eine Funktion muss in jeder Datei, in der sie verwendet wird, *deklariert* werden und darf insgesamt nur einmal *definiert* werden. Deklarationen und Definitionen haben eine ähnliche Form:

```
Rückgabetyp Funktionsname(Parameter-Deklarationen)          // Funktionsdeklaration

[inline] Rückgabetyp Funktionsname(Parameter-Deklarationen) // Funktionsdefinition
{
    // Funktionsrumpf
}
```

*Rückgabetyp* ist der Typ der Rückgabewertes, *Parameter-Deklarationen* ist eine durch Kommata getrennte Liste der Typen der Parameter der Funktion. Funktionen müssen vor ihrem Aufruf deklariert werden. Der Typ jedes Argumentes muss beim Aufruf mit dem Typ des entsprechenden Parameters übereinstimmen. Für komplizierte Rückgabetypen kann sich die Syntax ändern.

Funktionsnamen dürfen *überladen* werden: Der gleiche *Funktionsname* darf mehrfach definiert werden, sofern sich die Funktionen in Anzahl oder Typ der Parameter unterscheiden. Die Implementierung unterscheidet zwischen einer Referenz und einer `const` Referenz auf den gleichen Typ.

Wahlweise können wir eine Funktionsdefinition mit **`inline`** erweitern. Dies hat zur Folge, dass der Compiler die Funktionsaufrufe zu *inline* Sourcecode erweitert, falls es ihm sinnvoll erscheint, d.h. dass im Sourcecode jeder Funktionsaufruf durch eine Kopie des Funktionsrumpfes ersetzt wird und so der Aufwand für den Funktionsaufruf vermieden wird.

***Ausnahmebehandlung:***

```
try
{
    // Code
}
catch(t)
{
    // Code
}
```

Beginnt einen Block der möglicherweise eine Ausnahme wirft.

Beendet den `try` Block und behandelt Ausnahmen vom Typ `t`. Der im `catch`- Block definierte Code, führt die Aktionen aus, die zur Behandlung der in `t` übermittelten Ausnahme sinnvoll sind.

`throw e;` Beendet die aktuelle Funktion. Wirft die Ausnahme mit dem Wert `e` zurück zur aufrufenden Funktion.

***Ausnahme Klassen (exception classes)*:** Die Standardbibliothek definiert verschiedene Ausnahme Klassen, deren Namen bereits einen Hinweis darauf gegeben, für welche Fehler sie verwendet werden können:

```
logic_error     domain_error    invalid_argument
length_error    out_of-range    runtime_error
range_error     overflow_error  underflow_error
```

`e.what()` Gibt einen Wert zurück, der über die Fehlerursache informiert.

***Bibliotheksoperationen:***

| | |
|---|---|
| `s1 < s2` | Vergleicht die `strings` `s1` und `s2` lexikographisch. |
| `s.width(n)` | Setzt die Breite des Ausgabestroms `s` für die nächste Ausgabeoperation auf `n` Stellen (oder lässt die Breite unverändert, sofern kein Argument angegeben wird). Die Ausgabe wird von links mit der notwendigen Anzahl an Leerzeichen aufgefüllt. Rückgabewert ist die vorherige Breite. Die Standard Ausgabeoperatoren verwenden die eingestellte Breite und rufen dann `width(0)` auf, um die Breite zurückzusetzen. |
| `setw(n)` | Gibt ein Wert vom Typ `streamsize` zurück, der, sofern man ihn auf einen Ausgabestrom `s` ausgibt, die identische Auswirkung hat, wie der Aufruf `s.width(n)`. |

### *Aufgaben 1*

4.1 Schreiben Sie eine Funktion `power`, die über zwei Parameter verfügt, eine Fließkommazahl y und eine Ganzzahl x und deren Rückgabewert gleich $y^x$ ist.

4.2 Schreiben Sie eine Funktion, die eine positive ganz Zahl (n) als Parameter hat und als Rückgabewert n! liefert.

4.3 Ändern Sie die Funktion von Aufgabe 4.2 so, dass das Ergebnis in das beim Aufruf übergebene Argument geschrieben wird (Tipp: Referenzen verwenden).

4.4 Schreiben Sie eine Funktion `max`, die zwei Ganzzahlen als Parameter hat und die größere als Rückgabewert liefert (Verwenden Sie die bedingte Bewertung!).

4.5 Schreiben Sie noch eine Funktion `max`, die zwei Fließkommazahlen als Parameter hat und die größere als Rückgabewert liefert (Verwenden Sie die bedingte Bewertung).

4.6 Ändern Sie die Funktionen aus 4.4 und 4.5 so, dass jeweils Referenzen auf den größeren Parameter als Rückgabewert geliefert werden.

4.7 Schreiben Sie eine Funktion, mit einem Vektor von Ganzzahlen als Parameter, die die Elemente des (Original-) Vektors aufsteigend sortiert und das größte Element als Rückgabewert liefert.

### *Aufgaben 2*

4.8 Schreiben Sie eine Funktion `int quadrat(int i)`, die die Zahlen von 1 bis 100 quadriert. Definieren Sie die Funktion in einer eigenen cpp-Datei und schreiben Sie die Deklaration in eine eigene Header-Datei. Bei einem Argument außerhalb des angegebenen Wertebereichs, soll ein `domain_error` geworfen werden und im Hauptprogramm gefangen werden.

Die Ausgabe soll in zwei Spalten erfolgen. Benutzen Sie den Manipulator `setw(n)`, um die Spaltenbreite der Ausgabe einzustellen.

4.9 Ändern Sie die Funktion `quadrat` und verwenden Sie nun `double` Werte. Verwenden Sie Manipulatoren, um die Ausgabe in Spalten zu erzeugen.

4.10 Nehmen Sie an das folgende Programm sei syntaktisch korrekt. Wie lautet dann der Rückgabetyp der Funktion `f`:

```
double d = f()[n];
```

### *Aufgaben 3*

4.11 Schreiben Sie einen Vektor von Strukturen, wobei jede Struktur den Name eines Monats und die Anzahl seiner Tage enthalten. Initialisieren Sie den Vektor mit den 12 Monaten und geben Sie den Vektor aus.

4.12 Schreiben Sie ein Programm, das Name, Vorname, Vordiplom-, Diplom- und WP-Noten für mehrere Studenten einliest und alphabetisch sortiert wieder ausgibt. Definieren sie dazu eine Struktur und speichern Sie die Eingabe in einem Vektor von dieser Struktur ab.

4.13 Schreibe Sie einen Vektor von Strukturen, wobei jede Struktur zwei Ganzzahlen enthält. Die zweite Ganzzahl soll das Quadrat der ersten sein. Initialisieren Sie den Vektor mit den Zahlen von 1 bis 1000 und verwenden sie den Vektor beispielhaft zur Berechnung des Quadrates der natürlichen Zahlen von 1 bis 1000 (Eine Tabelle dieser Art nennt man auch *Look-Up-Table*, oder *LUT*).

4.14 Lesen Sie eine Folge von Wörtern aus der Eingabe. Benutzen Sie `Ende` als das Wort, das die Eingabe beendet. Geben Sie die Wörter in der Reihenfolge aus in der sie eingegeben wurden. Geben Sie kein Wort mehrmals aus. Schreiben Sie jedes Wort in einer neuen Zeile in Spalte 1 und geben Sie in der zweiten Spalte aus, wie oft das Wort eingegeben wurde. Modifizieren Sie das Programm so, dass es die Wörter vor der Ausgabe alphabetisch sortiert.

### *Aufgaben 4*

4.15 Schreiben Sie ein Programm, das die Zahlen im Intervall [0, 255] (die Grauwerte eines Grauwertbildes) invertiert. Verwenden Sie dazu eine Look-Up-Table (s. Aufgabe 4.13).

4.16 Schreiben Sie ein Programm zur Grauwertspreizung. Das Intervall [0,117] soll auf das Intervall [0, 255] abgebildet werden. Verwenden Sie eine LUT.

4.17 Schreiben Sie ein Programm, das eine beliebige Anzahl von Zahlen aus dem Intervall [0, 255] einliest. Geben Sie in Spalte 1 die Zahlen aufsteigend sortiert aus und in Spalte 2 wie oft die jeweilige Zahl eingegeben wurde.

4.18 Ändern Sie die Ausgabe aus 4.17 wie folgt: Geben Sie anstelle der Anzahl einen Strich aus, dessen Länge die Anzahl repräsentiert. Zusatzaufgabe: Normieren Sie die Strichlänge so, dass die größte Anzahl einer Strichlänge von 20 Zeichen entspricht.

# 5 Literaturliste

Aus folgendem Buch wurden weite Teile des Skriptes entnommen:

A. Koenig, B. Moo. *Intensivkurs C++*. Pearson Studium, 2003.

Eine gute Einführung in C++ gibt es vom Erfinder von C++:

B. Stroustrup. *Einführung in die Programmierung mit C++*. Pearson Studium, 2010.

Hinweise zur Bedeutung der Zahl *42* gibt es in:

D. Adams. *Per Anhalter durch die Galaxis*. Rogner & Bernhard, 1994.

Darüber hinaus gibt es eine Vielzahl von (oft schlechten) Büchern zum Thema C++. Bitte sprechen Sie mich an, falls Sie beabsichtigen ein anderes Buch zu kaufen.