Kapitel

3 Arbeiten mit größeren Datenmengen

Die Programme, die wir in den Kapiteln 1 und 2 besprochen haben, taten wenig mehr als eine Zeichenkette einzulesen und sie – manchmal noch mit etwas zusätzlicher Dekoration – wieder auszugeben. Die meisten Aufgabenstellungen sind jedoch komplizierter. Eine der häufigsten Problemstellungen in Programmen ist die Forderung, mehrere gleichartige Datenelemente zu behandeln.

Unsere Programme haben bereits damit begonnen – zumindest, sofern man `string` als mehrere einzelne Zeichen ansieht. Aber tatsächlich ist es genau die Fähigkeit, eine unbekannte Anzahl von Zeichen in ein einziges Objekt – zum Beispiel einen `string` – zu packen, die es uns ermöglicht, diese Programme einfach zu gestalten.

In diesem Kapitel werden wir weitere Wege verfolgen, wie wir mit Datenmengen umgehen, indem wir ein Programm schreiben, dass Prüfungs- und Hausarbeitsnoten von Studenten einliest und daraus eine Endnote errechnet. Im Laufe dieser Erörterung werden wir lernen, wie wir diese Noten speichern, auch wenn wir zu Beginn nicht wissen, wieviele Noten vergeben wurden.

3.1 Berechnung von Studiennoten

Stellen wir uns einen Studiengang vor, bei dem die Abschlussprüfung 40% der Endnote ausmacht, eine Vorprüfung 20% und der Durchschnitt der Hausarbeitsnoten die restlichen 40 %. Hier ist unser erster Versuch eines Programmes, mit dem ein Student sich seine Endnote ausrechnen kann:

```
#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()
{
    // Nach dem Namen fragen und einlesen.
    cout << "Bitte geben Sie Ihren Vornamen ein: ";
```

```
    string name;
    cin >> name;
    cout << "Hallo, "<< name << "!" << endl;

    // Nach den Prüfungsnoten fragen und einlesen.
    cout << "Bitte geben Sie die Vorprüfungs- "
         << "und Diplomprüfungsnote ein: ";
    double vor, diplom;
    cin >> vor >> diplom;

    // Nach den Hausarbeitsnoten fragen.
    cout << "Bitte alle Hausarbeitsnoten, "
            "gefolgt vom Datei-Ende-Zeichen: ";

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

    // eine Variable, in die wir einlesen
    double x;

    // Invariante:
    //    Wir haben anzahl Noten eingelesen und
    //    summe ist die Summe der ersten anzahl Noten.
    while(cin >> x) {
        ++anzahl;
        summe += x;
    }

    // Das Ergebnis ausgeben.
    streamsize prec = cout.precision();
    cout << "Ihre Endnote ist " << setprecision(3)
         << 0.2 * vor + 0.4 * diplom + 0.4 * summe / anzahl
         << setprecision(prec) << endl;
    return 0;
}
```

Wie üblich, beginnen wir mit `#include`-Direktiven und `using`-Deklarationen für die Bestandteile der Standardbibliothek, die wir benutzen wollen. Diese Bestandteile umfassen auch `<iomanip>` und `<ios>`, die wir bislang noch nicht benutzt haben. Der Header `<ios>` definiert `streamsize`, was dem Typ der Ein-/Ausgabe-Bibliothek zur Darstellung von Größen entspricht. Der Header `<iomanip>` definiert den Manipulator `setprecision`, der uns die Spezifikation der Anzahl von signifikanten Ziffern in der Ausgabeanweisung erlaubt.

Uns fällt auf, dass wir den Header `<iomanip>` nicht einbinden mussten, wenn wir den Manipulator `endl` benutzen wollen. Der Manipulator `endl` wird so oft benutzt, dass seine Definition in dem häufiger eingebundenen Header `<iostream>` enthalten ist, und nicht in `<iomanip>`.

Das Programm beginnt mit der Frage nach dem Namen des Studenten und den Vorprüfungs- und Diplomprüfungsnoten. Als Nächstes fragt es nach den Hausarbeitsnoten, die es einliest, bis es das Datei-Ende-Signal erkennt. Verschiedene C++-Implementierungen bieten den Nutzern verschiedene Wege an, dem Programm ein solches Signal zu senden; der häufigste Weg ist, eine neue Eingabezeile zu beginnen, die *Strg*-Taste zu betätigen und gleichzeitig `z` (bei Rechnern mit Microsoft Windows) oder `d` (bei Rechnern mit Unix- oder Linux-Systemen) zu drücken.

Während die Noten eingelesen werden, hält das Programm in `anzahl` die Anzahl der eingelesenen Noten fest, während in `summe` die Summe der bislang eingegebenen Noten gespeichert wird. Nachdem das Programm alle Noten eingelesen hat, gibt es eine Meldung mit der Endnote des Studenten aus. Dabei benutzt es `anzahl` und `summe`, um den Durchschnitt der Hausarbeitsnoten zu ermitteln.

Der größte Teil des Programmes sollte bereits bekannt sein, aber es gibt eine Reihe von neuen Nutzungsweisen, die wir erklären wollen.

Die erste neue Idee kommt in der Sektion, die die Prüfungsnoten einliest:

```
cout << "Geben Sie Ihre Vorprüfungs- "
     << "und Diplomprüfungsnote ein: ";
double vor, diplom;
cin >> vor >> diplom;
```

Die erste dieser Anweisungen sollte bekannt sein: Sie schreibt eine Nachricht, die in diesem Falle dem Studenten den nächsten Bedienungsschritt mitteilt. Die nächste Anweisung definiert `vor` und `diplom` als Variablen vom Typ `double`, was dem Built-in-Datentyp für Fließkommazahlen doppelter Genauigkeit entspricht. Es gibt auch einen Datentyp für Fließkommazahlen einfacher Genauigkeit, der `float` genannt wird. Obwohl es so scheint, als ob `float` der angemessene Datentyp für diese Aufgabe ist, ist es meist besser, `double` für Fließkommaberechnungen zu verwenden.

Die Namen dieser Datentypen gehen auf eine Zeit zurück, in der Speicher wesentlich teurer war als heutzutage. Der kürzere Fließkommatyp, `float` genannt, bietet lediglich eine Genauigkeit von ungefähr sechs signifikanten Dezimalstellen, was nicht einmal genügend ist, um den Preis eines Hauses auf den nächsten Cent genau anzugeben. Der Typ `double` bietet mindestens zehn signifikante Stellen, und uns sind Implementierungen bekannt, die mindestens 15 signifikante Stellen zur Verfügung stellen. Auf modernen Rechnern ist `double` meist genauer als `float`, und dabei nicht viel langsamer. Manchmal ist `double` sogar schneller.

Nachdem wir nunmehr die Variablen `vor` und `diplom` definiert haben, lesen wir Werte in sie ein. Genau wie der Ausgabeoperator (Abschnitt 0.7 auf Seite 19), so gibt auch der Eingabeoperator seinen linken Operanden als Ergebnis zurück. Wir können daher Eingabeoperationen analog zu Ausgabeoperationen verketten, also hat

```
cin >> vor >> diplom;
```

dasselbe Ergebnis wie

```
cin >> vor;
cin >> diplom;
```

Beide Formen lesen einen Zahlenwert von der Standardeingabe in `vor` und den nächsten Zahlenwert in `diplom` ein.

Die nächste Anweisung fordert den Studenten zur Eingabe der Hausarbeitsnoten auf:

```
cout << "Bitte alle Hausarbeitsnoten, "
        "gefolgt vom Datei-Ende-Zeichen: ";
```

Bei genauer Betrachtung stellt sich heraus, dass diese Anweisung nur einen einzigen <<-Operator enthält, obwohl sie zwei Stringliterale auszugeben scheint. Dies können wir so formulieren, denn zwei oder mehrere Stringliterale in einem Programm, die nur durch Whitespace getrennt sind, werden automatisch verkettet. Daher hat diese Anweisung dasselbe Ergebnis wie:

```
cout << Bitte alle Hausarbeitsnoten, gefolgt vom Datei-Ende-Zeichen: ";
```

Indem das Stringliteral in zwei Teile gesplittet wird, vermeiden wir Programmzeilen, die zu lang zum bequemen Lesen sind.

Der nächste Codeabschnitt definiert die Variablen, die wir benötigen, um die einzulesenden Informationen aufzunehmen. Der interessante Teil ist dabei:

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

Man beachte, dass wir sowohl `anzahl`, als auch `summe`, den selben Initialwert `0` geben. Der Wert `0` hat den Typ `int`, weswegen die Implementierung diesen in den Typ `double` konvertieren muss, um ihn für die Initialisierung von `summe` zu benutzen. Wir könnten diese Konvertierung vermeiden, indem wir `summe` mit `0.0` anstelle von `0` initialisieren, aber dies hat keinerlei praktischen Einfluss in diesem Zusammenhang: Jede vernünftige Implementierung wird die Konvertierung während der Compilierung vornehmen, so dass kein Laufzeit-Overhead entsteht, und das Ergebnis wird exakt das gleiche sein.

Wesentlich wichtiger als die Konvertierung ist in diesem Fall, dass wir diesen Variablen überhaupt einen Initialwert geben. Wenn wir einer Variable keinen Initialwert geben, verlassen wir uns implizit auf die ***Defaultinitialisierung***. Die Initialisierung die per default vorgenommen wird, hängt vom Typ der Variablen ab. Bei Objekten eines Klassentyps, legt die Klasse fest, wie das Objekt initialisiert wird, wenn kein expliziter Initialwert angegeben wird. Zum Beispiel haben wir in Abschnitt 1.1 auf Seite 27 festgestellt, dass wir einen `string` nicht explizit initialisieren müssen, denn `string` wird implizit als leerer String initialisiert. Für lokale Variablen von Built-in-Datentypen wird keine solche Initialisierung durchgeführt.

Lokale Variablen von Built-in-Datentypen, die nicht explizit initialisiert werden, sind ***undefiniert***, was bedeutet, dass der Wert der Variablen aus dem zufälligen Datenmüll besteht, der sich bereits in dem Stück Speicher befand, in dem die Variable angelegt wird. Es ist illegal, irgend etwas anderes mit einem undefinierten Wert zu tun, als ihn mit einem gültigen Wert zu überschreiben. Viele Implementierungen entdecken die Verletzung dieser Regel nicht und erlauben den Zugriff auf undefinierte Werte. Das Ergebnis ist fast immer ein Absturz oder ein fehlerhaftes Resultat, denn was auch immer in dem Moment im Speicher stand, ist fast niemals ein korrekter Wert; oftmals ist es sogar ein Wert, der für den angegebenen Typ ungültig ist.

Hätten wir `summe` oder `anzahl` keinen Initialwert gegeben, wäre unser Programm sehr wahrscheinlich fehlgeschlagen. Der Grund dafür ist, dass das Programm als Erstes die Werte dieser Variablen liest: Das Programm liest `anzahl`, um sie zu inkrementieren und es liest `summe`, um den gerade eingelesenen Wert aufzuaddieren. Aus ähnlichem Grund geben wir `x` keinen Initialwert, denn das erste, was wir damit machen, ist das Einlesen eines Wertes, so dass wir jeglichen Initialwert unmittelbar überschreiben würden.

Der einzige neue Aspekt in unserer `while`-Anweisung ist die Form ihrer Bedingung:

```
// Invariante:
//    Wir haben anzahl Noten eingelesen und
//    summe ist die Summe der ersten anzahl Noten.
while(cin >> x) {
    ++anzahl;
    summe += x;
}
```

Wir wissen bereits, dass die `while`-Anweisung so lange ausgeführt wird, wie die Bedingung `cin >> x` den Wahrheitswert `true` liefert. Wir werden die Details der Behandlung von `cin >> x` als Bedingung in Abschnitt 3.1.1 auf Seite 62 untersuchen, aber zum momentanen Verständnis können wir vorgreifen und feststellen, dass die Bedingung wahr wird, wenn der letzte Einleseversuch (also `cin >> x`) gelang.

Innerhalb des `while` benutzen wir Inkrementierungs- und Compound-Assignment-Operatoren, die wir bereits in Kapitel 2 benutzt haben. Aus der Diskussion dort wissen wir, dass `++count` `1` zu `count` addiert und dass `summe += x` `x` zu `summe` addiert.

Alles, was nunmehr noch übrig ist, ist die Ausgabe des Programmes zu erklären:

```
streamsize prec = cout.precision();
cout << "Ihre Endnote ist " << setprecision(3)
     << 0.2 * vor + 0.4 * diplom + 0.4 * summe / anzahl
     << setprecision(prec) << endl;
```

Unser Ziel ist es, die Endnote mit drei signifikanten Ziffern darzustellen, was wir mit Hilfe von `setprecision` erreichen. Wie auch `endl`, ist `setprecision` ein Manipulator. Es manipuliert den Stream, so dass nachfolgende Ausgaben auf diesem Stream mit der entsprechenden Anzahl signifikanter Ziffern vorgenommen werden. Mit `setprecision(3)`

sorgen wir dafür, dass die Implementierung die Noten mit drei signifikanten Ziffern ausgibt, im Allgemeinen eine vor dem Dezimalpunkt und zwei danach.

Indem wir `setprecision` benutzen, verändern wir die Genauigkeit der Ausgabe für alle nachfolgenden Ausgaben auf dem Stream `cout`. Da diese Anweisung am Ende des Programms vorkommt, wissen wir, dass es solche nachfolgenden Ausgaben nicht geben wird. Nichtsdestotrotz glauben wir, dass es sinnvoll ist, die Genauigkeit von `cout` auf den vor der Änderung gesetzten Wert zurückzusetzen. Wir machen dies, indem wir eine Memberfunktion (Abschnitt 1.2 auf Seite 31) von `cout` mit dem Namen `precision` aufrufen. Diese Funktion gibt uns die Genauigkeit eines Streams für Fließkommaausgaben zurück. Wir verwenden `setprecision`, um diese Genauigkeit auf `3` zu setzen, schreiben die Endnote und setzen die Genauigkeit dann auf den Wert zurück, den wir von `precision` erhalten haben. Der Ausdruck, der die Endnote berechnet, benutzt mehrere arithmetische Operatoren: `*` zur Multiplikation, `/` zur Division und `+` zur Addition.

Wir hätten die Memberfunktion `precision` benutzen können, um die Genauigkeit zu setzen:

```
// Genauigkeit auf 3 setzen, vorherigen Wert zurückliefern
streamsize prec = cout.precision(3);

cout << "Ihre Endnote ist: "
     << 0.2 * vor + 0.4 * diplom + 0.4 * summe / anzahl
     << endl;

// Genauigkeit zurücksetzen auf ursprünglichen Wert:
cout.precision(prec);
```

Wir bevorzugen jedoch die Benutzung des Manipulators `setprecision`, da wir auf diese Weise den Teil des Programms, in dem die Genauigkeit auf einen ungewöhnlichen Wert gesetzt wird, so klein wie möglich halten können.

3.1.1 Testen auf Eingabeende

Konzeptionell ist nur die Bedingung der `while`-Anweisung neu. Diese Bedingung benutzt implizit einen `istream` als Subjekt der `while`-Bedingung:

```
while (cin >> x) { /*...*/ }
```

Diese Anweisung versucht, von `cin` einzulesen. Wenn diese Leseoperation gelingt, erhält `x` den Wert, den wir gerade eingelesen haben, und außerdem liefert der Bedingungstest des `while` den Wahrheitswert `true`. Wenn das Einlesen fehlschlägt (zum Beispiel weil wir keine Eingabedaten mehr haben oder weil wir Eingaben erhalten, die für den Datentyp von `x` unangemessen sind), dann liefert der Bedingungstest des `while` den Wahrheitswert `false`, und wir können uns auf den Wert von `x` nicht verlassen.

Die Funktionsweise dieses Codes ist nicht leicht zu verstehen. Wir können uns zunächst vorstellen, dass der Operator `>>` seinen linken Operanden zurückgibt, so dass die Frage

nach dem Wert von `cin >> x` gleichbedeutend ist mit der Ausführung von `cin >> x` und der anschließenden Frage nach dem Wert von `cin`. Zum Beispiel können wir einen einzelnen Wert enlesen und anschließend nach dem Erfolg dieses Einlesens fragen, indem wir

```
if (cin >> x) { /* ... */ }
```

ausführen.

Diese Anweisung hat dieselbe Bedeutung wie

```
cin >> x;
if (cin) { /* ... */ }
```

Wenn wir `cin >> x` als eine Bedingung verwenden, dann testen wir nicht einfach nur die Bedingung; wir lesen als Seiteneffekt ebenfalls einen Wert in `x` ein. Nunmehr müssen wir nur noch herausfinden, wie man `cin` als Bedingung in einer `while`-Anweisung verwendet.

Da `cin` den Datentyp `istream` hat, der Teil der Standardbibliothek ist, müssen wir die Definition von `istream` zu Rate ziehen, um die Bedeutung von `if (cin)` und `while (cin)` zu verstehen. Die Details dieser Definition sind jedoch so kompliziert, dass wir sie bis Abschnitt 12.5 auf Seite 292 zurückstellen. Aber auch ohne diese Details können wir einen guten Teil der Abläufe verstehen.

Die Bedingungen, die wir in Kapitel 2 verwendet haben, benutzten alle relationale Operatoren, die direkt Werte vom Typ `bool` erzeugen. Weiterhin können wir Ausdrücke benutzen, die arithmetische Werte ergeben. Wenn wir in einer Bedingung arithmetische Werte benutzen, entstehen daraus Werte vom Typ `bool`: Nicht-Null-Werte werden zu `true`, Null-Werte zu `false` konvertiert. Für den Moment ist alles, was wir wissen müssen, dass `istream` eine Funktionalität anbietet, um `cin` in einen Wert zu konvertieren, der in einer Bedingung benutzt werden kann. Wir wissen noch nicht, welchen Typ dieser Wert hat, aber wir wissen genau, dass dieser Typ in einen `bool` konvertiert werden kann. Daher wissen wir nunmehr auch, dass dieser Wert in einer Bedingung genutzt werden kann. Der Wert, den diese Konvertierung ergibt, hängt vom internen Zustand des `istream` ab, der wiederum davon abhängt, ob der letzte Leseversuch erfolgreich war. Daher ist die Benutzung von `cin` als Bedingung einfach ein anderer Weg um festzustellen, ob der letzte Leseversuch von `cin` erfolgreich war.

Gründe, warum ein Leseversuch von einem Stream fehlschlagen könnte, sind:

- Wir haben das Ende der Eingabedatei erreicht.
- Wir sind auf Eingabedaten gestoßen, die inkompatibel zum Typ der Variablen sind, in die wir einlesen wollen. Dies kann zum Beispiel passieren, wenn wir in eine `int`-Variable einlesen wollten und ein Zeichen vorfinden, das keine Ziffer ist.
- Das System hat ein Hardwareproblem auf dem Eingabegerät entdeckt.

In jedem dieser Fälle ist das Resultat das gleiche: Die Benutzung eines Eingabestreams als Bedingung wird den Wert `false` liefern. Darüberhinaus werden danach alle weiteren Versuche, von diesem Stream zu lesen, fehlschlagen, bis wir den Stream zurücksetzen, worüber wir in Abschnitt 4.1.3 auf Seite 84 mehr lernen werden.

3.1.2 Die Schleifeninvariante

Das Verständnis der Schleifeninvariante in Abschnitt 2.3.2 auf Seite 39 erfordert spezielle Aufmerksamkeit, denn die Bedingung in der `while`-Anweisung hat Seiteneffekte. Diese Seiteneffekte beeinflussen den Wahrheitsgehalt der Invariante: Die erfolgreiche Ausführung von `cin >> x` macht den ersten Teil der Invariante – den Teil der aussagt, dass wir `anzahl` Noten eingelesen haben – falsch. Entsprechend müssen wir unsere Analyse ändern, um den Effekt, den die Bedingung auf die Invariante hat, mit einzubeziehen.

Wir wissen, dass die Invariante vor der Auswertung der Bedingung wahr war, also wissen wir, dass wir bereits `anzahl` Noten eingelesen haben. Wenn `cin >> x` gelingt, dann haben wir `anzahl + 1` Noten eingelesen. Wir können diesen Teil der Invariante erneut wahr werden lassen, indem wir `anzahl` inkrementieren. Dadurch jedoch wird der zweite Teil der Invariante falsch – der Teil, der aussagt, dass `summe` die Summe der ersten `anzahl` Noten ist – denn wir haben `anzahl` inkrementiert, wodurch `summe` die Summe der ersten `anzahl - 1` Noten ist. Glücklicherweise können wir dafür sorgen, dass der zweite Teil der Invariante erneut wahr wird, indem wir `sum += x` ausführen; damit wird die gesamte Invariante für nachfolgende Durchläufe durch das `while` wahr.

Wenn die Bedingung falsch ist, bedeutet das, dass unser Versuch, eine Eingabe zu lesen, fehlgeschlagen ist, wir haben also keine weiteren Daten erhalten und die Invariante ist nach wie vor wahr. Im Ergebnis müssen wir uns um die Seiteneffekte der Bedingung nicht kümmern, wenn die `while`-Schleife endet.

3.2 Median der Hausarbeitsnoten

Das Programm, das wir bisher geschrieben haben, hat eine Designschwäche: Es wirft jede Hausarbeitsnote weg, sobald es sie gelesen hat. Dies genügt für die Berechnung des Durchschnitts, aber was passiert, wenn wir anstelle dessen den Median der Hausarbeitsnoten berechnen wollen?

Der einfachste Weg zur Berechnung des Medians einer Reihe von Werten ist, die Werte zunächst in aufsteigender (oder absteigender) Ordnung zu sortieren und dann den mittleren Wert – oder, wenn die Anzahl der Werte gerade ist, den Durchschnitt der mittleren beiden Werte – zu verwenden. Der Median ist häufig aussagekräftiger als der Durchschnitt, denn obwohl er ebenfalls eine Aussage über fortdauernde Leistungen macht, verhindert er, dass ein paar schlechte Noten auf die Gesamtleistung Einfluss nehmen.

Nach kurzem Nachdenken stellen wir fest, dass wir für die Berechnung des Medians unser Programm grundlegend ändern müssen. Um den Median einer unbekannten Anzahl von Werten zu berechnen, müssen wir alle Werte speichern. Damit wir den Durchschnitt finden, genügt es dagegen, nur jeweils die Anzahl und Summe der gelesenen Noten zu speichern, denn der Durchschnitt wird lediglich aus der Summe, dividiert durch die Anzahl der Noten, berechnet.

3.2.1 Speicherung von Datensammlungen in einem `vector`

Um den Median zu berechnen, müssen wir alle Hausarbeitsnoten speichern, sortieren und schließlich den (oder die beiden) mittleren heraussuchen. Um diese Berechnung bequem und effizient zu erledigen, brauchen wir einen Weg, um

- eine Anzahl von Werten zu speichern, die wir einen nach dem anderen einlesen, ohne im Voraus zu wissen, wie viele Werte wir lesen werden,
- die Werte zu sortieren, nachdem wir alle eingelesen haben,
- an den/die mittleren Wert(e) effizient heranzukommen.

Die Standardbibliothek bietet einen Typ namens **`vector`** an, den wir benutzen können, um all diese Probleme auf einmal zu lösen. Ein `vector` speichert eine Sequenz von Werten eines bestimmten Typs, wächst nach Bedarf, um weitere Werte aufzunehmen, und lässt uns auf jeden einzelnen Wert effizient zugreifen.

Wir wollen nun unser Benotungsprogramm umschreiben, indem wir die Noten in einem `vector` sammeln, anstelle die Summe sofort zu berechnen und die Noten zu verwerfen. Dies betrifft den folgenden Abschnitt des Codes:

```
// originale Programmversion (Ausschnitt):
int anzahl = 0;
double summe = 0;
double x;

// Invariante:
//    Wir haben anzahl Noten eingelesen und
//    summe ist die Summe der ersten anzahl Noten.
while(cin >> x) {
    ++anzahl;
    summe += x;
}
```

Diese Schleife speicherte, wieviele Noten gelesen wurden und die Summe ihrer Werte. Das Bedürfnis, diese beiden Werte synchron zu den neu eingelesenen Noten zu halten, machte die Schleifeninvariante relativ kompliziert. Im Gegensatz dazu erhalten wir eine wesentlich einfachere Invariante, wenn wir einen `vector` verwenden, um die Werte zu speichern:

```
// revidierte Version des Ausschnittes:
double x;
vector<double> hausarbeiten;

// Invariante: hausarbeiten enthält alle Hausarbeitsnoten
// bis jetzt
while (cin >> x)
    hausarbeiten.push_back(x);
```

Wir haben die Grundstruktur unseres Codes nicht geändert: Er liest immer noch einen Wert nach dem anderen in die Variable `x` ein, bis das Datei-Ende-Zeichen oder eine fehlerhafte Eingabe gelesen wird. Der Unterschied liegt darin, was wir mit diesen Werten tun.

Beginnen wir mit `hausarbeiten`, definiert mit dem Typ `vector<double>`. Ein `vector` ist ein ***Container***, der eine Sammlung von Werten enthält. Alle Inhalte eines bestimmten `vectors` sind vom selben Typ, aber verschiedene `vectors` können Objekte unterschiedlicher Typen enthalten. Wenn wir einen `vector` definieren, müssen wir den Typ der im `vector` enthaltenen Werte angeben. Unsere Definition von `hausarbeiten` sagt aus, dass es sich um einen `vector` handelt, der `double`-Werte speichert.

Der Typ `vector` ist durch ein Sprachfeature namens ***Template-Klasse*** definiert. Wir werden in Kapitel 11 sehen, wie eine solche Template-Klasse definiert wird. Im Moment ist es wichtig zu verstehen, dass wir die Funktionsweise von `vector` unabhängig vom Typ der Objekte, die in dem `vector` enthalten sind, betrachten können. Wir spezifizieren diesen Typ der Objekte in Winkelklammern. Zum Beispiel sind Objekte vom Typ `vector<double>` Instanzen von `vector`, die Objekte vom Typ `double` speichern, Objekte des Typs `vector<string>` speichern Objekte des Typs `string` und so weiter.

Die `while`-Schleife liest Werte von der Standardeingabe ein und speichert sie im `vector`. Wie bereits zuvor, lesen wir in `x` ein, bis wir das Datei-Ende-Zeichen oder eine Eingabe lesen, die kein `double` ist. Neu ist:

```
hausarbeiten.push_back(x);
```

Wie auch `gruss.size()` in Abschnitt 1.2 auf Seite 31, handelt es sich bei `push_back` um eine Memberfunktion, die als Teil des `vector`-Typs definiert ist, und die wir an dem Objekt namens `hausarbeiten` arbeiten lassen. Wir rufen die Funktion auf und übergeben `x`. Die Memberfunktion `push_back` hängt daraufhin ein neues Element an das Ende des `vectors` an. Der Name `push_back` deutet bereits darauf hin, dass etwas in den `vector` gelegt wird (*push*), und zwar nach hinten (*back*), also an das Ende. Als Seiteneffekt erhöht die Funktion die Größe des `vectors` um eins.

Da die Funktion `push_back` gut zu der von uns gesuchten Funktionalität passt, ist es trivial zu sehen, dass ihr Aufruf die von uns gewählte Schleifeninvariante stützt. Daher ist es klar, dass wir alle Hausarbeitsnoten eingelesen und in `hausarbeiten` gespeichert haben, wenn wir die `while`-Schleife verlassen.

Als Nächstes müssen wir uns über die Ausgabe Gedanken machen.

3.2.2 Erstellung der Ausgabe

In der originalen Version des Programms aus Abschnitt 3.1 auf Seite 57 haben wir die Endnote innerhalb der Ausgabeanweisung selbst berechnet:

```
streamsize prec = cout.precision();
cout << "Ihre Endnote ist " << setprecision(3)
     << 0.2 * vor + 0.4 * diplom + 0.4 * summe / anzahl
     << setprecision(prec) << endl;
```

wobei `diplom` und `vor` die Prüfungsnoten enthielten und `summe` und `anzahl` jeweils die Summe und Anzahl der eingegebenen Hausarbeitsnoten darstellten.

Wie wir in Abschnitt 3.2.1 auf Seite 65 angemerkt haben, ist der einfachste Weg der Medianberechnung, die Daten zu sortieren und danach den mittleren Wert zu finden oder den Durchschnitt der beiden mittleren Werte zu berechnen, falls wir eine gerade Anzahl von Datenelementen haben. Wir können die Berechnung leichter verständlich machen, wenn wir die Berechnung des Medians vom Ausgabecode trennen.

Um den Median zu finden, benötigen wir die Größe des `vectors` `hausarbeiten` mindestens zweimal: einmal, um festzustellen, ob die Größe Null ist, und erneut, um den Index des mittleren Elements bzw. der mittleren Elemente zu berechnen.

Um die Größe nicht zweimal abfragen zu müssen, werden wir sie in einer lokalen Variable speichern:

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

Der Typ `vector` definiert einen Datentyp namens `vector<double>::size_type` und eine Funktion namens `size`. Diese Member funktionieren genauso wie die gleichnamigen Member in `string`: Der Typ, den `size_type` festlegt, ist ein `unsigned`-Datentyp, der die Größe des längstmöglichen `vectors` aufnehmen kann, und `size()` gibt einen Wert vom Typ `size_type` zurück, der die Anzahl der Elemente im `vector` angibt.

Da wir diese Größe an zwei Stellen verwenden müssen, legen wir den Wert in einer lokalen Variable ab. Verschiedene Implementierungen verwenden verschiedene Datentypen, um Größen darzustellen, also können wir den angemessenen Typ nicht direkt angeben und gleichzeitig implementierungsunabhängig bleiben. Aus diesem Grund ist es gute Programmierpraxis, den `size_type` zu verwenden, den die Bibliothek für die Angabe von Containergrößen definiert, was wir auch bei der Benennung des Typs von `groesse` tun.

In diesem Beispiel ist dieser Typ sehr umständlich zu schreiben – und zu lesen. Um unser Programm zu vereinfachen, haben wir daher ein Sprachmittel benutzt, dass wir noch nie vorher gesehen haben: einen `typedef`. Wenn wir den Begriff `typedef` als Teil einer Definition angeben, sagen wir, dass wir den Namen, den wir definieren, als Synonym für den angegebenen Typ verwenden wollen, und nicht zum Beispiel als Variable dieses Typs. Da unserere Definition `typedef` enthält, definiert sie den Namen `vec_sz` als ein Synonym für `vector<double>::size_type`. Namen, die mit Hilfe von `typedef` definiert werden, haben

denselben Gültigkeitsbereich wie alle anderen Namen. Das bedeutet, wir können den Namen `vec_sz` als Synonym für den `size_type` benutzen, bis der aktuelle Gültigkeitsbereich endet.

Nachdem wir nun geklärt haben, wie man den Typ des Rückgabewertes von `hausarbeiten.size()` benennt, können wir diesen Wert in einer lokalen Variable, genannt `groesse`, desselben Typs speichern. In manchen Programmierprojekten – zum Beispiel innerhalb multinationaler Firmen oder Teams – ist es üblich, englischsprachige Variablennamen zu verwenden. In diesem Fall ist der Programmierer versucht, die Variable, die wir mit dem Namen `groesse` versehen haben, `size` zu nennen. Könnte es in diesem Fall einen Konflikt mit dem Namen `size` der Memberfunktion geben? Dies ist nicht der Fall: Wollen wir die Memberfunktion `size` aufrufen, um die Größe eines `vectors` zu bestimmen, so müssen wir `size` rechts von einem Punkt schreiben, auf dessen linker Seite ein Variablenname des Typs `vector` steht. Mit anderen Worten ausgedrückt: `size` definiert als lokale Variable liegt in einem anderen Gültigkeitsbereich als `size` als Operation auf einem `vector`. Da diese Namen in unterschiedlichen Gültigkeitsbereichen verwaltet werden, weiß der Compiler (und auch der Programmierer) wie `size` jeweils interpretiert werden muss.

Da es unsinnig ist, den Median einer leeren Datenmenge zu bestimmen, müssen wir als Nächstes prüfen, ob wir Daten eingelesen haben:

```
if(groesse == 0) {
    cout << endl << "Sie müssen die Noten eingeben. "
                    "Bitte versuchen Sie es noch einmal."
         << endl;
    return 1;
}
```

Wir können dies daran erkennen, ob `groesse` den Wert Null hat. In diesem Fall beenden wir das Programm, indem wir `1` zurückliefern, um einen fehlerhaften Ablauf zu signalisieren. Bereits in Kapitel 0 haben wir festgestellt, dass das System einen Rückgabewert von `main` mit dem Wert `0` als erfolgreichen Programmablauf interpretiert. Die Rückgabe eines beliebigen anderen Wertes hat eine systemabhängige Bedeutung, aber die meisten Systeme behandeln alle Nicht-Null-Werte als fehlerhaften Programmlauf.

Nachdem wir nunmehr geprüft haben, dass wir Daten eingelesen haben, können wir mit der Berechnung des Medians beginnen. Als Erstes sortieren wir die Daten, indem wir eine Bibliotheksfunktion aufrufen:

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

Die Funktion `sort`, definiert im Header `<algorithm>`, ordnet die Inhalte eines Containers in nicht-absteigender Ordnung. Wir sagen „nicht-absteigend“ anstelle von „aufsteigend“, da mehrere Elemente den gleichen Wert haben können.

Die Argumente für `sort` geben den Bereich an, der zu sortieren ist. Die `vector`-Klasse bietet zwei Memberfunktionen, `begin` und `end`, dafür an. In Abschnitt 5.2.2 auf Seite 114 werden wir weiteres über `begin` und `end` sagen, aber vorerst ist nur wichtig, dass `hausarbeiten.begin()` das erste Element in dem `vector` `hausarbeiten`, und `hausarbeiten.end()` das letzte (eigentlich „eins hinter dem letzten“) Element in `hausarbeiten` angibt.

Die Funktion `sort` bearbeitet dabei die Elemente an Ort und Stelle: Es bewegt die Elemente des Containers anstelle einen neuen Container für das Ergebnis zu erzeugen.

Nachdem wir `hausarbeiten` sortiert haben, müssen wir das mittlere Element oder die mittleren Elemente finden:

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

Wir dividieren zunächst `groesse` durch 2, um das mittlere Element in `vector` zu finden. Wenn die Anzahl der Elemente gerade ist, stimmt das Ergebnis der Division exakt. Wenn die Anzahl ungerade ist, ist das Ergebnis die nächstkleinere Ganzzahl.

Wie wir den Median genau berechnen, hängt davon ab, ob die Anzahl der Elemente gerade oder ungerade ist. Ist sie gerade, ist der Median der Mittelwert der mittleren beiden Elemente. Anderenfalls gibt es genau ein mittleres Element, und dies ist gleichzeitig der Median.

Der Ausdruck, der den Wert an `median` zuweist, benutzt zwei neue Operatoren: den *Modulooperator* `%` und den *Bedingungsoperator* (in deutscher Fachliteratur auch *arithmetisches if* genannt – Anmerkung des Übersetzers) – den `?:`-Operator.

Der Modulooperator `%` gibt den Rest der Division des linken durch den rechten Operanden zurück. Wenn der Rest nach der Division durch `2` `0` ist, dann hat das Programm eine gerade Anzahl von Elementen eingelesen.

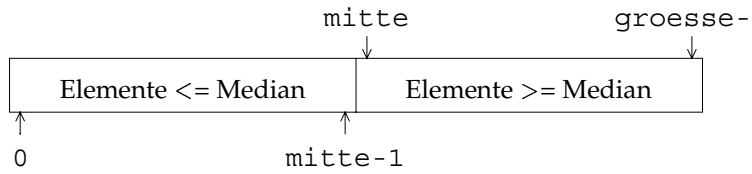
Der Bedingungsoperator ist ein Kürzel für eine einfache if-then-else–Anweisung. Zuerst wertet er den Ausdruck, der dem `?` vorausgeht, `size%2==0`, als Bedingung aus, um einen `bool`-Wert zu erhalten. Wenn diese Bedingung `true` ergibt, dann ist das Ergebnis der Operation der Wert des Ausdrucks zwischen `?` und dem darauf folgenden `:`; anderenfalls ist das Ergebnis der Wert des Ausdrucks nach dem `:`. Falls wir also eine gerade Anzahl von Werten eingelesen haben, setzen wir `median` auf den Mittelwert der beiden mittleren Elemente. Haben wir eine ungerade Anzahl eingelesen, dann setzen wir `median` auf `hausarbeiten[mitte]`. Analog zu `&&` und `||` wertet der `?:`-Operator zunächst den am weitesten links stehenden Operanden aus. Ausgehend von dessen Wert wertet er dann genau einen der beiden anderen Operanden aus.

Ein Weg, um auf die Elemente eines `vectors` zuzugreifen, wird durch die Benutzung von `hausarbeiten[mitte]` und `hausarbeiten[mitte-1]` dargestellt. Jedes Element eines `vectors` ist mit einer Ganzzahl, ***Index*** genannt, verbunden. Zum Beispiel ist `hausarbeiten[mitte]` das Element von `hausarbeiten` mit dem Index `mitte`. Wie man aus Abschnitt 2.6 auf Seite 51 schließen kann, ist das erste Element von `hausarbeiten` `hausarbeiten[0]` und das letzte Element ist `hausarbeiten[groesse-1]`.

Jedes Element ist ein (unbenanntes) Objekt des Typs, der im Container gespeichert werden kann. Somit ist `hausarbeiten[mitte]` ein Objekt des Typs `double`, worauf wir alle Operationen anwenden können, die der Typ `double` unterstützt. Beispielsweise können

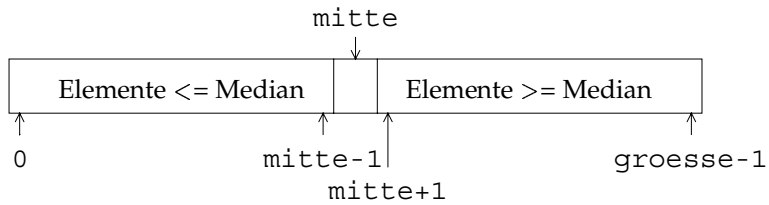
wir zwei Elemente addieren, und wir können die entstehende Summe durch 2 dividieren, um den Mittelwert dieser beiden Objekte zu erhalten.

Nachdem wir wissen, wie wir auf Elemente von hausarbeiten zugreifen können, ist es leicht zu erkennen, wie die Medianberechnung funktioniert. Nehmen wir zunächst an, dass groesse gerade ist, so dass mitte exakt size/2 ist. Dann müssen sich genau mitte Elemente von hausarbeiten auf jeder Seite der Mitte befinden:



Weil wir wissen, dass die Hälfte von hausarbeiten genau mitte Elemente enthält, sollte es einfach zu sehen sein, dass die Indizes der beiden Elemente die der Mitte am nächsten liegen mitte-1 und mitte sind; der Median ist der Mittelwert dieser beiden Elemente.

Wenn die Anzahl der Elemente ungerade ist, dann ist mitte exakt (groesse-1) / 2, da die Kommastelle abgeschnitten wird. In diesem Fall können wir uns unseren sortierten hausarbeiten-vector als zwei Segmente aus jeweils mitte Elementen vorstellen, die durch ein einzelnes Element in der Mitte getrennt sind. Dieses Element ist der Median:



Unsere Berechnung setzt auf die Fähigkeit, auf die Elemente eines vectors anhand ihres Indexes zuzugreifen.

Nachdem wir den Median berechnet haben, müssen wir nur noch die Endnote berechnen und ausgeben:

```
streamsize genau = cout.precision();
cout << "Ihre Endnote ist " << setprecision(3)
     << 0.2 * vor + 0.4 * diplom + 0.4 * median
     << setprecision(genau) << endl;
```

Das resultierende Programm ist nicht komplizierter als das in Abschnitt 3.1 auf Seite 57, obwohl es wesentlich mehr Arbeit leistet. Insbesondere wächst unser vector hausarbeiten bei der Eingabe der Hausarbeitsnoten nach Bedarf, ohne dass wir uns um den dafür erforderlichen Speicher kümmern müssen. Die Standardbibliothek erledigt das für uns.

Hier ist das gesamte Programm. Die einzigen Teile, die wir noch nicht erwähnt haben, sind die `#include` - Direktiven, die dazugehörigen `using`-Deklarationen und einige weitere Kommentare:

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

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

int main()
{
    // Nach dem Namen fragen und einlesen.
    cout << "Bitte geben Sie Ihren Vornamen ein: ";
    string name;
    cin >> name;
    cout << "Hallo, "<< name << "!" << endl;

    // Nach den Prüfungsnoten fragen und einlesen.
    cout << "Bitte geben Sie die Vordiplom- und "
            "Diplomprüfungsnoten ein: ";
    double vordiplom, diplom;
    cin >> vordiplom >> diplom;

    // Nach den Hausarbeitsnoten fragen.
    cout << "Geben Sie alle Hausarbeitsnoten ein, "
            "danach das Datei-Ende-Zeichen: ";
    vector<double> hausarbeiten;
    double x;

    // Invariante: hausarbeiten enthält alle
    // Hausarbeitsnoten
    while (cin >> x)
            hausarbeiten.push_back(x);

    // prüfen, ob Hausarbeitsnoten eingegeben wurden
    typedef vector<double>::size_type vec_sz;
    vec_sz groesse = hausarbeiten.size();
    if(groesse == 0) {
        cout << endl << "Sie müssen die Noten eingeben. "
                        "Bitte versuchen Sie es noch einmal."
                     << endl;
        return 1;
    }
```

```
    // Noten sortieren
    sort(hausarbeiten.begin(), hausarbeiten.end());

    // Median der Hausarbeitsnoten berechnen
    vec_sz mitte = groesse/2;
    double median;
    median = groesse % 2 == 0 ? (hausarbeiten[mitte]
                                 + hausarbeiten[mitte-1])/2
                              : hausarbeiten[mitte];

    // Endnote berechnen und ausgeben
    streamsize genau = cout.precision();
    cout << "Ihre Endnote ist " << setprecision(3)
         << 0.2 * vor + 0.4 * diplom + 0.4 * median
         << setprecision(genau) << endl;
    return 0;
}
```

Einige weitere Beobachtungen

Dieser Code enthält einige Punkte, die unsere besondere Aufmerksamkeit verdienen. Zunächst gibt es einige Gründe mehr, warum wir das Programm beenden, wenn `hausarbeiten` leer ist. Logisch betrachtet ist der Median einer leeren Menge undefiniert – wir haben keine Vorstellung davon. Daher ist das Abbrechen des Programmes eine gute Idee: Wenn wir nicht wissen, was wir tun sollen, können wir genauso gut die Abarbeitung beenden. Aber es ist wichtig zu wissen, was passiert, wenn wir die Ausführung fortsetzen würden. Wenn die Eingabe leer wäre und wir vergessen würden zu prüfen, ob wir nicht wenigstens einen Wert eingelesen hätten, würde der Code zur Berechnung des Medians fehlschlagen. Warum?

Wenn wir keine Elemente eingelesen hätten, wäre `hausarbeiten.size()`, und damit `groesse`, `0`. Wenn wir `hausarbeiten[mitte]` ausführen, betrachten wir das erste Element von `hausarbeiten`. Aber es gibt keine Elemente in `hausarbeiten`! Wenn wir `hausarbeiten[0]` ausführen, ist vollkommen unklar, was passieren wird. Variablen vom Typ `vector` prüfen nicht, ob der Index innerhalb des zulässigen Bereiches ist. Diese Prüfung muss der Benutzer selbst durchführen.

Die nächste wichtige Beobachtung ist, dass `vector<double>::size_type`, wie alle Größentypen der Standardbibliothek, ein ***vorzeichenloser Ganzzahltyp*** ist. Solche Typen können keine negativen Werte aufnehmen; anstelle dessen speichern sie die Werte modulo 2^*n*, wobei *n* von der Implementation abhängt. Es wäre daher auch unsinnig zu prüfen, ob `hausarbeiten.size() < 0`, denn dieser Vergleich würde immer `false` ergeben.

Immer wenn normale Integer und vorzeichenlose Integer in einem Ausdruck zusammen benutzt werden, wird der normale Integer in einen vorzeichenlosen Integer konvertiert. Infolgedessen ergeben Ausdrücke wie `hausarbeiten.size() - 100` vorzeichenlose Ergebnisse, was bedeutet dass sie ebenfalls nicht kleiner als Null werden können, auch wenn `hausarbeiten.size() < 100`.

Letztendlich sollten wir noch wissen, dass die Performance unseres Programmes sehr gut ist, auch wenn der `vector<double>` nach Bedarf wächst, anstatt sofort mit der richtigen Größe angelegt zu werden.

Wir können Vertrauen zur Performance unseres Programms haben, da der C++-Standard bestimmte Ansprüche an die Leistungsfähigkeit der Implementierung stellt. Die Bibliothek muss nicht nur den Verhaltensspezifikationen folgen, sondern sie muss ebenfalls wohldefinierte Leistungsmerkmale erfüllen. Jede standardkonforme C++-Implementierung muss

- `vector` so implementieren, dass das Anhängen einer großen Anzahl von Elementen an einen `vector` einen zur Anzahl der Elemente proportionalen Aufwand erfordert,
- `sort` so implementieren, dass es nicht langsamer als *n* log *n* ist, wobei *n* die Anzahl der zu sortierenden Elemente ist.

Das gesamte Programm läuft daher garantiert in einer Zeit von *n* log *n* oder besser auf jeder standardkonformen Implementierung. Tatsächlich wurde die Standardbibliothek mit einem sehr großen Interesse an Performance designed. C++ ist für den Einsatz in performance-kritischen Anwendungen vorgesehen, daher wird auch in der Bibliothek viel Wert auf Geschwindigkeit gelegt.

3.3 Details

Lokale Variablen

werden default-initialisiert, wenn sie nicht mit einem expliziten Initialwert versehen werden. Defaultinitialisierung eines Built-in-Datentyps bedeutet, dass der Wert undefiniert ist. Undefinierte Werte sollen nur auf der linken Seite einer Zuweisung benutzt werden.

Typdefinitionen:

`typedef Typ Name;` Definiert *`Name`* als Synonym für *`Typ`*.

Der Datentyp vector,

definiert in `<vector>`, ist ein Bibliotheksdatentyp, der einen Container für eine Sequenz von Werten eines angegebenen Typs darstellt. `vectors` wachsen dynamisch. Einige wichtige Operationen sind:

`vector<T>::size_type`
: Ein Typ, der die Anzahl der Elemente im größtmöglichen `vector` aufnehmen kann.

`v.begin()`
: Gibt einen Wert zurück, der das erste Element in `v` bezeichnet.

`v.end()`
: Gibt einen Wert zurück, der das letzte Element (den Platz hinter dem letzten Element) bezeichnet.

`vector<T> v;`
: Erzeugt einen leeren `vector`, der Elemente des Typs `T` aufnehmen kann.

`v.push_back(e)`
: Fügt an den `vector` ein neues Element mit dem Wert `e` an.

`v[i]`
: Gibt den an der Position `i` gespeicherten Wert zurück.

`v.size()`
: Gibt die Anzahl der Elemente in `v` zurück.

Andere Einrichtungen der Standardbibliothek

`sort(b, e)`
: Ordnet die Elemente im Bereich [*b*, *e*) in nicht-absteigender Ordnung. Definiert in `<algorithm>`.

`max(e1, e2)`
: Gibt den größeren der Ausdrücke `e1` und `e2` zurück; `e1` und `e2` müssen genau denselben Typ haben. Definiert in `<algorithm>`.

`while (cin>>x)`
: Liest einen Wert eines angemessenen Typs in `x` ein und testet den Zustand des Datenstroms. Wenn der Strom in einem Fehlerzustand ist, schlägt der Test fehl; anderenfalls ist der Test erfolgreich und der Körper des `while` wird ausgeführt.

`s.precision(n)`
: Setzt die Genauigkeit des Datenstroms `s` auf `n` für künftige Ausgaben (oder verändert sie nicht, falls `n` weggelassen wird). Gibt die vorherige Genauigkeit zurück.

`setprecision(n)`
: Gibt einen Wert zurück, den man auf einen Ausgabedatenstrom `s` schreiben kann, so dass `s.precision(n)` aufgerufen wird. Definiert in `<iomanip>`.

`streamsize`
: Datentyp des Wertes, der von `setprecision` erwartet und von `precision` zurückgegeben wird. Definiert in `<ios>`.

ÜBUNGEN

3-0. Compilieren, starten und testen Sie die Programme in diesem Kapitel.

3-1. Stellen Sie sich vor, Sie wollen den Median einer Reihe von Werten finden. Nehmen Sie an, dass wir einige der Werte bereits eingelesen haben und dass wir nicht wissen, wie viele Werte wir noch einlesen müssen. Beweisen Sie, dass wir es uns nicht erlauben können, bereits gelesene Werte zu verwerfen. *Hinweis:* Eine Beweisstrategie ist es, anzunehmen, dass wir einen Wert verwerfen könnten, und dann Werte für den ungelesenen und daher unbekannten Teil unserer Reihe zu finden, die dazu führen würden, dass der Median genau der von uns verworfene Wert wird.

3-2. Schreiben Sie ein Programm, das die Quartile einer Menge Integer (ein Viertel der Zahlen mit dem höchsten Wert, das nächsthöhere Viertel und so weiter) berechnet und ausgibt.

3-3. Schreiben Sie ein Programm, das zählt, wie oft jedes einzelne Wort in seiner Eingabe vorkommt.

3-4. Schreiben Sie ein Programm, das die Länge des längsten und kürzesten `string`s in seiner Eingabe berichtet.

3-5. Schreiben Sie ein Programm, das die Noten mehrerer Studenten auf einmal erfassen kann. Das Programm kann zwei `vector`s synchron halten: Der erste sollte die Namen der Studenten und der zweite ihre Endnoten, die während dem Einlesen der Werte berechnet werden können, enthalten. Wir werden in Abschnitt 4.2.3 auf Seite 93 sehen, wie eine variable Anzahl von Noten zusammen mit den Studentennamen verwaltet werden kann.

3-6. Die Berechnung der Durchschnittsnote in Abschnitt 3.1 auf Seite 57 könnte eine Division durch Null enthalten, falls der Student keine Noten eingab. Division durch Null ist in C++ undefiniert, was bedeutet, dass die Implementierung darauf beliebig reagieren darf. Was tut Ihre C++-Implementierung in diesem Fall? Schreiben Sie das Programm so um, dass sein Verhalten nicht mehr davon abhängt, wie die Implementierung mit der Division durch Null umgeht.