Einführung in die Programmierung mit C++

Teil I

Thomas Letschert

FH Giessen–Friedberg

Korrektur: Eugen Labun (labun@gmx.de)

Version vom 11. Dezember 2006

Inhaltsverzeichnis

1 Hardware und Software

1.1 Programme und Computer

Software – Maschinen aus Ideen

Lokomotiven, Toaster und Videorecorder werden von Maschinenbau– und Elektro–Ingenieuren hergestellt. Bauingenieure bauen Brücken und Häuser. Informatiker – *Software*–Ingenieure, wie sie sich oft selbst nennen – stellen Software her. Software ist anders als Brücken, Toaster, Lokomotiven oder alles andere sonst, das von anderen (richtigen ?) Ingenieuren gebaut wird. Wird Software gebaut? Was ist Software?

Software begegnet uns meist als Programm das auf einem PC installiert ist, oder das man in Form einer kleinen CD kauft und dann selbst installiert. Ein *installiertes Programm* erscheint meist als kleines Bildchen oder Symbol (oft *Icon* genannt) auf dem Bildschirm. Klickt man es mit der Maus an, dann beginnt der PC sich in einer besonderen Art und anders als zuvor zu verhalten. Vielleicht ertönt Musik, vielleicht hört man Motorengeräusche und man sieht das Innere eines Flugzeug–Cockpits auf dem Bildschirm, vielleicht zeigt der PC aber nur stumm einen Bildschirmausschnitt – ein "Fenster" – als langweilige weise Fläche in der die Zeichen erscheinen, die man auf der Tastatur tippt.

Richtige Maschinen tun etwas Richtiges. Toaster toasten, Lokomotiven ziehen Züge. Entweder sind die richtigen Maschinen so schwer, dass man sie nicht heben kann, oder, wenn doch, dann man kann sie fallen lassen und sind kaputt, oder die Füße tun weh, oder beides. Programme sind anders. Sie sind nicht schwer oder leicht und sie tun auch nicht wirklich etwas. Zumindest tun sie es nicht selbst. Wie ein Traum oder ein Albtraum den Geist, oder wie ein Virus eine Zelle, so besetzen Programme einen Computer und bringen ihn dazu sich in einer bestimmten Art zu verhalten. Meist ist dies erwünscht und mit dem Programm auch mehr oder weniger teuer bezahlt. Manche werden aber gegen unseren Willen installiert und aktiviert und bringen den PC dazu sich in einer Art zu verhalten, die uns nicht gefällt – dann nennen wir sie auch manchmal tatsächlich "Virus".

Computer ohne Programme sind nur *Hardware* – nutzlose Energie– und Platzfresser. Programme ohne Computer sind nur *Software* – wirkungslose Ideen von etwas, das geschehen könnte. Die geniale Idee der Informatik besteht darin, die Idee des Tuns – das Programm – und sein Ausführungsorgan – die Maschine – zu trennen. Mit den Computern hat sie Maschinen geschaffen, die keinen eigenen Willen, keinen eigenen Zweck, kein eigenes Verhalten haben, sondern sich jederzeit, wie Zombies, bereitwillig einem beliebigem fremden Willen, dem Programm, unterwerfen. Beide, Programme und Computer, können damit unabhängig voneinander weiterentwickelt und nahezu beliebig kombiniert werden. Diese Idee, so seltsam, exotisch und wenig anwendbar sie vor einigen Jahren einmal erschien, hat in kurzer Zeit dieWelt erobert und ihre Zombies kommen manchem von uns inzwischen recht selbstbewusst vor.

Grundbausteine eines Computers

Computer, Informatiker nennen sie oft auch "Rechner", sind dazu da, einen fremden Willen auszuführen, der ihnen in Form eines Programms aufgenötigt wird. Zu diesem Zweck sind sie in einer bestimmtenWeise aufgebaut. Bei aller Verschiedenheit im Detail folgt dieser Aufbau einem Grundmuster, das sich der Großcomputer eines Rechnenzentrums, der PC auf dem Schreibtisch, das Innere eines Handys und das Steuerungselement in einer Waschmaschine teilen. Dieses Grundmuster nennt man etwas hochtrabend "Architektur" der Rechner. Sie beruht auf folgenden Gundbestandteilen:

- *Prozessor*, auch: CPU (*Central Processing Unit*),
- *Hauptspeicher*, auch: *Arbeitsspeicher*, oder *RAM* (*Random Access Memory*),
- *Plattenspeicher* und
- *Ein–/Ausgabegeräte* wie Monitor, Tastatur, Maus, etc.

Eine Waschmaschine hat (noch) keinen Monitor und ein Handy (noch) keinen Plattenspeicher, aber der grundsätzliche Aufbau der Rechner in allen Geräten ist stets der gleiche.

Programm

Der Prozessor verarbeitet die Daten, die sich im Hauptspeicher befinden. Er befolgt dabei die Anweisungen eines *Programms*. Ein Programm besteht aus einer Folge von Anweisungen. *Anweisungen* nennt man auch Befehle. Eine

Anweisung ist eine Aktion, die vom Prozessor ausgeführt werden kann. Das Programm befindet sich im Hauptspeicher. Es sagt mit all seinen einzelnen Anweisungen dem Prozessor, und damit dem Computer insgesamt, was er tun soll.Wenn der Computer läuft, holt sich also der Prozessor die Befehle des Programms aus dem Hauptspeicher und führt sie aus – Befehl für Befehl.

Im Plattenspeicher befinden sich die Programme und Daten des Systems und seiner Benutzer. Sollen sie verarbeitet werden, dann müssen sie zuerst in den Hauptspeicher transportiert (geladen) werden. Die Ein–/Ausgabegeräte dienen dazu mit dem Computer zu kommunizieren. Der Plattenspeicher ist sehr groß. Er enthälle Programme und Daten. Der Hauptspeicher ist kleiner und enthält nur die Daten und Programme, die gerade verarbeitet werden. Der Prozessor ist noch kleiner. Er enthält nur den einzelnen Befehl der gerade verarbeitet wird und dazu noch einige wenige Daten die dazu gebraucht werden.

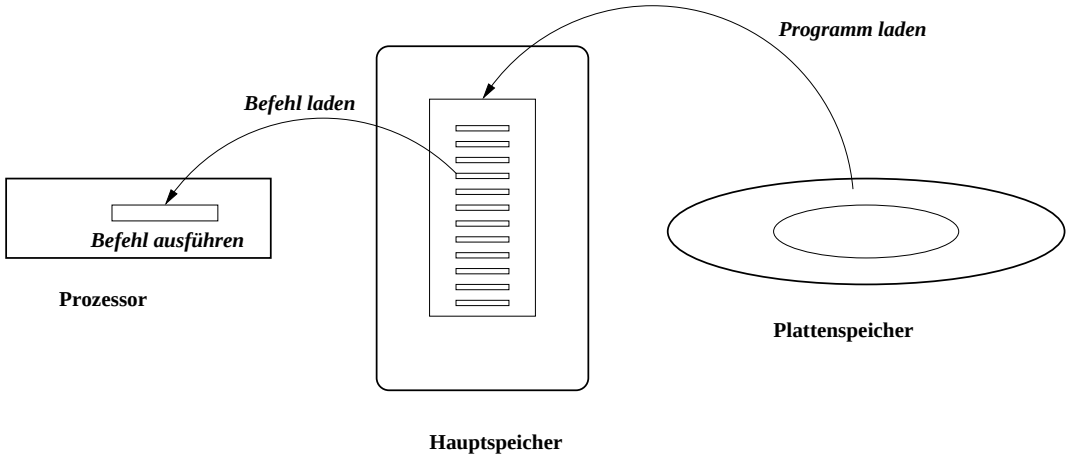


Abbildung 1: Programme und Befehle laden und ausführen

Prozessor

Der *Prozessor* (CPU) bildet zusammen mit dem Hauptspeicher den Kern des Rechners. Er enthält einige wenige Speicherzellen, *Register* genannt. In diese Register kann er aus dem Hauptspeicher Daten laden, sie durch Rechenoperationen verändern und dann wieder in den Hauptspeicher schreiben. Die Aktivität des Prozessors wird durch ein Programm gesteuert. Bei einem Programm handelt es sich ebenfalls um Daten. Jeder einzelne Befehl des Programms besteht aus ein paar Nulllen und Einsen und ein Programm besteht typischerweise aus Millionen von Befehen. Sie stehen im Hauptspeicher und der Prozessor holt sie sich häppchenweise, Befehl für Befehl in ein spezielles Register (eine Speicherstelle) und dann führt sie aus.

Manchmal passieren dabei Fehler. So kommt es öfter vor, dass der Prozessor irrtümlich eine Folge von Nullen und Einsen als Befehl laden und ausführen will die gar nicht als Befehl gemeint ist, sondern eine Zahl darstellt die verarbeitet werden soll. Oder der Befehl sagt, dass ein Wert von einer Speicherstelle geholt werden soll, die nicht exisitiert oder die gesperrt ist. Solche Fehler beruhen darauf das das gerade ausgeführte Programm fehlerhaft ist. Der Rechner reagiert darauf mit dessen sofortigem Abbruch. In den Urzeiten hat der Rechner dazu einfach seine insgesamt Aktivität eingestellt. Heute wird die Situation (in aller Regel) dadurch bereinigt, dass ein anderes Programm ausgeführt wird.

Hauptspeicher

Der *Hauptspeicher* enthält viele Speicherzellen: Sie sind direkt adressierbar und ihr Inhalt kann von der CPU (dem Prozessor) sehr schnell gelesen und geschrieben werden kann. Eine Speicherzelle enthält typischerweise ein *Byte*. Ein Byte sind 8 Bit (8 Binärzeichen, 0 oder 1). “Direkt adressierbar” bedeutet, dass jede Zelle, d.h. jedes Byte, eine Adresse hat und der Prozessor jederzeit auf jede beliebige Speicherzelle zugreifen kann. Dieser wahlfreie Zugriff (engl. *Random Access*) ist sehr wichtig. Ohne ihn könnten die Programme nicht vernünftig ausgeführt werden. Daten müssen an beliebigen, nicht vorhersehbaren Stellen geholt und gespeichert werden und von einer Anweisung muss je nach Bedarf zu einer beliebigen anderen verzweigt werden können.

Plattenspeicher

Die Schnelligkeit und die direkte Adressierbarkeit macht den Hauptspeicher sehr teuer. Zur Speicherung von Massendaten wird darum ein billigerer Hintergrundspeicher eingesetzt. Typischerweise ist das ein *Plattenspeicher*. Der Plattenspeicher – die "Festplatte" – kann sehr viele Daten aufnehmen. Er ist aber vergleichsweise langsam und nicht direkt adressierbar. Daten können nur in großen Einheiten und nicht byteweise gelesen und geschrieben werden. Daten im Plattenspeicher müssen darum immer zuerst in den Hauptspeicher geladen werden, bevor der Prozessor sie verarbeiten kann.

Externe Geräte

Den Plattenspeicher und die Ein/Ausgabegeräte wie Tastatur, Graphikkarte (steckt im Computer und steuert den Monitor) und Maus, CD-Laufwerk, etc. bezeichnet man als *externe Geräte*. Sie liegen außerhalb von Prozessor und Hauptspeicher, die den innersten Kern des Computers bilden. Die externen Geräte werden auch durch den Prozessor gesteuert. Informationen, die über die externen Geräte eintreffen, werden vom Prozessor angenommen und im Hauptspeicher und dann eventuell auf der Platte gespeichert.

Aus der Sicht des Rechners sind also Dinge wie Maus oder das CD-Laufwerk am alleräussersten Ende der Welt. Sie sind "extern" und liefern oder schlucken Daten. Dinge wie Auge, Hand oder auch die CD von denen kommen oder gehen, sind nicht einmal mehr extern, sie spielen in der Welt der Rechner (der Programme) keine Rolle.

Dateien und Programme

Die Masse der Daten liegt in Dateien auf der Festplatte. Dateien sind in Verzeichnissen (auch "Ordner", oder engl. *Directory*) organisiert. Eine Datei kann Daten beliebiger Art enthalten: Texte, Musik, Graphiken, etc. Eine Datei kann auch ein Programm enthalten. Programme können in den Hauptspeicher geladen und dann ausgeführt werden. Das nennt man "Aktivieren des Programms". Oft werden Programme durch kleine Symbole (Bildchen) auf dem Bildschirm dargestellt. Klickt man sie an, dann werden sie aktiviert, also von der Platte in den Hauptspeicher geladen und dann ausgeführt. Beim Ausführen liest der Prozessor Befehl für Befehl des Programms und befolgt ihn. Ein Programm besteht aus einer – meist sehr langen – Sequenz von Bits, die vom Prozessor häppchenweise als Befehle verstanden werden. Ein *Programm* ist also eine Datei deren Inhalt vom Prozessor verstanden wird.

Dateien die Befehle für den Prozessor enthalten nennt man ausführbare Dateien. Dateien mit einem anderen Inhalt können nicht aktiviert werden. Eine Textdatei beispielsweise kann aber gedruckt werden, man kann sie mit Hilfe eines Editor betrachten und verändern. Oft identifiziert man eine ausführbare Datei mit ihrem Inhalt und sagt auch "Programm" zu der Datei die das Programm enthält.

Das Betriebssystem startet Programme

Ein Programm wird also gestartet, indem der Inhalt der Datei, die es enthält, in den Hauptspeicher kopiert wird und der Prozessor dann von dort den ersten Befehl des Programms lädt und ausführt. Für das Kopieren in den Hauptspeicher und die Umorientierung der CPU (des Prozessors) auf die neue Aufgabe ist das *Betriebssystem* zuständig. Das Betriebssystem ist selbst ein Programm, das ständig aktiv ist und dem Benutzer und seinen Programmen dabei hilft die Hardware des Systems zu nutzen.

Meist startet das Betriebssystem ein Programm nicht aus eigenem Entschluss, sondern nachdem der Benutzer es dazu aufgefordert hat. Heutzutage klickt man dazu meist ein kleines Bildchen ("Icon") an, das das Programm symbolisiert. Das System kennt die Koordinaten des Bildchens und welches Programm (d.h. welche Datei) damit gemeint ist. Klickt man in diesem Bereich, dann startet das (Betriebs-) System das Programm. Das Programm und das zughörige Bild muss dem Betriebssystem dazu natürlich vorher bekannt gemacht werden, man sagt, das Programm wird registiert.

Wenn das Betriebssystem ein Programm startet, dann bedeutet das, dass es ihm die CPU zur Ausführung seiner Befehle überlässt. Natürlich ist das Programm nicht wirklich etwas Aktives. Der Prozessor ist der aktive Teil. Er führt einen Befehl des Systems aus, dieser veranlässt ihn den ersten Befehl des Programms zu laden und dessen Ausführung zieht das Laden uns Ausführen der anderen Befehle des Programms nach sich. Am Ende oder bei einem Fehler wird dafür gesorgt, dass es wieder mit Befehlen des Systems weitergeht.

Eingabe von Kommandos

Unsere einfachen Übungsprogramme funktionieren oft nicht richtig, und wenn doch, dann aktivieren wir sie ein einziges Mal, nur um zu sehen, dass sie korrekt sind. Für diese Fingerübungen wäre eine Registrierung beim Betriebssystem viel zu aufwändig. Wir benutzen darum eine andere, einfachere Methode, um Programme zu starten: die *Kommandoeingabe*.

Ähnlich wie der Prozessor seine Maschinenbefehle interpretiert (= versteht und ausführt), hat das Betriebssystem eine Menge von Befehlen, die es direkt ausführen kann. Die Befehle nennt man "Kommandos". Die Befehle, die von der CPU verstanden werden, und aus denen ein ausführbares Programm besteht sind kryptische Folgen von Nullen und Einsen. Die Kommandos dagegen sind Texte, die man verstehen kann. Man tippt sie an der Tastatur ein und sie werden dann vom System befolgt. Beispielsweise kann man

```
dir
```

eintippen und das System liefert eine Liste aller Dateien im aktuellen Verzeichnis. Die Kommandos werden von einem Teilprogramm des Betriebssystems ausgeführt, das dazu selbst erst gestartet werden muss. Man nennt es allgemein "Kommandointerpretierer". In einem Windowssystem wird der Kommandointerpretierer "DOS–Fenster" oder ähnlich genannt. Bei einem Linux/Unix–System nennt man ihn meist "Terminal".

Ein Kommando besteht oft einfach aus dem Namen eines Programms. Genau genommen ist es der Name der Datei die ein Maschenprogramm enthält. Tippt man ihn ein und schickt ein *Return* hinterher, dann weiß das System, dass es das Programm ausführen soll.

1.2 Programme und Algorithmen

Algorithmus: Definition einer zielgerichteten Aktion

Ein *Algorithmus* beschreibt, wie eine komplexe Aufgabe als Folge von einfacheren Aktionen gelöst werden kann. Es ist eine Handlungsanweisung, eine Aussage darüber "wie etwas gemacht wird". Ein Algorithmus beschreibt eine Problemlösung in Form von Einzelschritten. Ein Backrezept ist ein gutes Beispiel für einen Algorithmus. In ihm wird Schritt für Schritt beschrieben, wie aus den Backzutaten ein Kuchen gemacht wird. Ein anderes Beispiel sind die Verfahren zur Addition, Subtraktion und Multiplikation mehrstelliger Zahlen, die man, zumindest früher, in der Grundschule lernte.

"Algorithmus" ist ein informaler und allgemeiner Begriff. Ein Algorithmus sagt, was getan werden muss, um ein Ziel zu erreichen. Das kann richtig oder falsch sein, völlig unabhängig davon, in welcher Form und für wen es aufgeschrieben wurde. Ein Backrezept, in dem der Kuchen 15 Stunden bei 450 Grad gebacken wird, ist höchst wahrscheinlich falsch. Dabei ist es egal in welcher Sprache es aufgeschrieben wurde. Das heißt natürlich nicht, dass die Sprache, in der er verfasst wurde, gleichgültig ist. Algorithmen in einer Formulierung, die niemand versteht, sind nutzlos.

Ein Algorithmus muss nicht nur auf die sprachlichen Fähigkeiten dessen Rücksicht nehmen, der ihn ausführen soll. Auch seine sonstigen Fähigkeiten sind von Belang. So ist der folgende Algorithmus zur Erlangung einer größeren Geldmenge sowohl verständlich, als auch korrekt:

1. Reise in die Zukunft.
2. Stelle fest, welche Lottozahlen gezogen werden.
3. Kehre zurück und fülle einen Lottoschein mit diesen Zahlen.

Die meisten von uns sind aber wohl nicht in der Lage diesen Algorithmus auszuführen. Die Frage, ob er prinzipiell ausführbar ist, ist noch offen. Im Allgemeinem stetzt man immer dann, wenn informal von einem Algorithmus die Rede ist, voraus, dass er sowohl verständlich ist, als auch dass er ausgeführt werden kann.

Programme

Für praktische Zwecke muss der Begriff des Algorithmus präzisiert werden. Wir müssen dabei das Ausführungsorgan des Algorithmus genauer in Betracht ziehen. Es muss die Anweisungen sowohl verstehen, als auch ausführen können. Speziell dann, wenn das Ausführungsorgan eine Maschine ist, muss genau festgelegt sein, welche Formulierung welche Aktion genau auslösen soll. So präzise dargelegte Algorithmen nennen wir "Programme".

Programme sind Algorithmen, die in einer bestimmten festen Form aufgeschrieben sind. Die Regeln, nach denen ein Programm aufgeschrieben wird, werden als *Programmiersprache* definiert. Die feste Form und die strengen Regeln sind notwendig, da die *Anweisungen* des Programms von einem Rechner (= Computer) ausgeführt werden sollen. Die Programmiersprache legt zum einen die Notation fest und zum anderen sagt sie, was die Formulierungen in dieser Notation bedeuten oder bewirken sollen.

Programmiersprachen bestimmen damit zwei Aspekte eines Programms:

1. *Syntax* (Form): Die exakte Notation in der das Programm als Text aufgeschrieben werden muss.
2. *Semantik* (Inhalt): Was bedeutet ein solcher (Programm-) Text, welche Anweisungen sind möglich und was bewirken (bedeuten) sie genau.

Eine Programmiersprache sind damit eine *formale* Sprache: es ist zweifelsfrei und exakt festgelegt, welche Texte als korrekte Programme dieser Sprache gelten und was sie bedeuten.

Programme unterscheiden sich von Algorithmen durch den höhreren Grad an Präzision und Formalität. Sie sind in einer, bis auf das letzte Komma festgelegten, Notation zu verfassen und wenden sich an ein Ausführungsorgan, das ein ganz genau definiertes Repertoir an Fähigkeiten hat. Jedes Programm ist ein Algorithmus aber viele Algorithmen sind zu informal um Programme zu sein.

Syntax und Semantik

In der Informatik ist es wichtig zwischen einem Text und dem was er bedeutet zu unterscheiden. Beispielsweise ist "`Kuh`" keine Kuh und "`123`" ist keine Zahl. Beides sind kurze Texte, die aus einer Folge von drei Zeichen bestehen. Die Zeichen des Textes "`123`" stehen für eine Zahl, sie *bezeichnen* oder *bedeuten* eine Zahl. Normalerweise macht es keinen Sinn pedantisch zwischen Texten und dem was sie bedeuten zu unterscheiden. Manchmal ist es jedoch notwendig und in der Informatik ist dieses "manchmal" häufiger als im Alltagsleben.

Die Bedeutung des Textes "`123`" hängt von dem Zahlsystem ab, in dem wir uns bewegen. Im Zehnersystem ist die Zahl hundert-drei-und-zwanzig (123_{10}) gemeint. Im Vierer-System bedeutet er sieben-und-zwanzig ($27_{10} = 123_4$). Im Dreier- oder Zweiersystem ist "`123`" kein gültiger Ausdruck.

Zahlsysteme kann man als sehr einfache Art von formaler Sprache betrachten, die ihre eigene Syntax und Semantik haben. Die Syntax legt fest, welche Texte als korrekte Texte gelten, nennen wir sie gültige (korrekte) *Zahlausdrücke*. Die Semantik sagt, welche Zahl sie darstellen. Die Syntax des Viersystems legt beispielsweise fest, dass jede Folge der Ziffern `0`, `1`, `2`, `3` die nicht mit `0` beginnt, ein korrekter Zahlausdruck ist. Die Semaktik des Viersystems legt fest, dass mit einem Zahlausdruck die Zahl gemeint ist, die sich aus der Summe der Zifferwerte, multipliziert mit der jeweilgen Viererpotenz, ergibt ($123_4 = 1 \cdot 4^2 + 2 \cdot 4^1 + 3 \cdot 4^0 = 27_{10}$).

Maschinensprache und Maschinenprogramme

Die interessantesten Programme sind die, die von der *Hardware* eines Rechners verstanden und ausgeführt werden. Genauer gesagt ist es der Prozessor, der die Programme ausführt. Wer sonst will schon Programme ausführen. Um zum Ausdruck zu bringen, dass eine Maschine die Programme versteht, nennen wir sie genauer *Maschinenprogramme*.

Die Anweisungen eines Maschinenprogramms werden also vom Prozessor verstanden. Sie sind darum in einer ihm angenehmen Form vefasst, als Folgen aus 0-en und 1-en: Befehle in Form von Bitmustern, von denen jedes eine Bedeutung hat, die auf die Fähigkeiten des Prozessortyps zugeschnitten ist. Die Fähigkeiten eines Prozessors darf man dabei nicht überschätzen. Viel mehr, als einfache arithmetische Operationen, das Verschieben von Daten (auch wieder Bitmuster) von einer Speicherstelle zur anderen und das Laden neuer Anweisungen von bestimmten Speicherstellen im Hauptspeicher, ist nicht möglich.

Menschen sind kaum in der Lage Maschinenprogramme zu lesen, geschweige denn korrekte Maschinenprogramme zu schreiben. Die Programme der frühen Pioniere der Informatik wurden zwar in Maschinensprache verfasst. Sehr schnell hat man dann aber eingesehen, dass Bitmuster, die das Verändern und Verschieben von Bitmustern beschreiben, nicht unbedingt eine besonders angenehme Art sind einen Algorithmus zu beschreiben.

Für Menschen ist es einfach eine Zumutung sich auf die Ebene eines Prozessors zu begeben. Prozessoren, die in der Lage sind Anweisungen auf dem Niveau von Menschen zu bearbeiten, sind dagegen technisch und ökonomisch nicht realisierbar. Die Lösung des Problems besteht darin, Menschen Programme in "menschlicher" Form schreiben zu lassen und sie dann in Maschinensprache zu übersetzen. Das Übersetzen sollte dabei am besten von einem Computer übernommen werden.

Höhere Programmiersprache

Programme in einer *höheren Programmiersprache* enthalten Anweisungen, die sich in Form und Inhalt an den Fähigkeiten von Menschen orientieren. Beispiele für solche Sprachen sind *Pascal, C, Java* und eben die Sprache *C++* mit der wir uns hier näher beschäftigen wollen. Programme in höheren Programmiersprachen bestehen aus Anweisungen die Menschen verstehen und schreiben können. Beispielsweise ist

```
#include <iostream>

using namespace std;

int main () {
  cout << "Hallo !" << endl;
}
```

ein Programm in der höhreren Programmiersprache C++. Man sieht es vielleicht nicht auf den ersten Blick, aber es soll den Computer dazu bringen "`Hallo !`" auf dem Bildschirm auszugeben. Solche Programme sollen Menschen leicht schreiben oder verstehen können - zumindest wenn sie Programmierer sind. Zumindest ist es leichter zu verstehen als ein Maschinenprogramm in der Form:

```
48
00 00
00 90 88 04 08 04
00 00
00 21
00 0e
00 51 01
00 00
90
9a 04 08 8c 00 00 00
... noch viele derartige Zeilen ...
```

Der Nachteil der Programme in höheren Programmiersprachen ist, dass es keinen Prozessor - also keine Maschine - gibt, die ihre Anweisungen versteht. Nicht nur die Notation ist dabei für den Prozessor unverständlich, es werden auch Aktionen von ihm verlangt, die er in dieser Form nicht ausführen kann.

Die Programme der höheren Programmiersprachen enthalten Anweisungen an eine *virtuelle Maschine*, also eine nur gedachte Maschine. Im C++-Programm steht mit

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

so etwas wie "Gib Hallo aus!". Das ist für den Prozessor so unverständlich und unlösbar, wie wenn man dem Autor dieser Zeilen sagen würde: "Backe eine köstliche Sahnetorte!". So wie ich *im Prinzip* eine Sahnetorte backen kann, wenn man mir haarklein jeden einzelnen Schritt erklärt, so kann der Prozessor "Hallo" ausgeben. Man muss ihm nur jeden Schritt dazu haarklein erklären. Diese Erklärung steht aber nicht im C++-Programm. Es wendet sich darum an jemanden, der nicht exisitiert - an eine virtuelle Maschine eben.

Compiler: Programm in Maschinenprogramm übersetzen

Die Lücke zwischen einem Programm, das Menschen konzipieren und schreiben können und dem Prozessor, der nur mit 0–en und 1–en hantieren kann, wird vom Compiler gefüllt.

Ein *Compiler* ist ein Programm, das ein Programm in einer höheren Programmiersprache, das *Quellprogramm*, in ein äquivalentes Maschinenprogramm übersetzt. (Siehe Abbildung 2). Die für Menschen verstehbaren Anweisungen an eine gedachte (virtuelle) Maschine werden dabei in maschinenlesbare Anweisungen an den Prozessor umgesetzt. Das sollte er natürlich so tun, dass das Quellprogramm und das erzeugte Maschinenprogramm im Endeffekt die gleiche Wirkung haben. In der Regel ist der Compiler korrekt und tut was man von ihm erwartet. Das Quell-Programm von oben wird vom Compiler in ein Maschinenprogramm - eine lange unverständliche Folge von Nullen und Einsen - übersetzt, die der Prozessor ausfüren kann und dabei das Gewollte tut, nämlich `Hallo !` auf dem Bildschirm ausgeben.

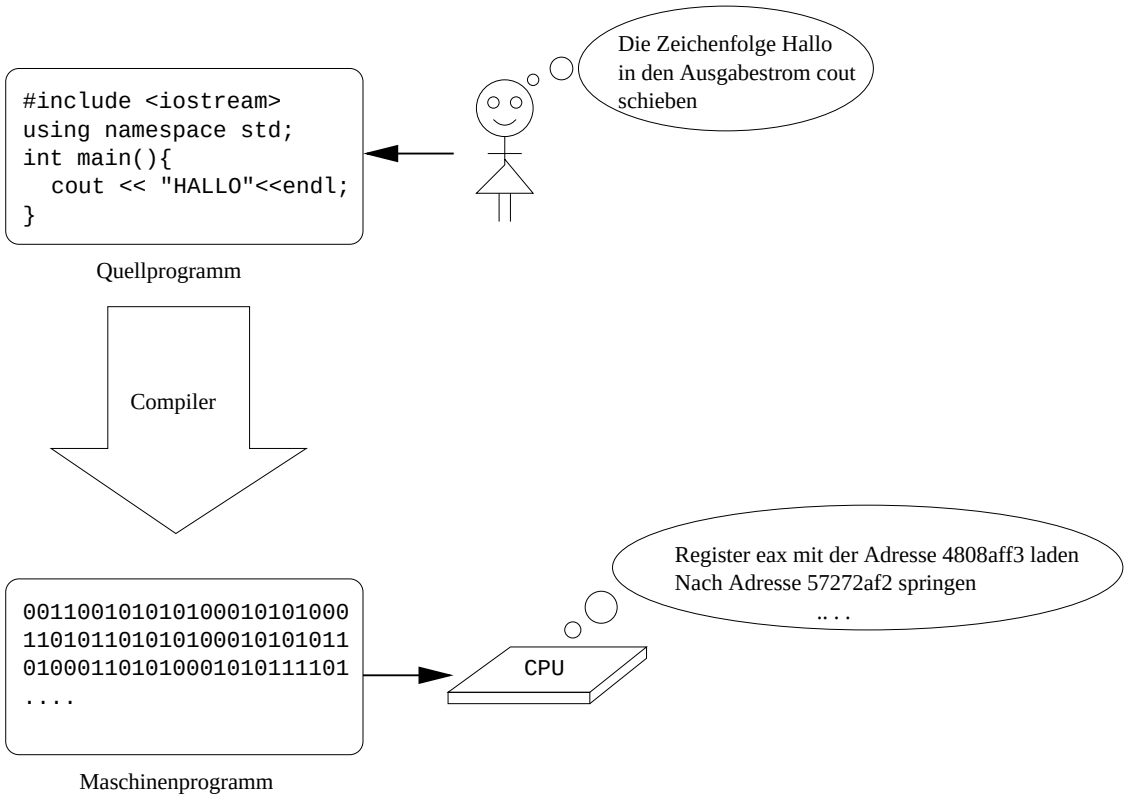


Abbildung 2: Quellprogramm, Compiler, Maschinenprogramm

Programme und Software

Die scheinbare Leichtigkeit, mit der Programme entwickelt werden können, und die fehlenden materielle Komponente verleiten dazu, ihren Wert zu unterschätzen. Tatsächlich stellt Software hat heutzutage einen beträchtlichen Anteil am Wert und den Kosten vieler Produkte. Der PC spielt dabei eine Rolle, aber er ist nicht der einzige. Praktisch jedes technische Produkt enthält inzwischen einen erheblichen Anteil an Software, deren Zuverlässigkeit überWohl undWehe vonWeltfirmen mitentscheiden kann. Die Effizienz mit der admistrative Vorgänge in Banken, Behörden und Wirtschaftsunternehmen abgewickelt werden können entscheidet mit über deren Rentabiltität und damit deren langfristige Existenz.

Die Fähigkeit Software von guter Qualität zu günstigen Preisen herstellen zu können wird in Zukunft von entscheidender Bedeutung für den denWohlstand ganzer Regionen und Volkswirtschaften sein. Trotzdem herrscht oft die Vorstellung, dass Software nichts anderes ist, als ein etwas gößeres Programm an dem ein paar Hacker mehr herum gehackt haben. Software ist die Steuerung komplexer Systeme und Vorgänge. Sie wird letztlich in Form von Programmen realisiert, ist aber viel mehr. Sie umfasst alle Aspekte der Analyse, der Konzeption und Planung von Steuerungsvorgänge sowie der langfristigen Wartung der erstellen Programme. Dazu gehört viel mehr als nur die Beherrschung einer Programmiersprache und die Fähigkeit ein Programm von ein paar Hundert Zeilen schreiben zu können. Umgekehrt kann aber ohne diese Fähigkeiten auch keine Software erstellt werden.

Ein Architekt muss mehr können als mauern, aber ohne mauern zu können, kann man kein Haus bauen. Genauso muss ein Software-Ingenieur mehr können als programmieren, aber ohne das ist sie oder er Nichts. Beginnen wir also damit Steine auf Stein zu setzen.

Zusammenfassung

Fassen wir noch einmal die Begriffe in Zusammenhang mit Programmen zusammen:

- *Algorithmus*: Ein Algorithmus ist ein Verfahren nach dem eine Aufgabe erfüllt werden kann.
- *Programm*: Ein Programm ist die Beschreibung eines Algorithmus' in einer bestimmten fest definierten Notation (Syntax) und klar definierter Bedeutung (Semantik) das sich an ein Ausführungsorgan mit exakt definierten Fähigkeiten wendet. Es ist entweder ein Maschinenprogramm oder ein Programm in einer höheren Programmiersprache.

- *Programmiersprache*: Festlegung der Syntax und Semantik von Programmen (Programm-Texten). Man unterscheidet Maschinensprachen und höhere Programmier-Sprachen. Programmiersprachen sind formale Sprachen.
- *Maschinenprogramm*: Ein Programm, das vom Prozessor eines Rechners verstanden und ausgeführt werden kann.
- *Programm in einer höheren Programmiersprache*: Ein Programm in einer höheren Programmiersprache kann von Menschen geschrieben und verstanden werden. Es kann nur von virtuellen (gedachten) Maschinen direkt ausgeführt werden. Um es auf realen Maschinen auszuführen benötigt man einen einen Compiler.
- *Compiler*: Ein Programm das ein Quellprogramm in ein äquivalentes Maschinenprogramm übersetzt.
- *Quellprogramm*: Ein Programm in einer höheren Programmiersprache. Es wird vom Compiler in ein Maschinenprogramm übersetzt. Es ist die “Quelle” Übersetungsvorgangs.

2 Erste Programme

2.1 Programmerstellung

Der Editor

Am Anfang steht das Programm als Idee im Kopf der Programmiererin. Mit Hilfe eines weiteren (Hilfs–) Programms, das *Editor* genannt wird, tippt man es ein und speichert es dann in einer Datei (engl. *File*). Ein Editor ist ein Programm, eine Art "kleines einfaches Word", das keine Formatierungen vornimmt und die eingegebenen Zeichen im ASCII–Code in einer Datei speichern kann. Der ASCII-Code ist eine einfache Form zur Speicherung von Texten. Im ASCII–Code gibt es große und das kleine Buchstaben ("`A`" und "`a`"), Zahlen und einige wenige Sonderzeichen, es gibt aber keine unterschiedlichen Schriftgrößen (10, 12, 14 Punkte, etc.) oder unterschiedliche Schriftarten (fett, unterstrichen, schräg, etc.).

ASCII steht für *American Standard Code of Information Interchange* und ist eine recht alte Norm nach der Zeichen als Bitfolgen codiert werden. Der ASCII-Code ist heute keine rein amerikanische Norm mehr, sondern eine internationale Konvention nach der Zeichen, wie man sie auf der Tastatur findet – Buchstaben und Zahlen –, jeweils als Folge sieben `0`–en und `1`–en gespeichert werden. – In einem Rechner wird ja alles in Form von 0–en und 1–en gespeichert. Der ASCII–Code enthält keine nationalen Sonderzeichen wie beispielsweise *ä* oder *é*. In den 80–ern wurde eine auf 8 Bit erweiterte Variante des ASCII-Codes definiert, die die nationalen Sonderzeichen der, auf dem lateinischen Alphabet basierenden, europäischen Sprachen enthält. Der offizielle Name dieses Codes ist *ISO 8859–1* oder *Latin–1*. *ISO 8859* gibt es in diversen Varianten, beispielsweise enthält *ISO 8859–15* das Euro–Zeichen.[1]

Am Ende des Editierens steht dann das Programm als *Quellcode* in der Datei. Man nennt das Programm in dieser Form *Quellprogramm*.

Beispiel: Das Hallo–Programm

Ein kleines C++–Programm, das einfach nur das Wort (die Zeichenfolge) `Hallo !` ausgibt, ist:

```
#include <iostream>

using namespace std;

int main () {
  cout << "Hallo !" << endl;
}
```

Datei mit C++ Quellcode erzeugen

Dieser Text muss in einer Datei gespeichert werden. Dateien sind auf der Festplatte des Rechners und diese ist in *Verzeichnisse* (Ordner, engl. *directory*) aufgeteilt. Zunächst erzeugen wir ein neues Verzeichnis für die C++–Programme von Programmieren I und machen es mit einem geeigneten Kommando (z.B. `cd`) zum aktuellen Verzeichnis (Wir "gehen in das Verzeichnis"). Dazu öffenen wir beispielsweise ein Kommandofenster und geben ein:

```
> mkdir ProgI
> cd ProgI
```

(Das Größerzeichen ist nicht Teil der Eingabe.)

Als nächstes muss ein Editor aktiviert werden. Es gibt viele Editoren und es ist eine Frage der persönlichen Vorlieben, welchem man den Vorzug gibt. Wir öffnen im Editor eine Datei namens `hallo.cc` und tippen das Programm von oben – den Quellcode – ein. Das muss sorgfältig geschehen, jedes einzelne Zeichen ist wichtig.

Der Code muss jetzt noch *gespeichert* werden. Achtung: wenn ein Text in den Editor eingegeben wurde, dann steht er noch nicht in einer Datei (auf der Platte des Rechners). Der Editor ist selbst ein Programm, das Ihren Text zunächst einmal intern – im Hauptspeicher – verwaltet. Nur was im Plattenspeicher abgelegt wurde bleibt erhalten!

Der Name der Datei ist in unserem Beispiel "`hallo.cc`", er kann aber im Prinizip frei gewählt werden. Die

[1] Die ISO ist die *International Standard Organisation*, der Dachverband aller nationalen Normungsbehörden. Die meisten Hersteller halten sich an ISO–Normen. Ein bekannter Hersteller von PC–Software benutzt allerdings eine eigene Variante des ASCII–Codes um nationale Sonderzeichen darzustellen. Das ist berechtigt, da er auch über ein größeres Vermögen verfügt als die Mehrzahl der an ISO beteiligten Staaten.

Datei–Endung "`.cc`" sollte sie als Datei mit C++–Quellcode kennzeichnen. Wir empfehlen die Endung `.cc`, insgesamt üblich sind folgende Dateiendungen:

- `.cc`
- `.cpp`
- `.cxx`
- `.C` (nur auf Systemen, die Groß– und Kleinschreibung in Dateinamen unterscheiden)
- `.c` (wenn nicht zwischen C– und C++–Programmen unterschieden wird.)

Compileraufruf

Jetzt ist der Text des Programms erstellt und in einer Datei gespeichert. Der Compiler kann aktiviert werden, um es in ein ausführbares (Maschinen–) Programm zu übersetzen. Wir tippen dazu das *Kommando*

```
> g++ hallo.cc
```

ein. (Das Zeichen ">" ist nicht Teil des Kommandos.) `g++` ist der Name des C++–Compilers und `hallo.cc` ist der Name der Datei, in der das zu übersetzende Programm steht. Enthält das Programm keinen Fehler, dann erzeugt der Compiler ein Maschinenprogramm in der Datei `a.out`. Es kann mit dem Kommando

```
> ./a.out
```

aktiviert werden und sollte dann die Ausgabe `Hallo !` erzeugen. `a.out` ist hier der Name der Datei, in die der Compiler das Maschinenprogramm schreibt und `./` bedeutet "das aktuelle Verzeichnis, der aktuelle Ordner". Wir aktivieren also das Programm `a.out` in dem Verzeichnis (Ordner) in dem wir uns gerade befinden.

Fehlermeldungen des Compilers

Wenn bei der Eingabe des Programms ein Fehler gemacht wurde, wird der Compiler sich mit einer oder mehreren Meldungen beschweren. Die Meldungen geben stets einen Hinweis auf die Ursache des Fehlers. Man sollte die Fehlermeldungen lesen und versuchen ihren Sinn zu verstehen.

Compileroptionen

Das vom Compiler aus der Quelldatei erzeugte Maschinenprogramm steht hier in einer Datei namens `a.out`. Man sagt dazu auch das Programm *heißt* `a.out`. Wenn dieser Name nicht gefällt, dann kann man mit der Option `-o` (`o` wie "Output") der erzeugten Datei ein beliebiger anderer Name gegeben werden. Beispielsweise erzeugt das Kommando

```
> g++ -o hallo hallo.cc
```

ein ausführbares Programm in der Datei `hallo`, das dann natürlich mit

```
> ./hallo
```

aktiviert wird. "Option" bedeutet "Wahlmöglichkeit". Hier wird mit

```
-o hallo
```

die Option des Compilers genutzt, der Ausgabe einen bestimmten Namen zu geben.

Syntaktische Fehler – Verstöße gegen die Regeln der Programmiersprache

In der Regel wird man es nicht schaffen, ein Programm gleich beim ersten Versuch korrekt einzutippen. Bei Programmen, in denen nicht penibel alle Regeln der Sprache C++ eingehalten werden, verweigert der Compiler die Übersetzung. Vergisst man auch nur ein Komma, oder schreibt ein Komma an eine Stelle, an der ein Semikolon stehen muss, wird das Programm nicht vom Compiler übersetzt. Statt dessen reagiert er mit einer Fehlermeldung.

Nehmen wir an, dass die vierte Zeile in unserem kleinen Beispiel nicht – wie es richtig wäre – mit einem Semikolon, sondern mit einem Komma beendet wird[2]:

[2] Zur Verdeutlichung haben wir hier die Zeilen nummeriert. Die Nummerierung ist natürlich nicht Teil des Quellprogramms

```
1. #include <iostream>
2. using namespace std;
3. int main () {
4.   cout << "Hallo" << endl,
5. }
```

Dann kommt prompt die Quittung in Form einer Fehlermeldung:

```
> g++ hallo.cc
hallo.cc: In function 'int main()':
hallo.cc:5: parse error before '}'
```

Die wesentliche Information dieser Fehlermeldung ist, dass der Compiler *vor* der geschweiften Klammer in Zeile 5 des Programms Probleme hat, die er *parse error* nennt. Damit meint er, dass die *Analyse des Textes* (engl. *to parse* = zergliedern, grammatikalisch analysieren) gezeigt hat, dass der Text nicht den syntaktischen Regeln der Sprache C++ entspricht. Er enthält einen *Syntaxfehler*. Am Anfang sehen solche Fehlermeldungen kryptisch und unverständlich aus. Es ist aber wichtig sich damit zu beschäftigen und sie lesen und verstehen zu lernen.

Semantische Fehler – Inhaltliche Fehler in formal korrekten Programmen

Manchmal wird der von uns eingegebene Quellcode vom Compiler ohne Fehlermeldung übersetzt, aber das erzeugte Programm verhält sich nicht so wie erwartet. Beispielsweise wollen wir, dass "`Hallo`" ausgegeben wird, tatsächlich erscheint aber "`HallX`" auf dem Bildschirm. Man sagt, das Programm enthält einen *semantischen* (inhaltlichen) Fehler. Es tut etwas, aber nicht das was wir von ihm erwarten. In diesem Fall ist die Ursache schnell festgestellt. Wir haben uns sicher einfach nur vertippt und das Programm sieht folgendermaßen aus:

```
1. #include <iostream>     // Ein Programm mit einem sehr
2. using namespace std;    // trivialen semantischen Fehler
3. int main () {
4.   cout << "HallX" << endl;
5. }
```

Im Gegensatz zu vorhin ist hier das Programm auch mit dem Tippfehler syntaktisch korrekt und kann übersetzt werden. Der Compiler kann ja nicht wissen, was wir eigentlich ausgeben wollten. Er hält sich an das was im Quellprogramm steht. Leider sind die semantischen Fehler meist nicht so trivial, wie in diesem Beispiel. Ein semantischer Fehler wird in der Regel nicht durch Tippfehler, sondern durch "Denkfehler" verursacht. Man glaubt das Programm sei korrekt und mache das, was es machen soll, in Wirklichkeit macht es zwar etwas, aber nicht das, was wir wollen, sondern das was wir aufgeschrieben haben und von dem wir nur dachten, es sei das, was wir wollen.

Semantische Fehler sind die ernsten, die richtigen Fehler. Man lässt darum meist die Kennzeichnung "semantisch" weg und spricht einfach von "Fehler". Nur wenn ausdrücklich betont werden soll, dass es sich um etwas so triviales wie einen syntaktischen Fehler handelt, spricht man von einem "Syntax–Fehler".

2.2 Programme: Analyse, Entwurf, Codierung

Programmieren: Analyse, Entwurf, Codierung

Ein Programm soll in der Regel ein Problem lösen oder eine Aufgabe erfüllen. Als erstes muss darum das Problem oder die Aufgabe *analysiert* werden. Als nächstes überlegt man sich wie die Aufgabe gelöst werden soll: man *entwirft* ein allgemeines Vorgehen zur Lösung und ein Programmkonzept. Schließlich muss noch der Quellcode geschrieben werden: das Programm wird *codiert*, oder wie man auch sagt *implementiert*.

Betrachten wir als Beispiel die Umwandlung von Temperaturwerten von Grad Fahrenheit in Grad Celsius.

Analyse

Am Anfang der Analyse wird die *Problemstellung präzise definiert*:

> Das Programm soll eine ganze oder gebrochene negative oder positive Zahl einlesen, sie als Grad–Angabe in Fahrenheit interpretieren und den eingegebenen Wert und den entsprechenden Wert in Celsius ausgeben.

Als nächstes macht man sich *mit dem Problemfeld vertraut*, indem man sich die zur Lösung notwendigen Informationen besorgt. In unserem Fall sagt uns ein elementares Physikbuch wie Fahrenheit in Celsius umgewandelt wird:

$t_C = \frac{(t_F - 32) \cdot 5}{9}$

Entwurf

Nachdem in der Analyse festgestellt wurde, *was* zu tun ist und die zur Lösung norwendigen Informationen besorgt hat, kann man sich jetzt dem *wie* zuwenden: dem Entwurf. Im *Entwurf* wird festgelegt, *wie* das Programm seine Aufgabe konkret lösen soll. Bei einer Berechnungsaufgabe wie in unserem Beispiel wird hier der *Algorithmus* festgelegt:

Algorithmus zur Umwandlung von Fahrenheit in Grad:

1. Grad–Fahrenheit in Variable `f` einlesen
2. Variable `c` mit dem Wert $\frac{(Wert(f) - 32) \cdot 5}{9}$ belegen
3. Wert von `f` und `c` ausgeben

Implementierung

Bei der Implementierung wird der Algorithmus aus dem Entwurf in der korrekten C++–Syntax niedergeschrieben:

```
#include <iostream>

using namespace std;

int main () {
  float f;  // Variable, enthaelt Grad in Fahrenheit
  float c;  // Variable, enthaelt Grad in Celcius

  //Grad in Fahrenheit einlesen:
  cout << "Bitte Grad in F : ";
  cin  >> f;

  //Grad in Celsius berechnen:
  c = ((f - 32) * 5) / 9;

  //Grad in F. und C. ausgeben:
  cout << f << " Grad Fahrenheit ist ";
  cout << c << " Grad Celsius " << endl;
}
```

Programmstruktur

Das Programm beginnt mit der Zeile

```
#include <iostream>
```

Mit ihr wird angekündigt, dass das Programm Daten einlesen und/oder ausgeben will und dass der Compiler darum die entsprechenden Definitionen bereit halten soll. Praktisch jedes Programm beginnt mit dieser Zeile. Mit der Zeile

```
using namespace std;
```

wird angekündigt, dass das Programm Standardfunktionen (hier zur Ein– und Ausgabe) nutzen will und der Compiler darum die entsprechenden Definitionen ohne weitere Umstände akzeptieren soll.. Auch diese Zeile fügen wir zunächst einmal in jedes unserer Programme ein.

Dem folgt die sogenannte *main–Funktion*:

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

In ihr stehen die Anweisungen, die ausgeführt werden sollen. Dabei ist alles hinter `//` ein *Kommentar*. Kommentare werden vom Compiler ignoriert. Sie dienen lediglich dem Verständnis menschlicher Leser.

Die main–Funktion beginnt mit *Variablendefinitionen*:

```
float f;
float c;
```

Hier werden zwei *Variablen* für Fließkommazahlen angelegt (*definiert*). Variablen sind Speicherplätze für Daten.

Die *Ausgabe–Anweisung*

```
cout << "Bitte Grad in Fahrenheit";
```

gibt den Text (die Zeichen innerhalb der Anführungszeichen) aus und die *Eingabe–Anweisung*

```
cin  >> f;
```

liest einen Wert in die folgende Variable (hier `f`) ein. Mit der *Zuweisung*

```
c = ((f - 32) * 5) / 9;
```

wird aus dem Wert in `f` der gesuchte Wert berechnet und in `c` abgelegt.

2.3 Variablen und Zuweisungen

Variablen in der Mathematik: Namen für Werte

Ein zentrales Konzept in C++ (und in fast allen anderen Programmiersprachen) sind die *Variablen*. Eine Variable ist hier als Behälter (Ablageplatz) für (wechselnde) Werte zu verstehen. Auch in der Mathematik spricht man von Variablen. Eine mathematische Variable ist aber etwas anderes als eine Variable in einem Programm! Es ist wichtig sich den Unterschied zwischen mathematischen und Programmvariablen klar zu machen.

Eine mathematische Variable bezeichnet einen Wert, es ist der Name eines Werts. Der Wert kann dabei durchaus beliebig sein. Beispielsweise ist eine Gleichung wie $x = 2*x-10$ eine Aussage, die wahr oder falsch sein kann, je nach dem, welcher Wert x zugeordnet ist. x ist eine Variable, die jeden (reellen) Wert annehmen kann. Für $x = 10$ ist diese Aussage wahr. Für alle anderen Werte von x ist sie falsch. Der Wert von x in der Gleichung ist zwar beliebig, aber er ist *fest*. x kann zwar jeden Wert annehmen, aber nicht links vom Gleichheitszeichen den einen und rechts vom Gleichheitszeichen einen anderen. Innerhalb einer Gleichung oder einer Berechnung ist der Wert einer mathematischen Variablen fest.[3]

Variable im Programm: Behälter für Werte

In einem Programm ist eine Variable statt dessen der *Name eines Behälters* in dem wechselnde Werte liegen können (Siehe Abbildung 3).

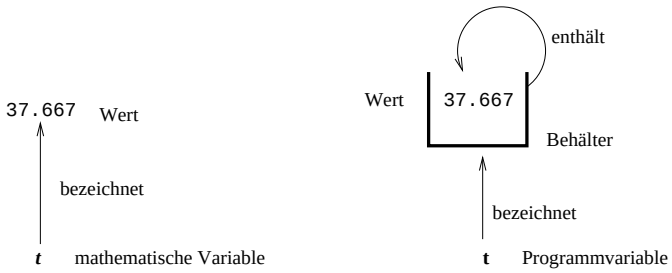


Abbildung 3: Mathematische Variablen und Programmvariablen

Der Unterschied wird an unserem Beispiel leicht klar. Nach Ausführung der Leseanweisung

```
cin  >> f;
```

enthält die Variable `f` einen Wert, beispielsweise $99,8$. Dem Wert können wir einen Namen geben, z.B. t_F. Die *mathematische Variable* t_F ist der Name für den Wert $99,8$. `f` ist der Name für den Behälter (die Programmvariable), der diesen Wert enthält.

[3] Mathematische Puristen mögen einwenden, dass eine Variable eine Lösungsmenge statt eines Wertes repräsentieren kann. Das ist aber eine weitergehende Interpretation der Gleichung als Menge von Aussagen über jeweils einen Wert und ändert nichts an der Tatsache, dass eine Gleichung nur sinnvoll ist, wenn in ihr jeweils jedem Vorkommen der Variablen der gleiche Wert zugeordnet wird.

Die *(Programm–) Variable* ist also ein Behälter für wechselnde Werte. Eine Programmvariable hat mal diesen und mal jenen Wert. `c` enthält auch am Anfang schon einen Wert. Er ist uns aber nicht bekannt, vielleicht ist er `0.0`, vielleicht aber auch `-22561809.001`. Der Wert der Variablen `c` vor der ersten Zuweisung ist ein *Zufallswert*. Nach der Zuweisung

```
c = ((f - 32) * 5) / 9;
```

enthält `c` den Wert $\frac{(t_F-32)\cdot 5}{9}$. Diese mathematische Formel ist dabei auch wieder nur eine Bezeichnung für einen konkreten Wert, z.B. $37,6667$.

Es ist sehr wichtig, sich von Anfang an klar zu machen, dass es Variablen und Werte gibt. Variablen sind in einer Programmiersprache die Namen von Ablageplätzen (Behältern) und nicht wie in der Mathematik die Namen von Werten.

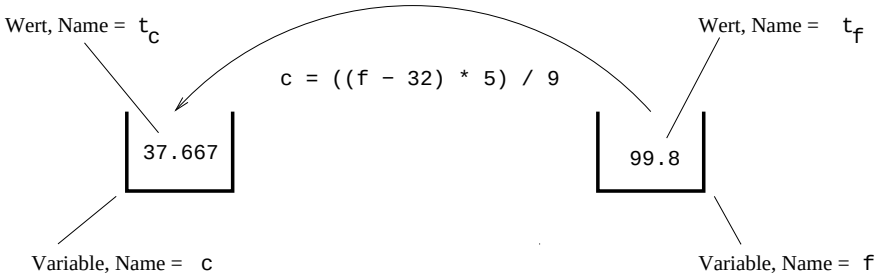


Abbildung 4: Die Zuweisung `c = ((f - 32) * 5) / 9;`

Zuweisung : den Wert (in) einer Variablen ändern

Zuweisungen sehen aus wie Gleichungen in der Mathematik. Es ist ist aber etwas völlig anderes. In der Mathematik ist die Gleichung

$x = 2 \cdot x - 10$

eine Aussage, die für bestimmte Werte von x korrekt ist, für andere nicht. In einem Programm ist

```
x = 2*x - 10;
```

eine *Anweisung*, also eine Aufforderung etwas zu tun. Genauer gesagt, soll der Wert in der Variablen (dem Behälter) `x` verändert werden. Der *neue* Wert von `x` soll auf den doppelten *alten* minus 10 gesetzt werden. Der Wert von `x` ändert sich, während die Anweisung ausgeführt wird, von einem alten Wert – nennen wir ihn x_{alt} (beispielsweise 7) – zu einem neuen Wert x_{neu} (beispielsweise 4). Zwischen den beiden besteht die mathematische Beziehung

$x_{neu} = 2 \cdot x_{alt} - 10.$

Diese mathematische Gleichung beschreibt die Situation nach Ausführung der Zuweisung `x=2*x-10`.

Die Zuweisung	führt zu
`x=2*x-10`	$x_{neu} = 2 \cdot x_{alt} - 10$

Die Zuweisung

```
c = ((f - 32) * 5) / 9;
```

aus unserem Beispiel ist also eine Aufforderung den Wert von `c` von einem alten Zufallswert c_{alt} auf einen neuen Wert $c_{neu} = t_C = \frac{(t_F-32)\cdot 5}{9}$ zu setzen (Siehe Abbildung 4).

Form der Zuweisung

Die *Zuweisung* (engl. *assignment*) hat die allgemeine Form

$< Variable >=< Ausdruck >;$

Die Begriffe in spitzen Klammern – $< Variable >$ und $< Ausdruck >$ – sollen kenntlich machen, dass es sich um irgendeinen *Text* handelt, der vom Compiler als korrekte Notation für eine Variable bzw. einen Ausdruck akzeptiert wird.

Die Bedeutung der Zuweisung ist jetzt klar: Der Wert des Ausdrucks rechts vom Zuweisungszeichen soll berechnet und dann in der Variablen links gespeichert werden (Siehe Abbildung 4). Mit

```
c = ((f - 32) * 5) / 9;
```

etwa wird $t_C = \frac{(t_F - 32) \cdot 5}{9}$ in `c` gespeichert, wenn t_F der aktuelle Wert von `f` ist. Die gleiche Wirkung wird übrigens auch durch folgende Serie von Anweisungen erreicht:

```
c = f;
c = c - 32;
c = c * 5;
c = c / 9;
```

Wertverlaufstabelle

Hier wird `c` schrittweise auf den gewünschten Wert gebracht. Nehmen wir an `f` habe am Anfang den Wert 100, dann sind die einzelnen Schritte:

nach Anweisung	Wert von `f`	Wert von `c`
`c = f;`	100	100
`c = c - 32;`	100	68
`c = c * 5;`	100	340
`c = c / 9;`	100	37,77

Solche kleinen Tabellen, mit denen man Computer spielen und den Verlauf der Wertebelegung von Variablen verfolgen kann, sind sehr nützlich für das Verständnis von Programmen. Meist lässt man dabei allerdings die Anweisungen weg:

`f`	`c`
100	100
100	68
100	340
100	37,77

l–Wert und r–Wert: Variablennamen in Zuweisungen

Variablennamen haben eine leicht unterschiedliche Bedeutung, je nachdem ob sie links oder rechts vom Zuweisungszeichen "=" auftauchen:

- *links* vom Zuweisungszeichen ist der Behälter gemeint: der *l–Wert* der Variable (*lvalue*)
- *rechts* vom Zuweisungszeichen ist der Inhalt des Behälters gemeint, der *r–Wert* der Variable (*rvalue*)

Variablen haben Namen. Die Namen bezeichnen einen l–Wert (Behälter) und in diesem steckt ein r–Wert:

Name → l–Wert (Variable, Behälter) → r–Wert

(Siehe Abbildung 5).

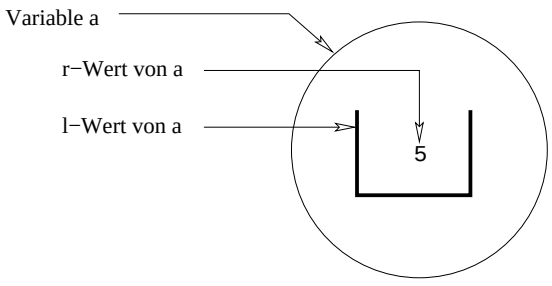


Abbildung 5: Variablen und ihre l– und r–Werte

Nehmen wir an die Variable `a` habe gerade den Wert 5, dann hat in der Anweisung

```
a = 2 * a + 6;
```

`a` auf beiden Seiten einen l–Wert – den Speicherplatz der `a` zugeordnet ist – und einen r–Wert – den Wert 5 . Nur rechts wird der r–Wert von `a` bestimmt. Links reicht der l–Wert. Generell kann ein beliebiger Ausdruck einen l– und/oder einen r–Wert haben. `2*a+6` hat einen r– aber keinen l–Wert. Darum wird

```
2 * a + 6 = a; // FEHLER: non-lvalue in assignment
```

mit einer Fehlermeldung des Compilers quittiert.

Definition einer Variablen: Variable anlegen

Bevor eine Variable benutzt werden kann, muss sie *angelegt* werden. Das geschieht mit der *Variablendefinition*.[4] Zwei Beispiele für eine Variablendefinition sind:

```
float f;
int i;
```

Hier wird gesagt, dass eine Variable `f` und eine Variable `i` angelegt werden sollen. Die möglichen Werte von `f` sind Fließkommazahlen und die von `i` sind ganze Zahlen. Kürzer ausgedrückt: `f` ist eine Fließkomma– oder `float`–Variable und `i` eine Ganzahlige– oder `int`–Variable. `float` und `int` sind die Typen der beiden Variablen, sie bestimmen welche Werte sie aufnehmen können.

Der Typ der Variablen und Werte

Der Typ einer Variablen ist so etwas wie ihre Art. Es gibt verschiedene Arten von Variablen: int–Variablen, float–Variablen, etc., je nachdem wie sie definiert wurden. Der Typ der Variablen hängt eng mit den Werten zusammen, die in ihnen gespeichert werden können. In einer int–Variablen werden beispielsweise nur int–Werte gespeichert, in einer float–Variabeln nur float–Werte. Die Werte sind also auch von von einer bestimmten Art. Sie haben wie die Variablen einen Typ.

Werte und ihre Darstellung (Speicherung) im Rechner muss man unterscheiden. Im Rechner gibt es nur Bits, also 0–en und 1–en. Egal ob es sich um die Zeichenkette “Karl-Eugen Huber”, die Zahl $37,77$ oder ein Bild unseres Hundes handelt, alles sind nur Bits. Der Typ einer Variablen oder eines Wertes unterscheidet darum nicht verschiedene Arten von möglichem Inhalt von Speicherstellen, sondern:

- *wieviele* Bits zu einer Variable (einem Wert) gehören, und
- *wie* die Bits in dieser Variablen (des Wertes) zu *interpretieren* sind.

Eine `float`–Zahl beispielsweise wird mit vielen Bits in einem komplizierten Muster dargestellt, eine `int`–Zahl hat weniger Bits und ihr Bitmuster ist wesentlich leichter zu entschlüsseln. Jede Operation kann nur dann korrekt ausgeführt werden, wenn bekannt ist, welche Art von Wert ein Bitmuster darstellen soll, also welchen Typ die Variable oder der Wert hat.

Eine Variablendefinition soll:

- der Variablen einen Namen geben,
- den nötigen Platz für ihren Wert im Speicher schaffen und
- Auskunft darüber geben, wie der Inhalt dieses Speicherplatzes zu interpretieren ist.

Informationen über die Zahl der Bits oder Bytes, die zu einer Variablen gehören, werden beispielsweise benötigt, wenn ihr Wert kopiert werden soll. Die Interpretation der Bits wird gebraucht, wenn Operationen ausgeführt oder der Wert ausgegeben werden soll.

2.4 Kommentare

Was sind Kommentare

Kommentare sind Bestandteile des Quellprogramms, die vom Compiler vollständig ignoriert werden und die infolgedessen keinen Einfluss auf das Verhalten des Programms haben.

[4] Manche sprechen in diesem Zusammenhang auch von “Deklaration”. Das ist nicht völlig falsch, da jede Definition auch eine Deklaration ist. Zwischen Deklaration und Definition gibt es aber feine Unterschiede auf die wir später noch zu sprechen kommen werden.

Wie sehen Kommentare aus

Kommentare können in C++ zwei Formen annehmen:

1. `// Dies ist ein Kommentar bis zum Zeilenende`
2. `/* Dies ist ein Kommentar */`

Alles hinter einem doppelten Schrägstrich bis zum Zeilenende und alles zwischen `/*` und `*/` ist Kommentar.

Wozu dienen Kommentare

Kommentare sind ein wichtiger Teil eines Programms. Sie machen nicht nur anderen dessen Bestandteile und Funktionen klar, sondern auch dem Autor selbst. Jeder, der einmal ein auch nur wenige Wochen altes Programm modifizieren musste, wird das sofort verstehen. Gelegentlich hat man – speziell bei der Fehlersuche – schon Probleme wenige Stunden alte eigene Konstrukte zu verstehen. Für professionelle Programmierer, die oft viele Tausende von Zeilen Code zu pflegen haben, kann eine vernünftige Kommentierung zur Überlebensfrage werden.

Beispiele für sinnvolle Verwendungen von Kommentaren sind:

- Am Anfang einer Quelldatei: Informationen zum Programm und dessen Autoren;
- Bei Variablendefinitionen: Zweck der Variablen;
- Vor Anweisungen: Ziel der Anweisungen;
- Nach Anweisungen: Von den Anweisungen erreichte Belegung der Variablen.

Beispiel

Ein Beispiel ist:

```
/* Datei:  temperatur.cc
   Autor:  Charlotte Programmatrise
   Datum:  12-Jan-2000
   Beschr: Wandelt Grad Fahrenheit in Grad Celsius um.
           Aufruf:  Programm-Name
           Eingabe: Grad Fahrenheit als positive oder negative float-Zahl
           Ausgabe: entsprechende Grad Celsius als float-Zahl
*/
#include <iostream>

using namespace std;

int main () {
  float f;  // Temperatur in Grad Fahrenheit
  float c;  // Temperatur in Grad Celcius

  // Grad in Fahrenheit einlesen:
  cout << "Bitte Grad in F : ";
  cin  >> f;

  // f: Temperatur
  // c: undefiniert

  //Grad in Celsius berechnen:
  c = ((f - 32) * 5) / 9;

  /* f enthaelt die eingegebene Temperatur in Grad Fahrenheit,
     c enthaelt die gleiche Temperatur in Grad Celsius
  */

  //Temperatur in Grad Fahrenheit und Celsius ausgeben:
  cout << f << " Grad Fahrenheit entspricht ";
  cout << c << " Grad Celsius " << endl;
}
```

Die Kommentierung offensichtlicher Dinge sollte man natürlich unterlassen. Ein negatives Beispiel ist:

```
x = x + 1; // x wird um 1 erhoeht
```

Schlimmer als fehlende und überflüssige sind nur noch falsche Kommentare.

```
x = 2*x; // x wird um 1 erhoeht
```

2.5 Die Inklusions–Direktive

Die Inklusions–Direktive

Unsere beiden Programme begannen jeweils mit einer *Inklusions–Direktive* (auch: “Include–Direktive”, “Include–Anweisung”):

```
#include <iostream>
```

Mit ihr wird der Compiler angewiesen vor unserem Programm noch eine andere – mit `<iostream>` bezeichnete – Datei zu lesen. In dieser Datei finden sich Deklarationen zur Ein– und Ausgabe die in unserem Programm verwendet werden (Z.B. `cin` und `cout`).

Include–Datei

Mit dem Inhalt der *Include–Datei* oder, wie man auch sagt, der *Header–Datei* `<iostream>` wollen wir uns hier noch nicht befassen. Es reicht vorerst zu wissen, dass sie am Anfang jedes Programms *inkludiert* werden muss, wenn dieses Daten lesen und/oder schreiben will. Neben `<iostream>` stellt das System noch viele andere Include–Dateien zur Verfügung. Diese Kollektion kann auch noch durch eigene Beiträge erweitert werden.

Include–Datei für Zeichenketten (Strings)

Viele Dinge sind in C++ ohne weitere Umstände verfügbar, da sie direkt “in die Sprache eingebaut sind”. Bei anderen muss vorher eine Header–Datei inkludiert werden. Die Eingabe und Ausgabe sind Beispiele für letzteres. Ein anderes sind *Zeichenketten*, oder *Strings* wie man oft sagt.

`string` ist wie `float` ein möglicher Typ einer Variable. Eine `string`–Variable enthält als Wert Zeichenketten, also Folgen von Zeichen. `string` kann aber nur benutzt werden, wenn vorher `<string>` inkludiert wurde. Ein Beispiel für ein Programm das mit Zeichenketten hantiert ist:

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

using namespace std;

int main () {
  string gruss;
  cout << "Bitte gruesse mich " << endl;
  cin >> gruss;
  cout << "Ja Dir auch ein " << gruss << endl;
}
```

Wird dieses Programm in einer Datei namens `gruss.cc` gespeichert, mit

```
> c++ -o gruss gruss.cc
```

übersetzt und mit `./gruss` aktiviert, dann gibt es die Eingabe mit einem `Ja Dir auch ein` zurück:

```
> gruss
Hallo
Ja Dir auch ein Hallo
>
```

2.6 Datentypen: Zeichen, Zeichenketten, ganze und gebrochene Zahlen

Datentyp

Mit dem Begriff *Datentyp* (kurz *Typ*) bezeichnet man Arten von Werten und Variablen. Jede Variable hat einen (Daten–) Typ. Er bestimmt welche Art von Wert die Variable aufnehmen kann. C++ kennt viele Typen und bietet dem Programmierer noch dazu die Möglichkeit beliebig viele dazu zu erfinden. Einige wichtige Datentypen sind:

- `float` : gebrochene Zahl
- `int` : ganze Zahl
- `char` : Zeichen
- `string` : Zeichenkette

Ein `char`–Wert ist ein einzelnes Zeichen beliebiger Art: Buchstabe, Zahl, Sonderzeichen. Ein `string`–Wert besteht aus einer Aneinanderreihung einzelner Zeichen.

Ein Beispiel

Ein etwas ausführlicheres Beispiel für die Verwendung dieser Typen in einem Programm ist:

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

using namespace std;

int main () {
  string vorname,
         nachname,
         name;
  char   initial_1,
         initial_2;
  int    l_vorname;

  cout << "Vorname: ";
  cin >> vorname;
  cout << "Nachname: ";
  cin >> nachname;

  name      = vorname + nachname;
  initial_1 = vorname.at(0);  // Hallo vorname, was ist bei Dir
                              // auf Position 0 ?
  initial_2 = nachname.at(0);
  l_vorname = vorname.length();  // Hallo vorname, wie lang bist Du?

  cout << "Hallo " << name << endl;
  cout << "Dein Initial ist " << initial_1 << initial_2 << endl;
  cout << "Dein Vorname hat " << l_vorname << " Zeichen" << endl;
}
```

Dieses Programm liest einen Namen und einen Vornamen – zwei Zeichenketten – ein und bildet daraus mit der Anweisung

```
name = vorname + nachname;
```

den gesamten Namen. Das +–Zeichen bedeutet hier keine Addition von Zahlen, sondern die *Verkettung* von zwei Zeichenfolgen zu einer neuen. Mit

```
initial_1 = vorname.at(0);
```

wird das Zeichen an Position 0 von `vorname` (`vorname.at(0)`) extrahiert und der Variablen `initial_1` zugewiesen. Mit

```
initial_2 = nachname.at(0);
```

wird der Anfangsbuchstabe des Nachnamens festgestellt und in `initial_2` gespeichert. Die Anweisung

```
l_vorname = vorname.length ();
```

bestimmt die Länge der Zeichenkette in der Variablen `vorname` und speichert sie in `l_vorname`. Schließlich werden die so bestimmten Werte ausgegeben.

Mit `at` und `length` kann man auf String–Variablen zugreifen. Man sagt `at` und `length` sind *Methoden* des Datentyps `string`.

Typpedanterie

C++ ist, wie viele andere Programmiersprachen auch, etwas pedantisch mit seinen Typen. Im Gegensatz zur allgemeinen Intuition und zur Mathematik muss man sich vor allem merken, dass ein `char` *keine* Zeichenkette der Länge 1 ist, und dass ein `int`–Wert (eine "ganze Zahl") *nicht* gleichzeitig auch ein `float`–Wert ist! `int`–Werte sind also keine Teilmenge der `float`–Werte und `char`–Werte sind keine Teilmenge der `string`–Werte.

Die Konsequenz dieser strikten Unterscheidung ist, dass die Zuweisung an `s` in

```
...
char   c;
string s;
c = 'A';
s = c;    //FALSCH: Typfehler
...
```

verboten ist und dass die Zuweisung an `f` in

```
...
int    i;
float  f;
i = 1;
f = i;    //Konversion
...
```

dazu führt, dass der Wert von `i` konvertiert (umgerechnet) werden muss: aus den Bits in `i`, die in der `int`–Darstellung eine "1" bedeuten, muss das Bitmuster erzeugt werden, das in der `float`–Darstellung eine "1" bedeutet.

2.7 Mathematische Funktionen

Die Mathematik–Bibliothek

Viele nützliche Dinge muss man als Programmierer nicht selbst schreiben. Andere haben das schon vorher getan. C++ kommt sogar mit einem sehr reichhaltigen Angebot an fertigen Programmteilen. Neben den schon bekannten Routinen zur Ein– und Ausgabe stehen etliche *mathematische Funktionen* zur Verfügung, wenn die Include–Datei `math` eingebunden wird. Ein Beispielprogramm das die Quadratwurzel einer eingegebenen Zahl zieht:

```
#include <iostream>  // Ein-/Ausgabe
#include <string>    // Strings (Zeichenketten)
#include <cmath>     // Mathematische Funktionen

using namespace std;

int main () {
  float  z;  // Eingelesene Zahl
  float  w;  // Wurzel von z

  // Deklaration und Zuweisung in einem:
  string gruss = "Hallo! Bitte positive Zahl eingeben: ";

  cout << gruss;
  cin >> z;

  // Wurzel ziehen:
  w = sqrt (z);
```

```
  cout << "Die Wurzel von " << z << " ist " << w << endl;
}
```

Hier wird mit

```
w = sqrt (z);
```

die Quadratwurzel aus dem aktuellen Wert von `z` wird gezogen und in `w` gespeichert. Das Programm enthält auch eine interessante Variante der Variablendefinition. Mit

```
string gruss = "Hallo! Bitte positive Zahl eingeben: ";
```

wird die Variable `gruss` definiert und gleichzeitig mit einem Wert (einer Zeichenkette) belegt. So etwas ist prinzipiell bei der Definition jeder Variablen möglich.

In einer Variablendefinition kann man auch gleichzeitig mehrere Variablen definieren. Beispiel:

```
...
int     i  = 12, j = 13, k;
string  s1 = "Hallo",
        s2 = "Tschau",
        s3;
float f, g, pi = 3.1415926;
...
```

Man beachte hier die Verwendung von Komma und Semikolon. Compiler sind sehr streng in diesen Kleinigkeiten. Ein Komma trennt Variablen in einer Definition. Das Semikolon beendet sie.

2.8 Konstanten

Konstanten

Konstanten sind Variablen, die ihren Wert nicht ändern können. Sie werden bei der Definition mit dem *Schlüsselwort* `const` markiert und natürlich gleich mit einem Wert belegt. Die allgemeine Form einer *Konstantendefinition* ist

`const` $< Typ > < Bezeichner > = < Wert >;$

Beispiele:

```
const float pi = 3.1415;
const string gruss = "Hallo Leute";
```

`const` wird Schlüsselwort (auch "reserviertes Wortsymbol") genannt, da seine Bedeutung im Programm nicht geändert werden darf. Z.B. darf keine Variable mit dem Namen `const` angelegt werden.

```
float const = 17.8; // FALSCH: const ist Schluesselwort
```

Beispielprogramm mit Konstanten

Berechnung der Fläche eines Kreises:

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

using namespace std;

int main () {
  float  r;  // Eingelesener Radius
  float  f;  // Berechnete Flaeche;
  const float pi = 3.1415;  // Konstante, unveraenderlich

  // Variable fuer Zeichenkette anlegen
  // und mit Wert belegen:
  const string gruss = "Hallo! Bitte Radius eingeben: ";
```

```
  cout << gruss;
  cin >> r;

  // Flaeche berechnen:
  // (pow (x,y) berechnet x hoch y. pow ist in cmath deklariert)
  f = pi * pow (r, 2);

  cout << "Die Flaeche bei Radius " << r << endl;
  cout << " ist " << f << endl;
}
```

Hier sehen wir auch mit `pow` die Benutzung einer weiteren Funktion aus `cmath`.

Sinn der Konstanten

Konstanten verdeutlichen die Absichten des Programmierers und dienen dem eigenen Schutz. Im Beispiel oben ist `pi` dazu gedacht den (angenäherten) Wert von π zu enthalten. Die Definition mit `const` macht nicht nur klar, dass `pi` einen unveränderlichen Wert enthalten soll, sie *verhindert auch jede Veränderung* des Wertes. Der Compiler quittiert einen entsprechenden Versuch mit einer Fehlermeldung:

```
const float pi = 3.1415;
...
pi = pi + 0.1; // FEHLER !!!
...
```

2.9 Eingabe, Ausgabe und Dateien

Standardeingabe und Standardausgabe: `cin` und `cout`

Ein Programm, das mit seiner Umwelt keine Daten austauscht, ist nutzlos. Jede Programmiersprache hat darum Konstrukte, die ihren Programmen ermöglichen Daten zu lesen und zu schreiben. Mit der Ein– und Ausgabe wird die geschlossene Welt des Programms verlassen. Das birgt allerdings auch eine gewisse Problematik. Ein C++–Programm soll im Prinzip auf jedem Rechner laufen können: auf einem PC genauso wie auf einem Großrechner und, da Programmiersprachen erfahrungsgemäß eine sehr lange Lebensdauer haben, werden C++Programme eventuell auf Rechner laufen, die mit ihren Benutzern per Gedankenübertragung kommunizieren.

Um all diese gegenwärtig oder zukünfig möglichen Interaktionen des Programms mit dem Rest der Welt unter einen einfachen Hut zu bringen, hat man in C++ das Konzept der *Standardeingabe* und der *Standardausgabe* eingeführt. Mit der Anweisung

```
cin >> v;       // Lesen von der Standardeingabe
cout << v;      // Schreiben auf die Standardausgabe
```

wird ein Wert für die Variable `v` von der Standardeingabe gelesen und anschließend auf die Standardausgabe ausgegeben. Die Definition von C++ lässt dabei völlig offen, was genau die Standardeingabe und die Standardausgabe ist. Es ist Sache der jeweiligen Implementierung diese Begriffe mit einer genauen Bedeutung zu belegen. Typischerweise ist die Tastatur die Standardeingabe (`cin`) und der Bildschirm, bzw. das Fenster in dem das Programm aktiv ist, die Standardausgabe (`cout`).

Standardfehlerausgabe: `cerr`

Neben `cout` definiert C++ eine zweite Standardausgabe: `cerr`, die *Standardfehlerausgabe*. In ihr sollten Fehlermeldungen des Programms ausgegeben werden. Beispiel:

```
cerr << "Falsche Eingabe, bitte einen positiven Wert eingeben" << endl;
```

In der Regel werden Ausgaben auf `cout` und `cerr` auf das gleiche Medium ausgegeben, den Bildschirm. Allerdings kann beides unabhängig voneinander in andere Medien gelenkt werden. Dies liegt jedoch ausserhalb der Sprache und des Programms.

Ausgabeformatierung

Die Gestaltung – das Format – der Ausgabe kann in vielfältiger Weise beeinflusst werden. Mit der Ausgabe von `endl` (lies *end–line*) wird beispielsweise ein Zeilenvorschub erzeugt. Die Genauigkeit der Ausgabe kann beispielsweise mit `precision` gesetzt werden und mit `width` wird die Zahl der ausgegebenen Zeichen angegeben. Beispielsweise erzeugt das Programm

```
#include <iostream>
#include <cmath>

using namespace std;

int main () {
  float x = 2;
  float y = sqrt(x);

  cout << "|";
  cout.width(6);      // Naechste Ausgabe (die von x) auf 6 Zeichen
  cout << x << "|";
  cout.precision(6); // Naechste Ausgabe: Genauigkeit 6 Stellen
  cout.width(8);      // Naechste Ausgabe (die von y) auf 8 Zeichen
  cout << y << "|" << endl;

  cout << "Zum Vergleich ohne Formatangaben:" << endl;
  cout << "|" << x << "|" << y << "|" << endl;
}
```

die Ausgabe

```
|     2| 1.41421|
Zum Vergleich ohne Formatangaben:
|2|1.41421|
```

Die Ausgabe von `x` wird auf 6 Zeichen verteilt und die von `y` auf 8. Füllzeichen ist dabei jeweils das Leerzeichen. `y` wird mit 6 Stellen Genauigkeit angezeigt.

Ausgabe in eine Datei

Programme können ihre Daten auch in Dateien ausgeben. Das ist dann nützlich, wenn die Ausgabe aufbewahrt werden soll, oder wenn eine umfangreiche Ausgabe in Ruhe studiert werden soll. Folgendes Programm gibt beispielsweise seine Daten in eine Datei aus:

```
#include <fstream>    // <- fuer die Arbeit mit Dateien --
#include <cmath>

using namespace std;

int main () {
  float x = 2;
  float y = sqrt(x);

  ofstream ausgabe("Daten.txt");    // Die Datei "Daten.txt" ist Ausgabedatei

  ausgabe << x << " " << y << " " << endl;    // Ausgabe in eine Datei
}
```

Die Daten des Programms werden hier nicht in die Standardausgabe, sondern in die Datei mit dem Namen "`Daten.txt`" ausgegeben. Sollte die Datei bereits existieren, dann wird ihr bisheriger Inhalt vor der ersten Ausgabe gelöscht. Existiert die Datei noch nicht, dann wird sie erzeugt. Die vom Programm erzeugte und beschriebene Datei kann anschließend mit einem Texteditor betrachtet werden.

Eingabe von einer Datei

Ein Programm kann seine Daten nicht nur in eine Datei einlesen, es kann auch aus einer Datei lesen. Das folgende Programm liest die Daten, die von dem vorherigen geschrieben wurden. Die Datei "`Daten.txt`" ist jetzt

Eingabedatei.

```
#include <iostream>
#include <fstream>

using namespace std;

int main () {
  float x;
  float y;
  ifstream eingabe("Daten.txt");     // Die Datei "Daten.txt" ist Eingabedatei
  eingabe >> x >> y;                 // Zwei Werte von Datei einlesen
  cout << x << ", " << y << endl;    // Ausgabe auf Standardausgabe
}
```

Dateien werden in Teil 2 genauer behandelt.

2.10 Zusammenfassung: Elemente der ersten Programme

- Struktur eines Programms

  ```
  #include <...>  // Inklude-Direktiven
  ...             //

  using namespace std; // using Anweisung

  int main () {   // Startpunkt der Ausfuehrung
    ... Variablendefinitionen ...
    ... Anweisungen ...
  }
  ```

- Allgemeine Form der Definition von Variablen

  <Typ> *<Variablenname>* `=` *<Wert>*`,` *<Variablenname>* `=` *<Wert>*`, ... ;`

  oder einfacher

  <Typ> *<Variablenname>*`, ` *<Variablenname>*`, ... ;`

  oder ganz einfach:

  <Typ> *<Variablenname>*`;`

- Definition einer Konstanten

  `const` *<Typ>* *<Bezeichner>* `=` *<Wert>*`;`

- Zuweisung

  <Variablenname> `=` *<Ausdruck>*`;`

- Text und Zahlen einlesen

  `cin >>` *<Variablenname>*`;`

- Text und Zahlen ausgeben

  `cout <<` *<Ausdruck>*`;`
  `cout <<` *<Ausdruck>* `<<` *<Ausdruck>* `<< ... ;`

- Eingabedatei definieren und benutzen

  `ifstream eingabe("`*<Dateiname>*`");`
  `eingabe >> x;`

- Ausgabedatei definieren und benutzen

  `ofstream ausgabe("`*<Dateiname>*`");`
  `ausgabe << x;`

- Einbindung von Include–Dateien

  `#include <...>`

- Datentypen:

  - `float` : Gebrochene Zahlen
  - `int` : ganze Zahlen
  - `char` : Zeichen
  - `string` : Zeichenketten

- Funktionen und Methoden verwenden

  - Funktion auf Werte (eventuell von Variablen) anwenden:
    <Funktion> `(`*<Argumentliste>*`)`
    Beispiel: `pow (r, 2)`
    liefert einen Wert: die 2–te Potenz des aktuellen Wertes von `r`
  - Variable mit Hilfe einer ihrer Methoden befragen:
    <Variable>`.`*<Methodenname>* `(`*<Argumentliste>*`)`
    Beispiel: `s.at (2);`
    liefert das dritte (!) Zeichen des aktuellen Wertes (eine Zeichenkette) von `s`
    Beispiel: `s.length ();`
    liefert die Länge des aktuellen Wertes (eine Zeichenkette) von `s`

2.11 C++ und C

Historie von C++

Die Entwicklung von C++ begann in den frühen 80er Jahren als experimentelle Erweiterung der damals etwa 10 Jahre alten Programmiersprache C. Zunächst sehr eng mit dem Betriebssystem Unix verbunden, war C in jenen Jahren die dominierende Programmiersprache. Mit der objektorientierten Euphorie in den 80ern begannen die Hersteller von Compilern Sprachelemente von C++ in ihre C–Compiler zu integrieren. C++ war damals eine Sammlung recht unausgereifter und oft inkompatibler Eweiterungen der stabilen und weit verbreiteten “richtigen” Programmiersprache C.

1989 begann ein internationales Kommitee mit der Standardisierung von C++. Parallel dazu sammelten mutige (und auch einfältige) OO–Gläubige erste Erfahrungen mit ernsthaften Anwendungen in C++. Im Jahre 1998 lag dann endlich eine offizielle Sprachdefinition als ISO–Standard vor. In den neun Jahren wurden viele der frühen Konzepte weiterentwickelt und in einen konsistenten Zusammenhang gebracht. C++ ist damit heute eine sehr ausgereifte und gleichzeitig auch sehr moderne Programmiersprache. Die Entwicklung von C++ ist sicher noch nicht am Ende und das eine oder andere Sprachelement mag verbesserungswürdig sein und verbessert werden. Die Sprache ist jetzt aber im wesentlichen stabil und unterscheidet sich stark von früheren unausgereiften Versionen.

C++ ist eine große Sprache, gedacht für professionelle Software–Entwickler, die ein mächtiges Werkzeug benötigen und im Allgemeinen wissen, was tun ist. Es ist auch eine schöne und elegante Sprache. Vielleicht ist das auch darum so, weil Entwicklung und Standardisierung sehr wesentlich auf der Arbeit von Freiwilligen basieren, die oft mit nichts als der Freude an interessanten Projekten belohnt wurden. Mächtige Softwarekonzerne, die viel Geld mit C++–Compilern verdienen, haben eher wenig zur Entwicklung der Sprache beigetragen. Ihre Compiler sind darum auch oft nicht auf dem aktuellen Stand.

C–Kompatibilität von C++

C++ wurde als Erweiterung von C konzipiert. C–Programmierer sollten damit einen erleichterten Zugang zu C++ erhalten. Im Wesentlichen ist dieser Ansatz beibehalten worden und man kann C als Teilmenge von C++ betrachten. Ein C–Programm ist in der Regel auch ein korrektes C++-Programm. Die Ausnahmen von dieser Regel sind seltsam oder von schlechtem Stil.

C ist zwar “der einfache Teil” von C++. Das heißt aber nicht, dass mit C alles einfacher ist – im Gegenteil. So wie man mit einem Schweizermesser Dosen leichter öffenen kann, als mit einem Küchenmesser, einfach weil es auch einen Dosenöffner hat, so sind etliche elementare Probleme mit C++ leichter zu lösen als mit C. Der wichtigste Bereich ist zunächst die Ein– und Ausgabe.

Ein– und Ausgabe in C

In C++ werden Daten mit Hilfe von Definitionen aus `iostream` eingelesen und ausgegeben (z.B. `cin` und `cout`). `iostream` ist nicht Bestandteil von C. Als Beispiel vergleichen wir ein C– und ein C++–Programm zur Konversion von Temperaturangaben.

C++:

```
#include <iostream>

using namespace std;

int main () {
  float f;
  float c;

  cout << "Bitte Grad in F : ";
  cin  >> f;

  c = ((f - 32) * 5) / 9;

  cout << f << " G Fahr. = ";
  cout << c << " G Celsius " << endl;
}
```

C:

```
#include <stdio.h>


int main () {
  float f;
  float c;

  printf ("Bitte Grad in F : ");
  scanf ("%f", &f);

  c = ((f - 32) * 5) / 9;

  printf (" %f G Fahr. = ", f);
  printf (" %f G Celsius\n", c);
}
```

Mit den Details der Ein– und Ausgabe in C wollen wir uns hier nicht beschäftigen. Der wesentliche Unterschied ist, dass in C spezielle Funktionen benutzt werden müssen, über deren Arbeitsweise man Bescheid wissen muss. Eine rein intuitive Bedienung wie in C++ ist nicht möglich.

Zeichenketten

Der Datentyp `string` von C++ stellt viele nützliche Operationen zur Verarbeitung von Zeichenketten zur Verfügung. C kennt diesen Typ nicht und zur Textverarbeitung stehen nur die elementaren *C–Strings* zur Verfügung. (C–Strings können und werden auch in C++ genutzt.) Ein kleines Beispiel soll auch hier den Unterschied klarmachen:

C++:

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

using namespace std;

int main () {
  string name,
         vorname,
         nachname;

  cout << "Vorname: ";
  cin  >> vorname;

  cout << "Nachname: ";
  cin  >> nachname;

  name = vorname + " " + nachname;


  cout << "Name = " << name
       << " hat " << name.length()
       << " Buchstaben" << endl;
}
```

C:

```
#include <stdio.h>
#include <string.h>

int main () {
  char name    [32],
       vorname [32],
       nachname[64];

  printf ("Vorname: ");
  scanf ("%s", vorname);

  printf ("Nachname: ");
  scanf ("%s", nachname);

  strcpy (name, vorname);
  strcat (name, " ");
  strcat (name, nachname);

  printf ("Name = %s", name);
  printf (" hat %d ", strlen (name));
  printf (" Buchstaben\n");
}
```

Auch hier wollen wir uns nicht mit Details von C beschäftigen. Wer sich näher für die verwendeten Funktionen interessiert, kann sich mit dem `man`–Kommando nähere Auskunft geben lassen. Z.B. gibt

```
> man strcat
```

eine Beschreibung der `strcat`–Funktion aus. (“>” ist hier der Prompt und wird nicht eingetippt.) Die C–Funktionen stehen auch unter C++ zur Verfügung, die C–Programme sind darum auch korrekte C++–Programme.

Die Standardbibliothek und Header–Dateien

Eine große Zahl von Definitionen sind nicht Bestandteil der Sprache selbst, müssen aber mit jeder C++–Implementierung mitgeliefert werden. Man bezeichnet sie als *Standardbibliothek* (engl. *Standard Library*). Diese “Bibliothek” spielt eine bedeutende Rolle bei der Nutzung von C++, da sie die Programmierarbeit wesentlich erleichtern kann. Die Definitionen der Standardbibliothek müssen durch Includeanweisungen dem Programm zugänglich gemacht werden. Die folgenden Beispiele sind uns bereits bekannt:

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

Die Standardbibliothek von C++ geht weit über die Standardfunktionen von C hinaus. Aber auch C hat seine Standardbibliothek, die ebenfalls mit Includeanweisungen in ein C–Programm eingeführt werden kann. Auch hierzu haben wir oben Beispiele gezeigt:

```
#include <stdio.h>
#include <string.h>
```

Inklusionsdateien für Standardfunktionen von C haben stets die Dateiendung `.h`. Inklusionsdateien für C++ dagegen haben keine Dateiendung. Inklusionsdateien für Standardfunktionen von C können auch in C++ verwendet werden – C++ ist ja eine Erweiterung von C. Zu jeder Standardinklusionsdatei von C gibt es eine C++–Variante ohne Endung und mit vorangestelltem “`c`”:

```
C-Programm:                C++-Programm:

#include <stdlib.h>        #include <cstdlib>
#include <stdio.h>         #include <cstdio>
#include <string.h>        #include <cstring>
#include <math.h>          #include <cmath>
                           #include <string>    //keine C-Variante
                           #include <iostream>  //keine C-Variante
```

Man beachte, dass `string` C++–Strings definiert und diese etwas völlig anderes darstellen als die in `string.h` bzw. `cstring` definierten C–Strings.

Auch in C++ können Headerdateien mit der Endung `.h` verwendet werden. Bestandteile der Standardbibliothek – sei es die C++– oder die C–Standardbibliothek – sollten jedoch in der endungsfreien Version verwendet werden.

2.12 Übungen

Bitte bearbeiten Sie alle Übungen! Sehen Sie sich die Lösungshinweise erst nach einem Lösungsversuch oder einem *längeren* erfolglosen Nachdenken an. Als erstes sollten Sie sich mit der Entwicklungsumgebung vertraut machen, also lernen wie ein Programm eingegeben, übersetzt und ausgeführt wird. Als nächstes müssen Sie lernen eigenständig einfache Programme zu schreiben. Erfahrungsgemäß ist es gelegentlich wenig erfreulich, erfolglos über ein Problem nachzudenken und die Versuchung liegt nahe, das Denken zu schnell durch Nachlesen zu ersetzen. Dieser Versuchung müssen Sie unter allen Umständen wiederstehen. Wie alle die schreiben, müssen auch Programmierer es aushalten, eine gewisse Zeit von einem leeren Blatt Papier oder einem leeren Bildschirm angestarrt zu werden. Studieren Sie die Beispiele im Skript, klappen Sie dieses und alle Bücher zu, legen Sie alle Unterlagen weg und versuchen Sie dann die Lösung der Aufgabe selbst zu finden.

Aufgabe 1

1. Schreiben sie ein Programm, dass die Zeichenkette “Hallo” ausgibt, übersetzen Sie es und führen Sie das übersetzte Programm dann aus.
2. Erklären Sie an einem Beispiel den Unterschied zwischen syntaktischen und semantischen Fehlern.
3. Schreiben Sie ein Programm, das zwei Zahlen einliest und ihre Summe und ihr Produkt ausgibt.
4. `f1, f2` und `f3` sind `float`–Variablen mit bestimmten aktuellen Werten. Geben Sie Zuweisungen an, mit denen erreicht wird, dass `f2` das doppelte und `f3` das vierfache des aktuellen Werts von `f1` enthalten.
5. Haben die im folgenden nebeneinander gestellten Anweisungen stets die gleiche Wirkung (alle Variablen sind vom Typ `int`, ohne Beachtung von einem eventuellen Überlauf)?

```
a = (b - c)*2;        a = b;
                      a = a - c;
                      a = a*2;

a = (b - a)*2;        a = b;
                      a = a - a;
                      a = a*2;

a = (b - c)*2;        a = b*2;
                      a = a - c*2;

a = (b - c)*2;        b = b*2;
                      a = b - c*2;
```

Aufgabe 2

Einige wichtige Operationen auf Zeichenketten (strings) sind (`z`, `z1` und `z2` seien Zeichenketten; (`string z, z1, z2;`), `i` eine ganze Zahl (`int i;`) `c` ein Zeichen (`char c;`)):

`i= z.length()`	liefert die Länge von `z` als `int`–Wert
`c = z.at (p)`	liefert Zeichen an Position `p` in `z`
`z = z1 + z2;`	`z` ist die Verkettung von `z1` und `z2`

Schreiben Sie ein Programm das Vor– und Nachnamen jeweils als Zeichenkette einliest und beide zu einem Namen zusammensetzt und dann den Namen, die Initialen und die Länge des Vor– und des Nachnamens ausgibt.

Aufgabe 3

1. Programme enthalten manchmal Fehler. Welche Fehler findet der Compiler, welche muss die/der Programmierer(–in) finden?
2. Was bedeuten die folgenden Begriffe:
   - Analysieren
   - Übersetzen

- Entwerfen
- Editieren

Wer führt diese Tätigkeiten mit welchem Ziel aus?

3. Vergleichen Sie den Gebrauch der Begriffe “Variable” und “Konstante” in der Mathematik, Physik und der Programmierung.

4. Was bedeutet:
   $x = 2 * x - 5$
   in der Mathematik und in C++?

5. C++ ist eine höhere Programmiersprache. Was bedeutet das? Wie unterscheidet sich eine höhere von einer nicht so hohen Programmiersprache?

6. Ein Compiler wird auch “Übersetzer” genannt. Was übersetzt er in was? Wo findet er das, was er übersetzen soll und wo schreibt er das Ergebnis des Übersetzungsvorgangs hin?

7. Ist C++ eine Maschinensprache? Was ist das überhaupt?

8. In einem Programm kommen folgende Zeilen vor:

```
int a, b;
float x, y;
a = 1; b = 2;
x = 1; y = 2;
a = a+b;
x = x+y;
```

   Erläutern Sie informal was der Compiler damit macht und was dann mit dem Ergebnis der Compileraktivitäten weiter passiert.

9. Wer muss wissen wieviele Bytes (= Gruppen von 8 Bits) eine `float`–Zahl im Speicher belegt: Die/der Programmierer(in), der Compiler, die Hardware (welche Komponente der Hardware)? Wer legt es fest: Bill Gates, Bill Clinton, die Firma Intel, eine DIN–Norm, jeder wie er will, ..?

2.13 Lösungshinweise

Aufgabe 1

1. Bei Problemen lesen Sie bitte das Skript, besuchen Sie das Praktikum.
2. Siehe Skript!
3. Ein Programm, das zwei Zahlen einliest und ihre Summe und ihr Produkt ausgibt:

```
#include <iostream>

using namespace std;

int main () {
  int x, y;
  int s, p;

  cout << "Bitte 1-te Zahl eingeben ";
  cin >> x;

  cout << "Bitte 2-te Zahl eingeben ";
  cin >> y;

  s = x + y;
  p = x * y;

  cout << x << " + " << y << " = " << s << endl;
  cout << x << " * " << y << " = " << p << endl;
}
```

   Beachten Sie hier vor allem, wie Werte eingelesen und die Werte von Variablen zusammen mit kommentierendem Text ausgegeben werden. Mit

   ```
   cin >> x;
   ```

   wird ein ganzzahliger Wert eingelesen und in der Variablen `x` abgelegt. Mit

   ```
   cout << x << " + " << y << " = " << s << endl;
   ```

   werden drei Werte ausgegeben: Die Werte der Variablen `x`, `y` und `s`. Zu diesen Werten kommt der kommentierende Text. Insgesamt entsteht eine Ausgabezeile nach fogendem Muster:

   Wert_von_x + Wert_von_y = Wert_von_s <Zeilenvorschub>

   Die Ausgabe von `endl` erzeugt einen Zeilenvorschub. Bei der Eingabeaufforderung

   ```
   cout << "Bitte 1-te Zahl eingeben ";
   ```

   haben wir auf den Zeilenvorschub verzichtet: Der geforderte Wert wird auf der gleichen Zeile ausgegeben, wie die Aufforderung ihn einzugeben.

4. `f1, f2, f3` und `f1` sind `float`–Variablen. Geben Sie Zuweisungen an, mit denen erreicht wird, dass `f2` das doppelte und `f3` das vierfache des aktuellen Werts von `f1` enthalten.

   Eine Möglichkeit wäre:

   ```
   f2 = f1 * 2; f3 = f1 * 4;
   ```

   Eine andere:

   ```
   f2 = f1 + f1; f3 = f2 + f2;
   ```

5. Haben die im folgenden nebenander gestellten Anweisungen stets die gleiche Wirkung:

```
a = (b - c)*2;      a = b;        Gleiche Wirkung
                    a = a - c;
                    a = a*2;
-------------------------------------------------------
a = (b - a)*2;      a = b;        NICHT die gleiche Wirkung
                    a = a - a;    (entspricht a = 0)
                    a = a*2;
```

```
        ------------------------------------------------------------
        a = (b - c)*2;      a = b*2;      Gleiche Wirkung
                            a = a - c*2;
        ------------------------------------------------------------
        a = (b - c)*2;      b = b*2;      NICHT die gleiche Wirkung
                            a = b - c*2;  (Orginal-Wert von b wird zerstoert)
```

Aufgabe 2

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

using namespace std;

int main () {
  string vorname, nachname, name;

  cout << "Vorname: ";
  cin  >> vorname;

  cout << "Nachname: ";
  cin  >> nachname;

  name = vorname + " " + nachname;

  cout << "Name: " << name << endl;
  cout << "Initialen: " << vorname.at(0) << ". " << nachname.at(0) << "." << endl;
  cout << "Der Vorname hat " << vorname.length () << " Buchstaben" << endl;
  cout << "Der Nachname hat " << nachname.length () << " Buchstaben" << endl;
}
```

Aufgabe 3

1. Syntaktische Fehler findet der Compiler, semantische muss die/der Programmierer(in) finden.
2. - Analysieren: Problem und Lösungsprinzipen verstehen. Wird von Softwareentwicklern gemacht.
   - Übersetzen: Von einer (höhren) in eine andere (Maschinen–) Programmiersprache überführen, d.h. äquivalente Anweisungen finden. Wird vom Compiler gemacht.
   - Entwerfen: Zu einer ausreichend klaren Problemstellung einen konkreten Lösungsweg definieren. Konzept eines Programms angeben. Wird von Softwareentwicklern gemacht.
   - Editieren: Text erstellen oder verändern. Wird von Programmieren genacht.
3. Siehe Skript!
4. $x = 2 \cdot x - 5$
   ist in der Mathematik eine Aussage über einen Wert mit Namen x (Gleichung), oder auch die Definition aller Werte mit der Eigenschaft gleich ihrem doppeltem Wert minus fünf zu sein (Lösungsmenge der Gleichung). In C++ ist es eine Anweisung wie der Wert einer Variablen zu verändern ist.
5. Siehe Skript!
6. Siehe Skript!
7. C++ ist keine Maschinensprache, da C++–Anweisungen nicht direkt vom Prozessor verarbeitet werden können.
8. Siehe Skript!
9. Wer muss wissen wieviele Bytes eine `float`–Zahl im Speicher belegt:
   - Die/der Programmierer(in) muss es nicht (immer oder unbedingt) wissen.
   - Der Compiler muss es wissen.
   - Die Hardware (= Prozessor) die arithmetische Operationen ausführen und die Software (z.B. die Hilfsroutinen zur Ausgabe oder Eingabe) müssen es wissen.

Wer legt es fest:

- Die Hersteller von Prozessoren (z.B. die Firma Intel) bestimmen die durch Hardware realisierten arithmetischen Operationen. Diese sind dann in den Programmiersprachen als arithmetische Datentypen bestimmter Größe verfügbar.
- Der Hersteller von Compilern (z.B. die Firma Microsoft) übernehmen die Vorgaben der Hardware. Sie können aber dazu noch weitere Datentypen per Software nachbilden (“emulieren”).

3 Verzweigungen und Boolesche Ausdrücke

3.1 Bedingte Anweisungen

Die Bedingte Anweisung (If–Anweisung)

Der wesentliche Bestandteil der Programme, die wir bisher kennengelernt haben, ist die main–Funktion. Sie besteht aus Variablendefinitionen und Anweisungen. Die Anweisungen werden in der Reihenfolge, in der wir sie aufschreiben, ausgeführt. Anweisungen wie die Zuweisung und die Ein– und Ausgabeanweisung werden dabei in jedem Fall ausgeführt. Bedingte Anweisungen werden dagegen nur unter Umständen ausgeführt. Eine Bedingung wird getestet und nur, wenn sie erfüllt ist, wird etwas getan.

Mit einer *bedingten Anweisung* (If–Anweisung) können also die Aktionen eines Programms davon abhängig gemacht werden, ob eine Bedingung erfüllt ist. Damit lassen sich Programme konstruieren, die flexibel auf ihre Eingabewerte eingehen. Ein Beispiel für eine bedingte Anweisung ist:

```
if ( x > y ) cout << x << endl;
```

Mit dieser Anweisung wird der Wert von `x` nur dann ausgegeben, wenn er größer ist als der von `y`.

Die If–Anweisung gibt es in zwei Varianten, ein– und zweiarmig.

Die *Einarmige If–Anweisung* hat die Form:

`if (`$<Bedingung>$`)`$<Anweisung>$

Die *Zweiarmige If–Anweisung* sieht folgendermaßen aus:

`if (`$<Bedingung>$`)`$<Anweisung>$`else`$<Anweisung>$

Die einarmige If–Anweisung

Die einarmige If-Anweisung ist ein “Wenn–Dann”. Wenn die Bedingung erfüllt ist, dann wird die Anweisung ausgeführt. Wir betrachten dazu ein kleines Beispiel:

```
#include <iostream>

using namespace std;

int main () {
  int a, b,  // eingelesene Zahlen
      max;   // Maximum von a und b

  cout << "1. Zahl: ";
  cin >> a;
  cout << "2. Zahl: ";
  cin >> b;

  if (a > b)  max = a;
  if (b > a)  max = b;
  if (a == b) max = b;  // Achtung: == fuer Vergleich

  cout << "Das Maximum ist: " << max << endl;
}
```

Das Programm erwartet zwei ganze Zahlen als Eingabe, bestimmt deren Maximum[5] und gibt es dann aus. Mit den drei Bedingungen `a > b`, `b > a` und `a == b` werden alle möglichen Größenbeziehungen zwischen `a` und `b` abgedeckt. Genau eine trifft zu und genau eine Zuweisung an `max` wird ausgeführt.

Achtung: der *Vergleich* von zwei Werten wird mit == ausgedrückt! Ein Gleichheitszeichen allein ist eine Zuweisung! Schreibt man irrtümlich

```
if (a = b) .... // FALSCH  statt
if (a == b) ... // RICHTIG
```

[5] Das Maximum von zwei Zahlen a und b ist die kleinste Zahl c für die gilt $a \leq c$ und $b \leq c$.

dann zeigt der Compiler keinen Fehler an. Das Programm wird übersetzt, verhält sich aber sicher nicht so wie erwartet.

Die zweiarmige If–Anweisung

Die zweiarmige If-Anweisung ist ein "Wenn–Dann–Andernfalls". Wenn die Bedingung erfüllt ist, dann wird die erste Anweisung ausgeführt, andernfalls die zweite. Die Bestimmung des Maximums sieht mit ihr folgendermaßen aus:

```
...
if ( a > b )
  max = a;
else
  max = b;
...
```

Auch bei dieser Anweisung wird in jedem der drei möglichen Fälle genau eine der beiden Zuweisungen ausgeführt.

Formatierung von C++ Programmen

Es kommt bei C++ nicht auf die Formatierung und die Einrückung an. Der Compiler interessiert sich ausschließlich für die aufgeschriebenen Zeichen und nicht für die Art, wie sie in einer Datei verteilt sind. So sind beispielsweise die Anweisungen

```
// OK:
if (a>b) max=a;
else max=b;

//Pfui:
if(a>b)
max = a; else
max = b;

// OK:
if ( a > b )
     max = a;
else
     max = b;
```

```
// OK:
if ( a > b ) max = a; else max = b;

// Pfui:
if(a>b) max = a; else
max = b;

// Pfui:
if(a>b)
max=a;
else
max = b;
```

vollkommen äquivalent. Man sollte sich allerdings an die üblichen Konventionen halten und seinen Programmen ein ansprechendes Aussehen geben. Die Formatierung eines Quellprogramms ist "ansprechend", wenn seine innere logische Struktur an der Textform erkennbar wird. Ein gewisser eigener Stil wird zwar allgemein toleriert, mit einer zu eigenwilligen Form riskiert man jedoch seinen Ruf und eventuell die kollegiale Zusammenarbeit.

Statische und dynamische Auswahl von Anweisungen

Verzweigungen definieren Alternativen in der Abarbeitung, über die *dynamisch* entschieden wird. Der Prozessor, der das (aus dem C++–Programm erzeugte Maschinen–) Programm abarbeitet, entscheidet darüber, welche Anweisungen ausgeführt werden und welche nicht. Damit wird über den Verlauf des Programms nicht am Tisch oder Bildschirm des Programmierers sondern während der Laufzeit des Programms entschieden. Dinge die *während des Programmlaufs* entschieden werden, nennt man in der Welt der Programmiersprachen *dynamisch*. Das Gegenteil von "dynamisch" ist "statisch". *Statisch* ist alles, was vor der Laufzeit – im *Quelltext des Programms* – festgelegt wird.

Anweisungen erzeugen Zustandsänderungen

An einigen Beispielen zeigen wir, wie der *Zustand* des Programms – d.h. die aktuelle Belegung seiner Variablen – durch verschiedene Anweisungen verändert wird:

- – Zustand vorher: $a = 3, b = 4$

  - Anweisung: `if (a==b) a = 5; b = 6;`
  - Zustand nachher: $a = 3, b = 6$

- Zustand vorher: $a = 3, b = 4$
  - Anweisung: `if (a==b) a = 5; else b = 6;`
  - Zustand nachher: $a = 3, b = 6$

- Zustand vorher: $a = 2, b = 2$
  - Anweisung: `if (a==b) a = 5; b = 6;`
  - Zustand nachher: $a = 5, b = 6$

- Zustand vorher: $a = 2, b = 2$
  - Anweisung: `if (a==b) a = 5; else b = 6;`
  - Zustand nachher: $a = 5; b = 2$

- Zustand vorher: $a = 2, b = 2$
  - Anweisung: `if (a==b) a = 5; if (a==b) a = 5; else b = 6;`
  - Zustand nachher: $a = 5; b = 6$

3.2 Flussdiagramme und Zusicherungen

Flussdiagramm

Ein *Flussdiagramm* ist eine grafische Darstellung des Programmverlaufs (des *Kontrollflusses*). Beispielsweise sind

```
if (x > y) m = x; else m = y;
if (x > y) m = x;
```

als Flussdiagramm (Siehe Abbildung 6):

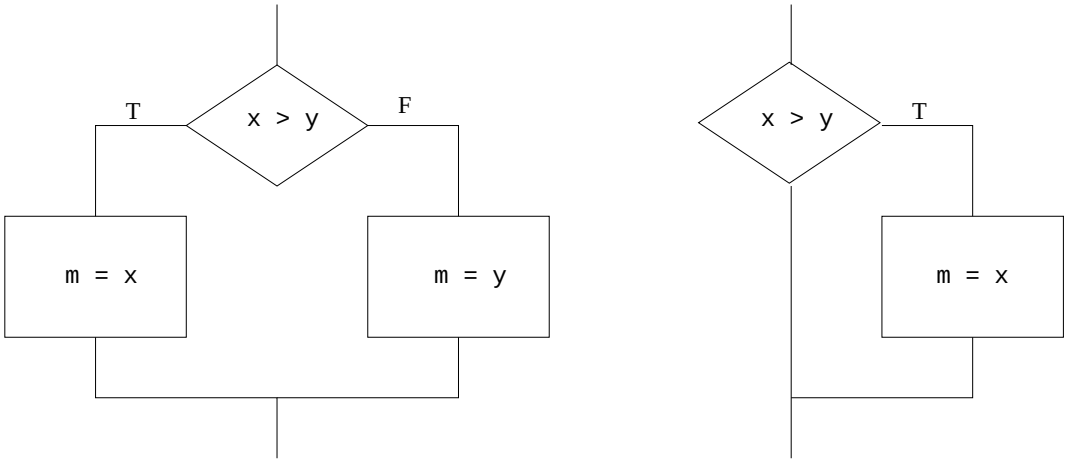


Abbildung 6: Einfache Flussdiagramme

Zusicherung

Eine *Zusicherung* (engl. *Assertion*) ist eine Aussage über den Programmzustand (d.h. die Belegung der Variablen) wenn der Kontrollfluss eine bestimmte Stelle erreicht hat. Beispiel:

```
if (x > y) max = x;
else       max = y;
// max >= x, max >= y
```

Hier soll ausgedrückt werden, dass, wenn immer und mit welchen Werten von `a` und `b` auch immer, der Kontrollfluss den Kommentar

```
// max >= x, max >= y
```

erreicht, dann ist der Wert von `max` größer–gleich dem von `x` und dem von `y`. Derartige Zusicherungen sind gut geeignet die Wirkung eines Programmabschnitts klar zu machen. Sie helfen auch bei der Programmentwicklung, denn der erste, der die Wirkung der Anweisungen des Programms verstehen muss, ist dessen Autor.

In einem Flussdiagramm können Zusicherungen noch besser platziert und feiner eingesetzt werden als im Quellprogramm (Siehe Abbildung 7):

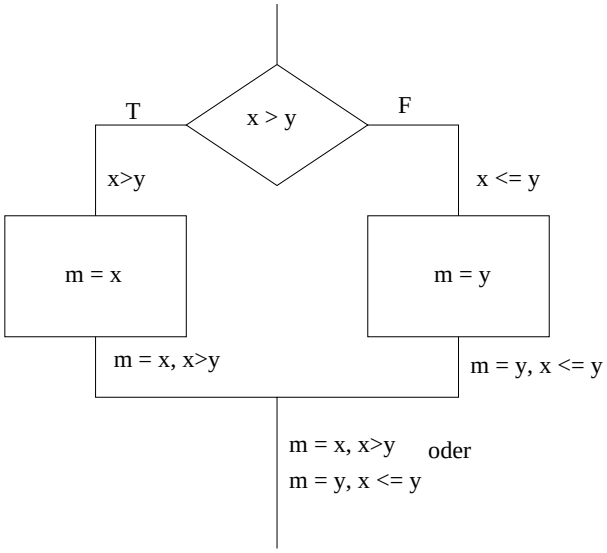


Abbildung 7: Flussdiagramm mit Zusicherungen

assert

Zusicherungen können während der Programmausführung von der Maschine überprüft werden. Dazu gibt es die `assert`–Anweisung:

```
#include <cassert>
assert (< Bedingung >);
```

Wenn die Bedingung nicht erfüllt ist, wird die Programm-Bearbeitung mit einer Fehlermeldung abgebrochen. Eine solche automatisch überprüfte Zusicherung kann bei der Fehlersuche hilfreich sein. Gibt man folgendem fehlerhaften Programm beispielsweise zwei gleiche Zahlen als Eingabe

```
#include <iostream>
#include <cassert>

using namespace std;

int main () {
  int a, b,  // eingelesene Zahlen
      max;   // Maximum von a und b

  cout << "1. Zahl: ";
  cin >> a;
  cout << "2. Zahl: ";
  cin >> b;

  if (a > b)  max = a;
  if (b > a)  max = b;
```

```
  // max ist das Maximum von a und b:
  assert (max>=a);               // max ist mindestens so gross wie a
  assert (max>=b);               // max ist mindestens so gross wie b

  cout << "Das Maximum ist: " << max << endl;
}
```

dann wird es – zumindest mit hoher Wahrscheinlichkeit – statt einen falschen Wert als Maximum auszugeben, mit einer Fehlermeldung abbrechen. Was ist falsch an diesem Programm? Der Fehler ist, dass keine Zuweisung an `max` erfolgt, wenn `a` und `b` gleich sind. Dahinter steht die vergessene Tatsache, dass das Maximum ja auch noch gleich einem der beiden Werte sein muss.

Die korrigierte Version enthält die fehlende Behandlung des Falls gleicher Werte für `a` und `b` und die fehlende Zusicherung:

```
#include <iostream>
#include <cassert>

using namespace std;

int main () {
  int a, b,  // eingelesene Zahlen
      max;   // Maximum von a und b

  cout << "1. Zahl: ";
  cin >> a;
  cout << "2. Zahl: ";
  cin >> b;

  if (a > b)  max = a;
  if (b > a)  max = b;
  if (b == a) max = a;           // <<---------

  // max ist das Maximum von a und b:
  assert (max>=a);
  assert (max>=b);
  assert (max == a || max == b); // <<-- Das Maximum ist entweder = a oder = b !!

  cout << "Das Maximum ist: " << max << endl;
}
```

Im Flussdiagramm mit Zusicherungen für dieses Programm (siehe Abbildung 8) sieht man deutlich, dass `max` nach den ersten beiden If–Anweisungen dann keinen definierten Wert hat, falls `a` und `b` gleich sind. "`max` hat keinen definierten Wert" oder "`max` ist undefiniert" bedeutet, dass `max` irgendeinen *zufälligen Wert* hat. Will man nicht mit zufälligen Werten arbeiten, dann müssen Variablen mit einem Wert belegt *(initialisiert)* werden.[6]

3.3 Geschachtelte und zusammengesetzte Anweisungen

Geschachtelte If–Anweisungen

Eine If–Anweisung besteht aus einer Bedingung und einer oder zwei (Unter–) Anweisungen. In den bisherigen Beispielen waren die Anweisungen innerhalb einer If–Anweisung stets Zuweisungen. Das muss jedoch nicht immer so sein. Eine If–Anweisung kann aus beliebigen anderen Anweisungen aufgebaut sein – auch aus anderen If–Anweisungen. Ein Beispiel (kein Druckfehler !) ist:

```
if (a>b) if (c>d) x = y;
```

Hier sieht man drei ineinander geschachtelte Anweisungen: eine Zuweisung in einer If–Anweisung in einer If–

[6] Es sei denn es sind globale Variable, die automatisch mit 0 initialisiert werden.

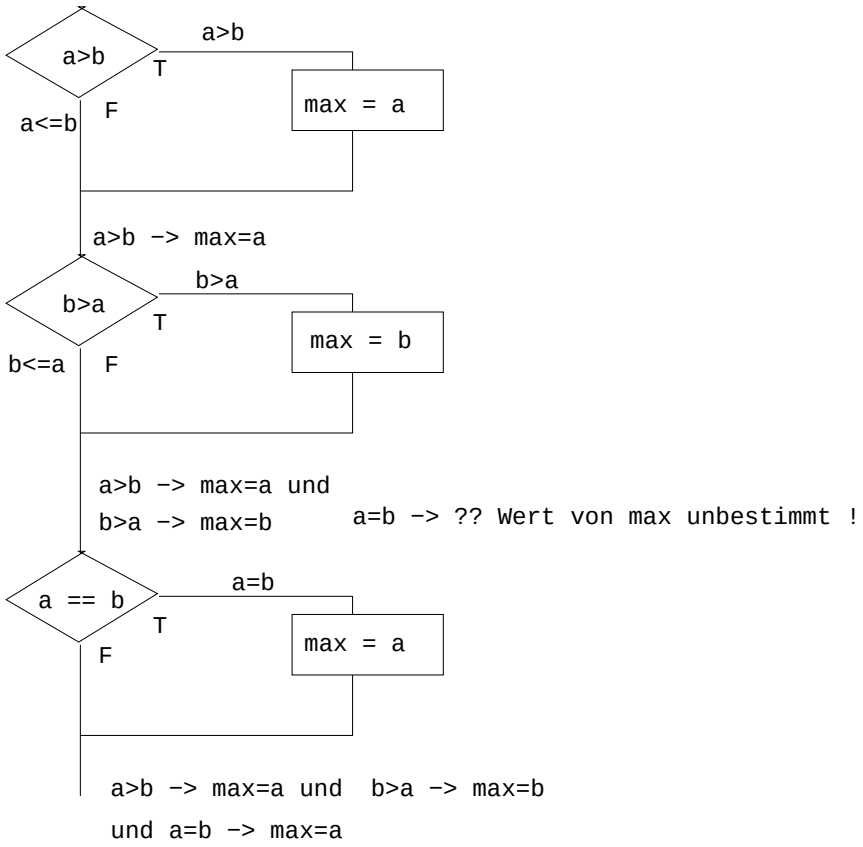
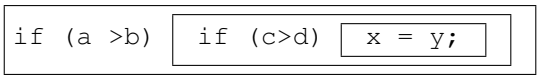


Abbildung 8: Flussdiagramm mit Zusicherungen

Anweisung:



Üblicherweise macht man die Struktur solcher etwas komplexeren Anweisungen durch das Layout des Quelltextes klar:

```
if (a>b)
   if (c>d)
      x = y;
```

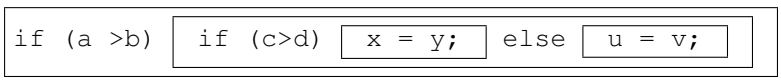
Mit jeder Stufe der Einrückung geht dabei tiefer in die logische Struktur.

Geklammerte Anweisungen

Bei manchen etwas komplexeren Anweisungen ist die logische Struktur nicht sofort aus dem Text ersichtlich. Das gilt insbesondere, wenn in einer Anweisung mehr `if` als `else` vorkommen. Ein Beispiel ist:

```
if (a>b) if (c>d) x = y; else u = v;
```

Die Regeln von C++ legen fest dass ein `else` immer zu nächsten vorherigen `if` gehört. Damit ist die Anweisung oben als



zu interpretieren ist. Das `else` gehört also zum zweiten `if`. (Siehe Abbildung 9)

Es ist wichtig zu beachten, dass die Einrückung dem Verständnis des menschlichen Lesers zwar hilft, dass die Einrückung die Interpretation durch den Compiler aber nicht beeinflusst. Selbst wenn wir

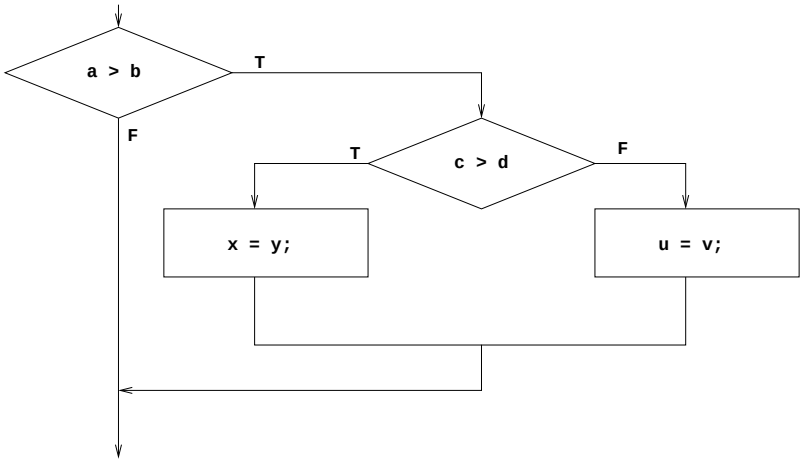
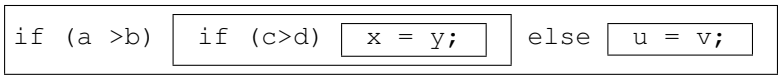


Abbildung 9: Flussdiagramm zu `if (a>b) if (c>d) x = y; else u = v;`

```
if (a>b)       // Achtung: FALSCHES und verwirrendes Layout
   if (c>d)
      x = y;
else
   u = v;
```

schreiben, ändert das nichts daran, dass das `else` zum zweiten `if` gehört. Layout ist für Menschen, es wird vom Compiler ignoriert. Das Layout sollte immer der Interpretation des Textes durch den Compiler entsprechen.

Soll das `else` im letzten Beispiel sich tatsächlich auf das erste `if` beziehen:



dann muss die innere If–Anweisung (in dem Fall ohne `else`–Teil) geklammert werden:

```
if (a>b) { if (c>d) x = y; } else u = v;
```

Ein passendes Layout macht die Struktur klar:

```
if (a>b) {   // OK: Layout passt zur logischen Struktur
   if (c>d)
      x = y;
} else
      u = v;
```

Das Layout passt jetzt zur logischen Struktur des Programmtexts.

Zusammengesetzte Anweisungen

Mit geschweiften Klammern kann man auch mehrere Anweisungen zu (formal) einer zusammenfassen. Das ist beispielsweise dann nützlich, wenn mehrere Anweisungen ausgeführt werden sollen, wenn eine Bedingung erfüllt ist. Mit

```
if (a>b) {
   t = a;
   a = b;
   b = t;
}
```

wird beispielsweise der Inhalt von `a` und `b` getauscht, wenn a größer als `b` ist. Die Klammern müssen sein. Schreibt man etwa – ein häufiger Fehler –

```
if (a>b)       // Achtung: FALSCHES und verwirrendes Layout
   t = a;
   a = b;
   b = t;
```

dann wird dies vom Compiler genau wie

```
if (a>b) t = a;
a = b;
b = t;
```

interpretiert. Formal folgt der Bedingung genau eine Anweisung. Mit geschweiften Klammern kann aber eine beliebige Folge von Anweisungen zu einer *Zusammengesetzten Anweisung* werden:

$$\{ < Anweisung > \ldots < Anweisung > \}$$

Beispiele: Maximum von drei Zahlen

Als Beispiel zum Umgang mit If–Anweisungen zeigen wir hier zwei Arten das Maximum von drei Zahlen zu berechnen. Die Zahlen seien jeweils in den Variablen `a`, `b` und `c` zu finden.

Die einfachste Strategie besteht darin `a`, `b` und `c` der Reihe nach zu inspizieren und in einer Hilfsvariablen `max` das bis jetzt gefundene Maximum aufzuheben.

```
...
int main () {
  int a, b, c, max;
  cout << "1. Zahl: "; cin >> a;
  cout << "2. Zahl: "; cin >> b;
  cout << "3. Zahl: "; cin >> c;

  max = a;               // max = Maximum (a)
  if (b > max)  max = b; // max = Maximum (a, b)
  if (c > max)  max = c; // max = Maximum (a, b, c)

  cout << "Das Maximum ist: " << max << endl;
}
```

Eine ausführliche Analyse aller möglichen Größenverhältnisse zwischen den Variablen ist wesentlich komplexer. Wir zeichnen zunächst ein Flussdiagramm mit Zusicherungen. (Siehe Abbildung 10):

Das entsprechende Programm kann dann leicht konstruiert werden:

```
#include <iostream>

using namespace std;

int main () {
  int a, b, c, max;
  cout << "1. Zahl: "; cin >> a;
  cout << "2. Zahl: "; cin >> b;
  cout << "3. Zahl: "; cin >> c;

  if (a>b) {
    if (a>c)   max = a; else max = c;
  } else {
    if (a>c)   max = b;
    else
      if (b>c) max = b; else max = c;
  }
  cout << "Das Maximum ist: " << max << endl;
}
```

Dieses Beispiel ist natürlich ein wenig konstruiert, aber ein Programmierer sollte ein wenig Training im Umgang mit logischen Formeln haben und wissen, dass komplexe Bedingungen mit Hilfe von Zusicherungen gut analysiert werden können.

Beispiele

Wir zeigen noch einige Beispiele für die Auswertung bedingter und zusammengesetzter Anweisungen (`int x, y;` Bitte nachvollziehen!):

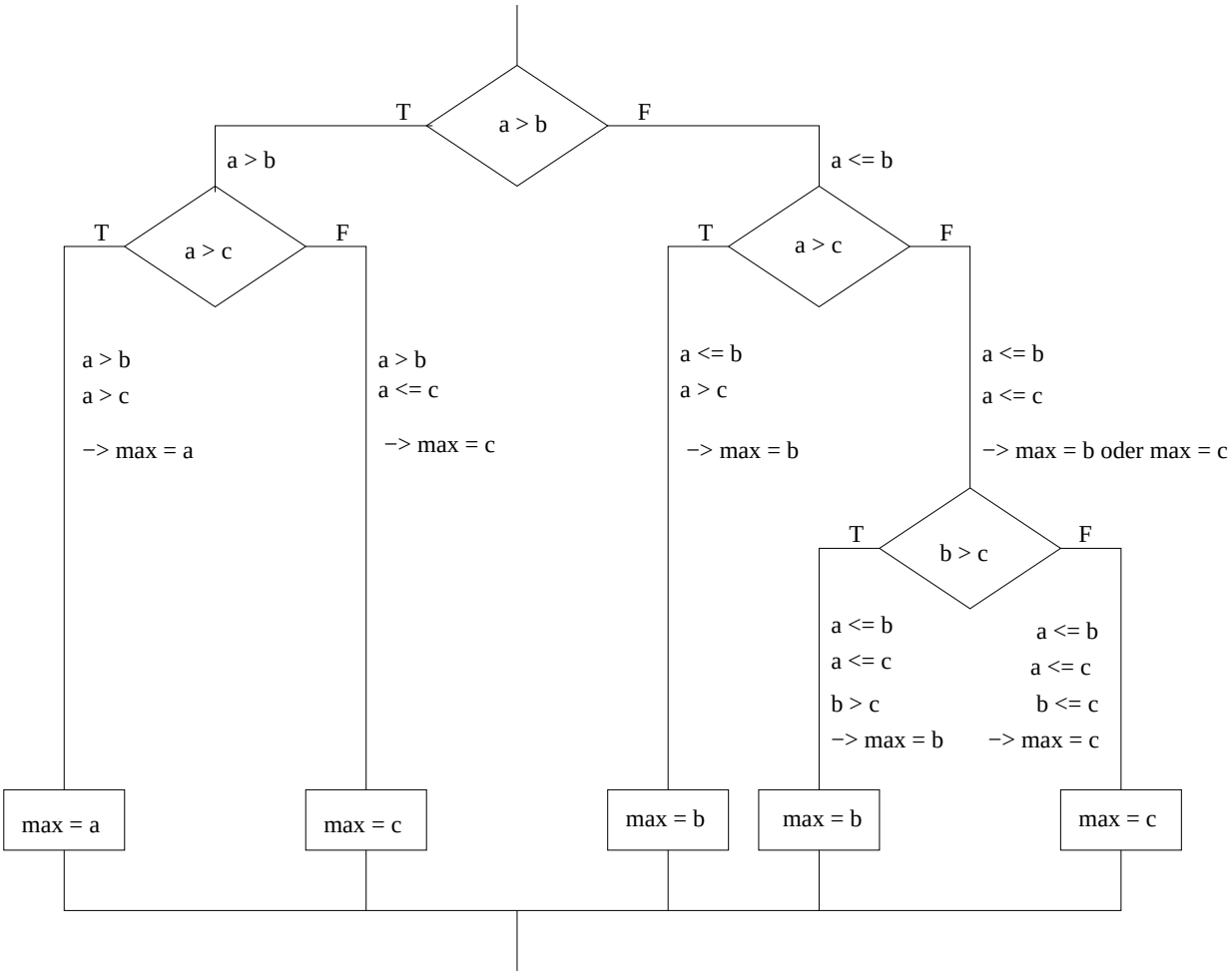


Abbildung 10: Flussdiagramm zur Maximumberechnung ohne Hilfsvariable

- – Zustand vorher: $x = 3, y = 4$
  – Anweisung: `if (x==y) x = 5; y = 6;`
  – Zustand nachher: $x = 3, y = 6$

- – Zustand vorher: $x = 3, y = 4$
  – Anweisung: `if (x==y) {x = 5; y = 6;}`
  – Zustand nachher: $x = 3, y = 4$

- – Zustand vorher: $x = 3, y = 4$
  – Anweisung: `if (x==y) x = y; if (x==y) y = 6;`
  – Zustand nachher: $x = 3, y = 4$

- – Zustand vorher: $x = 3, y = 4$
  – Anweisung: `if (x==y) y = x; {y = x; if (x==y) y = 6;}`
  – Zustand nachher: $x = 3, y = 6$

3.4 Die Switch–Anweisung

Beispiel

Neben den beiden Varianten der IF–Anweisung gibt es eine dritte Form der Verzweigung: die Switch–Anweisung. Beginnen wir mit einem Beispiel. Eine einfache Switch–Anweisung ist:

```
int a, b, max;
...
switch (a>b) {
case true:
   max = a;
   break;
case false:
   max = b;
   break;
}
```

Hier wird wieder einmal der Variablen `max` das Maximum von `a` und `b` zugewiesen. Die Switch–Anweisung beginnt mit dem Schlüsselwort `switch`, dem ein Ausdruck in Klammern folgt. Der Ausdruck – im Beispiel `a>b` – wird ausgewertet und dann wird die erste `case`–*Marke* gesucht, deren Wert dem Wert des Ausdrucks entspricht. Die Programmabarbeitung wird dann dort mit der entsprechenden *Alternative* fortgesetzt. Mit `break` wird die gesamte Switch–Anweisung verlassen.

De facto ist das natürlich nichts anderes, als

```
if (a>b) max = a; else max = b;
```

auf etwas umständliche Weise auszudrücken.

Die Switch–Anweisung ist eine Verallgemeinerung der bedingten Anweisung

Die Switch–Anweisung kann also als bedingte Anweisung verwendet werden. Sie kann aber mehr, sie ist eine verallgemeinerte Form des `if-then-else`. Die Verallgemeinerung besteht darin, dass

1. der Ausdruck kein boolescher Ausdruck sein muss und,
2. statt ein oder zwei, beliebig viele Alternativen erlaubt sind.

Dazu ein Beispiel:

```
char   note;
string ton;
...
switch (note) {
case 'C' : ton = "do"; break;
case 'D' : ton = "re"; break;
case 'E' : ton = "mi"; break;
case 'F' : ton = "fa"; break;
case 'G' : ton = "so"; break;
case 'A' : ton = "la"; break;
case 'H' : ton = "ti"; break;
default  : ton = "??"; break;
}
```

Der Ausdruck, dessen Wert den weiteren Verlauf des Programms bestimmt, ist hier mit `note` eine `char`–Variable, also ein Ausdruck mit einem `char`–Wert. Die Entscheidung wird hier über acht Alternativen gefällt.

Die `default`–Alternative

Man beachte das Schlüsselwort `default` im letzten Beispiel. Es ist so etwas wie das `else` der Switch–Anweisung. Die `default`–Alternative wird immer gewählt, wenn *vorher* keine andere `case`–Marke dem Wert des Ausdrucks entsprochen hat. Die `default`–Alternative muss nicht die letzte sein, die ihr folgenden werden auch beachtet. Üblicherweise schreibt man sie aber als letzte.

`break` in Switch–Anweisungen

Die `break`–Anweisung war in den bisherigen Beispielen stets die letzte Anweisung in einer Alternative. Das wird von den syntaktischen Regeln von C++ *nicht unbedingt* gefordert. Ohne `break` wird die Switch–Anweisung mit

dem Ende einer Alternative nicht beendet, sondern mit der textuell nächsten Alternative fortgesetzt. Sie "fällt" von einer Alternative in die nächste.[7] So erzeugt beispielsweise

```
int c = 1;
switch (c) {
case 1 : cout << "c  = 1" << endl;
case 2 : cout << "c  = 2" << endl;
case 3 : cout << "c  = 3" << endl;
default: cout << "c != 1, 2, 3" << endl;
}
```

die Ausgabe

```
c  = 1
c  = 2
c  = 3
c != 1, 2, 3
```

Man sollte sich darum angewöhnen jede Alternative mit einem `break` zu beenden. Bei der letzten Alternative ist ein `break` natürlich ohne Wirkung. Wir fügen es trotzdem ein, damit es nicht vergessen werden kann, wenn später noch eine weitere Alternative hinzugefügt wird.

Mehrfache Marken

Eine Alternative kann mit mehreren Marken begonnen werden. Das ist dann nützlich, wenn bei unterschiedlichen Werten die gleiche Alternative betreten werden soll. Z.B.:

```
char c;
...
switch (c) {
case  'a': case 'e': case 'i': case 'o': case 'u':
case  'A': case 'E': case 'I': case 'O': case 'U':
      cout << "c ist ein Vokal" << endl;
      break;
default: cout << "c ist kein Vokal" << endl;
      break;
}
```

Allgemeine Form der Switch–Anweisung

Die Switch–Anweisung hat folgende allgemeine Form:

`switch` (< *Ausdruck* >) {< *Alternative* > ... < *Alternative* > }

Wobei jede < *Alternative* > entweder eine `case`–Alternative:

`case` < *Wert* >: ... `case` < *Wert* >: < *Anweisung* > ... < *Anweisung* >

oder eine `default`–Alternative

`default` : < *Anweisung* > ... < *Anweisung* >

ist.

3.5 Arithmetische und Boolesche Ausdrücke und Werte

Boolesche Ausdrücke und Werte

Wie in vielen anderen Programmiersprachen ist in C++ eine Bedingung wie

[7] Diese Obskurität hat C++ von C geerbt.

```
x > y + 3
```

ein *Ausdruck* (engl. *Expression*), der sich von einem "normalen" *arithmetischen Ausdruck* wie beispielsweise

```
x * y + 3
```

nur dadurch unterscheidet, dass sein Wert keine Zahl, sondern ein Wahrheitswert ist.

Bedingungen sind *logische* – oder wie man allgemein sagt – *boolesche Ausdrücke*. Sie werden wie diese ausgewertet und haben den Wert *wahr* (engl. *true*) oder *falsch* (engl. *false*). Wahr und falsch sind die beiden möglichen *Wahrheitswerte*, oder die beiden *booleschen*[8] Werte.

Die Wahrheitswerte: `true` und `false`

Wahr und falsch haben auch in C++ Namen: `true` und `false`. Diese beiden Namen kann man auch in Programmen verwenden. So kann statt

```
if (a == a) ...
```

ebensogut

```
if (true) ...
```

geschrieben werden. `a == a` hat ja stets den Wert `true`. Zugegeben, weder das eine noch das andere ist besonders sinnvoll. Die Sache wird allerdings dadurch interessanter, dass boolesche Ausdrücke nicht nur als Bedingungen verwendet werden können.

Boolesche Variable: Variable mit logischem Wert

Wahrheitswerte sind in C++ "richtige" Werte: Sie haben einen Datentyp (`bool`) und man kann Variablen definieren, deren Wert ein Wahrheitswert (`true` oder `false`) ist. Die Definition boolescher Variablen hat die gleiche Form wie die aller anderen Variablen. Ein Beispiel ist:

```
bool b1, b2 = true;
```

Sie wird auch genauso in Zuweisungen und Ausdrücken verwendet. Im folgenden Beispiel wird festgestellt welche von zwei eingegeben Zahlen die größere ist:

```
...
int main () {
  int  a, b;
  bool ma = false, // soll den Wert true erhalten falls a > b
       mb = false; // soll den Wert true erhalten falls b > a

  cout << "1. Zahl: "; cin >> a;
  cout << "2. Zahl: "; cin >> b;

  if (a > b)  ma = true; else ma = false;
  if (b > a)  mb = true; else mb = false;

  if (ma == true) cout << "a > b" << endl;
  if (mb == true) cout << "b > a" << endl;
}
```

Was gibt das Programm aus, wenn zwei verschiedene und wenn zwei gleiche Zahlen eingegeben werden?[9]

Boolesche Ausdrücke

Eine Bedingung ist ein boolescher Ausdruck, also ein Ausdruck mit booleschem Wert. Boolesche Variablen sind Variablen denen boolesche Werte zugewiesen werden können. Damit ist klar, dass ein boolescher Ausdruck nicht nur in einer Bedingung verwendet werden darf, sondern auch rechts und links vom Zuweisungszeichen ("=") stehen kann.

```
if (a > b) ma = true; else ma = false;
```

kann also zu

[8] Nach einem engl. Mathematiker namens Boole benannt.
[9] Das Programm gibt bei der Eingabe gleicher Zahlen nichts aus.

```
ma = a > b;
```

vereinfacht werden. Weiterhin ist jede boolesche Variable schon für sich ein boolescher Ausdruck. Statt

```
if (ma == true) ...
```

schreibt man darum einfach

```
if (ma) ...
```

Damit kann das letzte Beispiel vereinfacht werden zu:

```
...
int main () {
  int  a, b;
  bool ma = false,
       mb = false;

  cout << "1. Zahl: "; cin >> a;
  cout << "2. Zahl: "; cin >> b;

  ma = a > b; // == if (a > b) ma = true; else ma = false;
  mb = b > a; // == if (b > a) mb = true; else mb = false;

  if (ma)     // == if (ma == true)
     cout << "a > b" << endl;
  if (mb)     // == if (mb == true)
     cout << "b > a" << endl;
}
```

Beispiel

Ein weiteres Beispiel zur Berechnung des Maximums und der Verwendung boolescher Variablen und Ausdrücke ist:

```
...
int main () {
  int  a, b;        // eingelesene Zahlen
  bool a_max      = false,  // wahr wenn a > b
       b_max      = false,  // wahr wenn b > a
       a_gleich_b = false;  // wahr wenn a gleich b

  cout << "1. Zahl: "; cin >> a;
  cout << "2. Zahl: "; cin >> b;

  a_max      = a > b;
  b_max      = b > a;
  a_gleich_b = a_max == b_max;

  if (a_max) cout << "Das Maximum ist: " << a << endl;
  if (b_max) cout << "Das Maximum ist: " << b << endl;
  if (a_gleich_b) cout << "Beide Zahlen sind gleich" << endl;
}
```

Hier wird in der Zuweisung

```
a_gleich_b = a_max == b_max;
```

der Variablen `a_gleich_b` der Wert `true` zugewiesen, wenn `a_max` und `b_max` den gleichen Wert – `true` oder `false` – haben. Da sie aber niemals beide gleichzeitig den Wert `true` haben können, kommt nur `false` in Frage. Das kann aber nur der Fall sein, wenn weder `a > b`, noch `b > a` zutrifft, `a` und `b` also den gleichen Wert haben.

Arithmetische und logische Operatoren

Ein *Operator* wie z.B. + oder $\star$ verknüpft Teilausdrücke zu neuen Gesamtausdrücken. Operatoren können logische oder arithmetische Argumente und auch logische oder arithmetische Werte haben. Operatoren mit arithmetischen Argumenten und arithmetischem Ergebnis sind +, −, $\star$ /.

Operatoren mit arithmetischen oder logischen Argumenten und logischem Wert sind:

`==`	gleich
`!=`	ungleich
`<`	kleiner
`<=`	kleiner–gleich
`>`	größer
`>=`	größer–gleich

Beim Vergleich von logischen Werten gilt: `false < true`.

Operatoren mit logischen Argumenten und logischem Wert sind:

`||`	oder
`&&`	und
`!`	nicht

Einige Beispiele für die Auswertung von Ausdrücken mit arithmetischen und logischen Operatoren sind (`int x, y; bool a, b;`. Bitte nachvollziehen!):

- – Zustand vorher: $a = true, b = true, x = 3, y = 4$
  – Anweisung:
  ```
  a = (x == y);
  if (a) x = 5; y = 6;
  ```
  – Zustand nachher: $a = false, b = true, x = 3, y = 6$

- – Zustand vorher: $a = true, b = true, x = 3, y = 4$
  – Anweisung:
  ```
  a = ((x != y) == false);
  if (a == false) x = 5; else y = 6;
  ```
  – Zustand nachher: $a = false, b = true, x = 5, y = 4$

- – Zustand vorher: $a = true, b = true, x = 3, y = 4$
  – Anweisung:
  ```
  if (a==b) x = 5; y = 6;
  ```
  – Zustand nachher: $a = true, b = true, x = 5, y = 6$

Vorrangregeln

Die *Vorrangregeln* (Prioritäten) der Operatoren definieren, wie ein nicht vollständig geklammerter Ausdruck zu interpretieren ist. Durch den höheren Vorrang von $*$ gegenüber $+$ wird beispielsweise

$a * b + c * d$

als

$(a * b) + (c * d)$

und nicht etwa als

$a * (b + c) * d$

interpretiert.

Die Vorrangregeln sind (Vorrang zeilenweise abnehmend, innerhalb einer Zeile gleich):

`!, +, -`	unäre Operatoren
`*, /, %`	Multiplikation, Division, Modulo
`+, -`	Addition, Subtraktion
`<, <=, >, >=`	Vergleich
`==, !=`	Gleich, Ungleich
`&&`	Und
`||`	Oder
`=`	Zuweisung

Im Zweifelsfall und bei komplizierten Konstrukten sollten *Klammern verwendet* werden! Auch wenn jeder Ausdruck in C++ eine eindeutige Interpretation hat, sollte man doch nicht zu sparsam mit Klammern umgehen.

3.6 Übungen

Aufgabe 1

1. Zeichnen Sie Flussdiagramme zu folgenden Anweisungen:

   a) `if (a>b) if (c>d) x = y; else u = v;`

   b) `if (a>b) if (c>d) x = y; u = v;`

2. Welchen Wert hat `max` jeweils nach den bedingten Anweisungen

```
(i)                 (ii)                (iii)
a = 3; b = 4;       a = 4; b = 3;       a = 3; b = 3;
if ( a > b )        if ( a > b )        if ( a > b )
  max = a;            max = a;            max = a;
else                else                else
  max = b;            max = b;            max = b;
```

   Welche Zuweisung wird jeweils ausgeführt?

3. Was bedeuten die Begriffe “statisch” und “dynamisch” in der Welt der Programmiersprachen?

4. Mit den bedingten Anweisungen wird der Programmverlauf “dynamisch”, d.h. es ist nicht mehr (allein) durch den Quelltext festgelegt in welcher Reihenfolge welche Anweisungen ausgeführt werden. Wovon hängt der Programmverlauf (die Reihenfolge der ausgeführten Aunweisungen) denn jetzt (noch) ab?

5. Was versteht man unter dem “Zustand” eines Programms?

6. Mit den drei nebeneinander gestellten Anweisungsfolgen soll das Maximum von `a` und `b` bestimmt werden:

```
(i)                     (ii)                    (iii)
if (a > b) max = a;     if (a > b) max = a;     if (a > b) max = a;
if (b > a) max = b;     else max = b;           max = b;
```

   Welche funktioniert/funktionieren wie gewünscht?

7. Schreiben Sie ein Programm, das vier ganze Zahlen einliest und deren Minimum bestimmt. Das Programm soll folgende Struktur haben:

```
int main () {
  int a, b, c, d,
      m1, m2, m3;
  // a .. d einlesen:
  ?...?
  // Minimum von a und b bestimmen:
  ?...?
  // m1 = Minimum (a, b)
  // Minimum von c und d bestimmen:
  ?...?
  // m2 = Minimum (c, d)
  // Minimum von m1 und m2 bestimmen:
  ?...?
  // m3 = Minimum von (m1, m2)
  // m3 ausgeben:
  ?...?
}
```

8. Schreiben Sie ein Programm mit der gleichen Funktionalität mit genau *zwei* Hilfsvariablen!

9. Schreiben Sie ein Programm mit der gleichen Funktionalität mit genau *einer* Hilfsvariablen!

10. Schreiben Sie ein Programm mit der gleichen Funktionalität *ohne* Hilfsvariablen! Das Programm soll also nur die Variablen `a`, `b`, `c`, und `d` benutzen. Diese sollen aber auch nicht als Hilfsvariablen missbraucht werden. Einmal mit dem eingelesenen Wert belegt, dürfen `a`, `b`, `c`, und `d` nicht mehr verändert werden.

    Gehen Sie systematisch vor. Zeichnen Sie zunächst ein Flussdiagramm mit Zusicherungen.

Aufgabe 2

1. Was ist falsch an: `if (a>b) if (c>d) x = y; s = t; else u = v;`
   und was ist falsch an: `if (a>b) if (c>d) { x = y; else u = v; }` ?
2. Wie kann die zweiarmige Verzweigung durch eine (oder mehrere) einarmige ersetzt werden?
3. Vereinfachen Sie so weit wie möglich:

   a) `b = (x == y); if ( b == true ) x = x + y; else x = y + x;`

   b) `if (x >= y) x = x; else x = 2*x;`

   c) `if ((x >= x) == true) x = x; else x = 2*x;`

   d) `if (((x >= x) == true) == false) x = x; x = 2*x;`
4. Welchen Wert haben `a` und `b` nach folgendem Programmstück:

```
int a = 3, b = 1, t = 0;
if (a < b)
   t = a;
   b = a;
   a = t;
cout << a << b;
```

Aufgabe 3

`a` habe den Wert 3 und `b` den Wert 4. Welche Werte haben die Variablen jeweils nach den folgenden Anweisungen:

1. `if (a==b) a = 5; b = 6;`
2. `if (a==b) a = 5; else b = 6;`
3. `if (a==b) { a = 5; b = 6;}`
4. `if (a==b) a = 5; else if (a>b) a = 5; b = 6;`
5. `if (a==b) a = 5; else if (a>b) a = 5; else b = 6;`
6. `if (a==b) a = 5; else { if (a>b) a = 5; else b = 6; }`
7. `if (a==b) a = 5; else if (a>b) { a = 5; b = 6; }`
8. `if (a==b) a = 5; if (a>b) a = 5; b = 6;`
9. `if (a==b) a = 5; if (a>b) { a = 5; b = 6; }`
10. `if (a>=b) a = 5; if (b >= a) a = 6;`
11. `if (a>=b) a = 5; else if (b >= a) a = 6;`
12. `if (a>=b) a = 5; else a = 6;`

Wie ist es, wenn `a` den Wert 4 und `b` den Wert 3 hat und wenn `a` und `b` den gleichen Wert haben?

Aufgabe 4

Wichtiger als die absoluten Werte der Programmvariablen sind die Verhältnisse, welche die Werte verschiedener Variablen zueinander haben. Das Verhältnis der Variablen zueinander kann durch Anweisungen verändert werden oder – bei veränderten Variablenwerten – gleich bleiben.

Die Variable `a` habe den Wert a; die Variable `b` den Wert b, etc. Durch welche Anweisung(en) lässt (lassen) sich die folgenden Wirkungen erzielen (Nach den Anweisungen können `a`, `b`, ... eventuell andere Werte als a, b, ... haben):

1. Vorher : `a` == 4, `b` == 2
   Nachher : `a` == 6

2. Vorher : a == 2*b
   Nachher : a == 2*(b+1)

3. Vorher : a == b
   Nachher : a + b == 0

4. Vorher : x == Min (a, b)
   Nachher : x == Min (a, b, c)

5. Vorher : x == Min (a, b), y == Max (a, b)
   Nachher : x == Min (a, b, c), y == Max (a, b, c)

6. Vorher : x == 2*a, a == a
   Nachher : x == 2*a, a == $a + 1$

7. Vorher : x == (1 + 2 + ... + a), a == a
   Nachher : x == (1 + 2 + ... + a), a == $a + 1$

Hier ist mit a die Variable a mit ihrem aktuellen Wert gemeint. Das kursiv gedruckte a bezeichnet jeweils einen bestimmten festen Wert.

Aufgabe 5

Die logische Implikation $A \Rightarrow B$ bedeutet "aus A folgt B". $A \Rightarrow B$ ist genau dann wahr, wenn B immer dann wahr ist, wenn A wahr ist. ("Wenn es regnet wird die Strasse nass." Das ist wahr, weil immer wenn es regnet auch die Strasse nass ist, aber natürlich ist gelegentlich die Stasse nass, ohne dass es regnet. – Beispielsweise wenn Fifi sein Bein gehoben hat.)

Die Wahrheitstafel der Implikation ist:

A	B	$A \Rightarrow B$
w	w	w
w	f	f
f	w	w
f	f	w

$A \Rightarrow B$ lässt sich auf die elementaren logischen Operationen "und", "oder" und "nicht" zurückführen:

"Aus A folgt B" genau dann wenn "Nicht (A und Nicht B)" (Es kann nicht sein dass A gilt aber nicht B)

Schreiben Sie ein Progamm, das zwei Werte für A und B einliest ("wahr" bzw. "falsch") den Wert für "Aus A folgt B" berechnet und (als "wahr" bzw. "falsch") ausgibt.

Aufgabe 6

Schreiben Sie äquivalente Programmstücke ohne die Verwendung der switch-Anweisung (c sei vom Typ char):

1.
```
switch (c) {
case 'A' : cout << "A" << endl;
           break;
case 'B' : cout << "B" << endl;
           break;
default:   cout << "weder A noch B" << endl;
}
```

2.
```
switch (c) {
case 'A' : cout << "A" << endl;
case 'B' : cout << "B" << endl;
default:   cout << "weder A noch B" << endl;
}
```

3.
```
switch (c) {
case 'A' : cout << "A" << endl;
           break;
```

```
case 'B' : cout << "B" << endl;
           break;
}
```

Aufgabe 7

1. Schreiben Sie ein äquivalentes Programmstück *mit* switch-Anweisung:

```
if (note == 'C') {
   cout << "do";
} else
   if (note == 'D') {
     cout << "re";
   } else
     if (note == 'E') {
       cout << "mi";
     } else
       if (note == 'F') {
         cout << "fa";
       } else
         if (note == 'G') {
           cout << "so";
         } else
           if (note == 'A') {
             cout << "la";
           } else
             if (note == 'H') {
               cout << "ti";
             } else {
               cout << "Ungueltiger Wert von note";
             }
```

2. Was ist falsch an folgendem Programmstück zur Entscheidung ob ein Zeichen ein kleiner oder großer Vokal ist:

```
char c;
cin >> c;
switch (c) {
  default : cout << "Kein Vokal" << endl;
  case 'a', 'e', 'u', 'i' :
    cout "kleiner Vokal'' << endl;
  case 'A', 'E', 'U', 'I' :
    cout "grosser Vokal'' << endl;
  case
}
```

Aufgabe 8

Schreiben Sie ein Programm das vier ganze Zahlen einliest und sowohl deren Minimum, als auch deren Maximum ausgibt.

3.7 Lösungshinweise

Aufgabe 1

1. Flussdiagramme

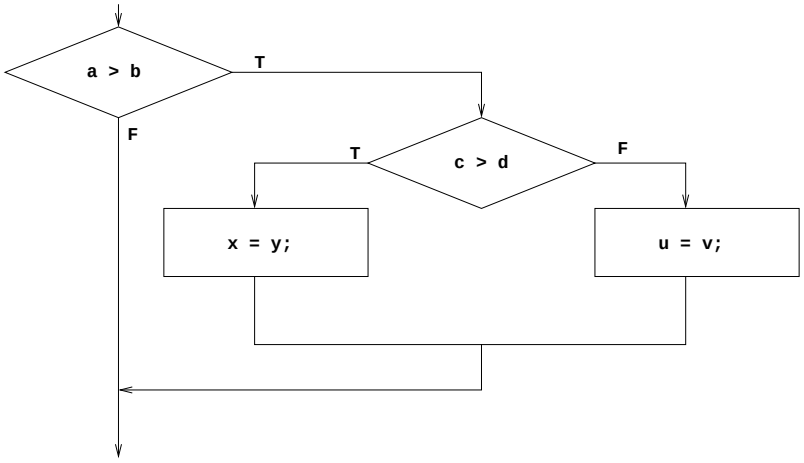


Abbildung 11: Flussdiagramm zu `if (a>b) if (c>d) x = y; else u = v;`

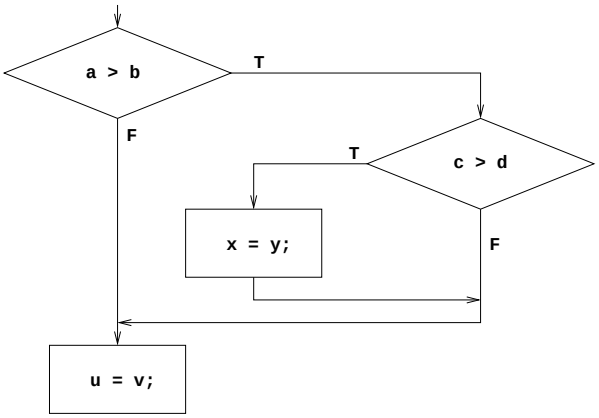


Abbildung 12: Flussdiagramm zu `if (a>b) if (c>d) x = y; u = v;`

2.
```
(i)                 (ii)                (iii)
a = 3; b = 4;       a = 4; b = 3;       a = 3; b = 3;
if ( a > b )        if ( a > b )        if ( a > b )
  max = a;            max = a;//<--       max = a;
else                else                else
  max = b;//<--       max = b;            max = b; //<--
// max == 4         // max == 4         // max == 3
```

3. Siehe Skript!

4. Mit den bedingten Anweisungen hängt der Programmverlauf (die Reihenfolge der ausgeführten Anweisungen) noch von der Eingabe ab.

5. Der Zustand eines Programms ist die aktuelle Belegung all seiner Variablen.

6. Mit den drei nebeneinander gestellten Anweisungsfolgen soll das Maximum von `a` und `b` bestimmt werden:

```
(i)                     (ii)                    (iii)
if (a > b) max = a;     if (a > b) max = a;     if (a > b) max = a;
if (b > a) max = b;     else max = b;           max = b;

NICHT OK,               OK                      NICHT OK
Bei a == b                                      max = b wird IMMER
```

```
erhaelt max nicht                          ausgefuehrt!
den richtigen Wert
```

7. Das Einlesen und Ausgeben von Werten sollte keine Probleme bereiten. Mit

   ```
   // Minimum von a und b bestimmen
   ?...?
   // m1 = Minimum (a, b)
   ```

   ist gemeint, dass an Stelle von `?...?` Programm-Code eingesetzt werden soll, an dessen Ende in der Variablen `m1` das Minimum der Werte der Variablen `a` und `b` zu finden ist. Beispielsweise kann das mit den Anweisungen:

   ```
   // Minimum von a und b bestimmen:
   if (a < b)
      m1 = a;
   else
      m1 = b;
   // m1 = Minimum (a, b)
   ```

   erreicht werden. Das gesamte Programm ist dann:

   ```
   #include <iostream>
   using namespace std;
   int main () {
     int a, b, c, d,
         m1, m2, m3;
     // a .. d einlesen:
     cin >> a >> b >> c >> d;

     // Minimum von a und b bestimmen:
     if (a < b) m1 = a;
     else       m1 = b;
     // m1 = Minimum (a, b)

     // Minimum von c und d bestimmen:
     if (c < d) m2 = c;
     else       m2 = d;
     // m2 = Minimum (c, d)

     // Minimum von m1 und m2 bestimmen:
     if (m1 < m2) m3 = m1;
     else         m3 = m2;
     // m3 = Minimum von (m1, m2)

     // m3 ausgeben:
     cout << m3 << endl;
   }
   ```

8. Durch Wiederverwendung von `m2` kann man `m3` einsparen:

   ```
   ... wie oben ...

   // Minimum von m1 und m2 bestimmen:
   if (m1 < m2) m2 = m1;

   cout << m2 << endl;
   }
   ```

9. Mit einer Hilfsvariablen muss die Strategie etwas geändert werden:

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

```
int main () {
  int a, b, c, d,
      m;
  // a .. d einlesen:
  cin >> a >> b >> c >> d;

  m = a;
  // m ist das Minimum (a)

  if (b < m) m = b;
  // m ist das Minimum (a, b)

  if (c < m) m = c;
  // m ist das Minimum (a, b, c)

  if (d < m) m = d;
  // m ist das Minimum (a, b, c, d)

  cout << m << endl;
}
```

10. Diese Aufgabe ist anspruchsvoll und kann im ersten Anlauf ausgelassen werden.

    Wenn keine Hilfsvariable benutzt werden darf, dann muss die Belegung der Variablen `a, ... d` in verschachtelten If–Anweisungen untersucht werden. Zeichnen Sie ein Flussdiagramm und versehen Sie es mit Zusicherungen. Anders geht es nicht oder nicht ohne besondere Mühe.

Aufgabe 2

1. `if (a>b) if (c>d) x = y; s = t; else u = v;`
   Diese Anweisung ist syntaktisch falsch. Zu `else` gibt es kein passendes `if`. Korrekt wäre bespielsweise:
   `if (a>b) if (c>d) { x = y; s = t; } else u = v;`
   (`else` zum zweiten `if`) oder auch
   `if (a>b) { if (c>d) x = y; s = t; } else u = v;`
   (`else` zum ersten `if`)

2. Die zweiarmige Verzweigung `if (B) A1; else A2;` kann durch `if (B) A1; if (!B) A2;` ersetzt werden.

3. a) `b = (x == y); if ( b == true ) x = x + y; else x = y + x;`
      ≡ `b = (x == y); if (b) x = x + y; else x = y + x;`
      ≡ `if (x == y) x = x + y; else x = y + x;`
      ≡ `x = y + x;` Beide Zweige der `if`–Anweisung sind identisch

   b) `if (x >= y) x = x; else x = 2*x;`
      ≡ `if (!(x >= y)) x = 2*x; else x = x;`
      ≡ `if (!(x >= y)) x = 2*x;`
      ≡ `if (x < y) x = 2*x;`

   c) `if ((x >= x) == true) x = x; else x = 2*x;`
      ≡ `if (x < x) x = 2*x;`
      Da $x < x$ immer falsch ist, kann weiter vereinfacht werden: die Anweisung tut nichts, kann also komplett gestrichen werden.

   d) `if (((x >= x) == true) == false) x = x; x = 2*x;`
      ≡ `if ((x >= x) == false) x = x; x = 2*x;`
      ≡ `if (!(x >= x)) x = x; x = 2*x;`
      ≡ `if (x < x) x = x; x = 2*x;`
      ≡ `if (false) x = x; x = 2*x;`
      ≡ `x = 2*x;`

4. Experimentieren Sie! Das Problem hier ist die falsche und verwirrende Formatierung, die die Illusion erweckt alle Zuweisungen – und nicht nur die erste – würden zu der If–Anweisung gehören.

Aufgabe 3

Bitte überlegen Sie und prüfen Sie Ihre Ergebnisse dann (zumindest stichprobenartig) mit einem kleinen Testprogramm.

Aufgabe 4

1. Vorher : a == 4, b == 2
   Nachher : a == 6

   Die Anweisung `a = 6;` erledigt das Gewünschte. `b` wird im "Nachher" nicht erwähnt, es ist irrelevant und jeder Wert von `b` ist erlaubt. Damit wäre auch `b = 6; a = b;` eine korrekte Lösung.

2. Vorher : a == 2*b
   Nachher : a == 2*(b+1)

   `b = b-1;` wäre hier eine korrekte Lösung. Gilt beispielsweise vorher `a==6, b==3`, dann ist nachher `a==6, b==2` und damit auch `a==2*(b+1), b==2`

   `a=a+2;` ist ebenfalls korrekt, denn $2 * (b + 1) = 2 * b + 2$ und $2 * b$ ist der alte Wert von `a`.
   Gilt beispielsweise vorher `a==6, b==3`, dann ist nachher `a==8==6+2, b==3` und damit auch `a==2*(b+1)==2*b+2==6+2, b==3`

   Selbstverständlich ist auch `a = 2*(b+1);` eine korrekte Lösung.

3. Vorher : a == b
   Nachher : a + b == 0

   `a = -b;` oder `b = -a;`

4. Vorher : x == Min (a, b)
   Nachher : x == Min (a, b, c)

   ```
   if (c<x) x = c;
   ```

5. Vorher : x == Min (a, b), y == Max (a, b)
   Nachher : x == Min (a, b, c), y == Max (a, b, c)

   ```
   if (c<x) x = c; if (c>y) y = c;
   ```

6. Vorher : x == 2*a, a == a
   Nachher : x == 2*a, a == $a + 1$

   Hier ist gefordert, dass die Werte von `x` und `a` in einer bestimmten Beziehung stehen und ausserdem dass der Wert von `a` sich um eins erhöhen soll.

   `a = a+1;` erfüllt die zweite Forderung, bringt aber die Beziehung `x == 2*a` "aus dem Lot". Sie muss durch Veränderung von `x` wieder hergestellt werden – `a` darf ja nicht verändert werden.

   `x = x+2;` stellt die durch `a = a+1;` zerstörte Beziehung `x == 2*a` wiederher.

   Folgende Zuweisungsfolgen sind Beispiele für korrekte Lösungen:

   ```
   a = a+1; x = x+2;
   a = a+1; x = 2*a;
   ```

   Die Reihenfolge der beiden Anweisungen jeweils wesentlich! Die erste ist vorzuziehen, da die Addition eine wesentlich einfache und damit schnellere Operation ist als die Multiplikation.

7. Vorher : x == (1 + 2 + ... + a), a == a
   Nachher : x == (1 + 2 + ... + a), a == $a + 1$

   Vorher enthält `x` die Summe bis zum aktuellen Wert von `a` (zu diesem Zeitpunkt ist das a). Nachher soll `x` ebenfalls die Summe bis zum aktuellen Wert von `a` enthalten. Dier Wert von `a` soll aber jetzt um eins grösser – also $a + 1$ – sein.

   `a = a+1; x = x+a;` oder auch
   `x = x+a+1; a = a+1;` sind korrekt. Man mache sich dies auch an Hand eines Beispiels klar!

Aufgabe 5

Informatiker(–innen) sollten mit der elementaren Aussagenlogik vertraut sein. Sollten Sie in Ihrem Unterricht an der Schule nicht gelernt haben, mit Dingen wie $A \Rightarrow B$ umzugehen, dann fragen Sie bitte Ihren Mathematikprofessor!

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

using namespace std;

int main () {
  bool a, b;
  string a_i, b_i;  // fuer die Eingabe

  cout << "A : ";
  cin >> a_i;
  if (a_i == "wahr")
    a = true;
  else if (a_i == "falsch")
    a = false;
  else
    cout << "Falsche Eingabe !" << endl;

  cout << "B : ";
  cin >> b_i;
  if (b_i == "wahr")
    b = true;
  else if (b_i == "falsch")
    b = false;
  else
    cout << "Falsche Eingabe !" << endl;

  cout << " A => B  == ";
  if ( !(a && !b) ) cout << "wahr";
  else              cout << "falsch";
  cout << endl;
}
```

Aufgabe 6

Zum Knobeln.

Aufgabe 7

1. Das äquivalente Programmstück ist:

   ```
   switch (note) {
   case 'C': cout << "do"; break;
   case 'D': cout << "re"; break;
   case 'E': cout << "mi"; break;
   case 'F': cout << "fa"; break;
   case 'G': cout << "so"; break;
   case 'A': cout << "la"; break;
   case 'H': cout << "ti"; break;
   default : cout << "Ungueltiger Wert von note"; break;
   }
   ```

2. ```
   char c;
   cin >> c;
   switch (c) {
     default :
        cout << "Kein Vokal" << endl;        break;
     case 'a': case 'e': case 'u': case 'i': case 'o':
   ```
   ```

```
        cout << "kleiner Vokal" << endl;     break;
      case 'A': case 'E': case 'U': case 'I': case 'O':
        cout << "grosser Vokal" << endl;
        break;  //nicht unbedingt notwendig aber guter Stil
    }
```

Aufgabe 8

Kein Lösungshinweis! Arbeiten Sie am Rechner bis Sie eine funktionierende Lösung haben!

4 Schleifen

4.1 Die While–Schleife

Schleifen: Anweisungen wiederholt ausführen

Schleifen sind ein Mechanismus zur Wiederholung. Mit ihnen können Anweisungen mehrfach ausgeführt werden. Es gibt mehrere Arten von Schleifen. Eine *While–Schleife* führt Anweisungen so lange aus, wie eine Bedingung erfüllt ist. Das folgende Beispiel enthält eine While–Schleife die fünfmal "Hallo" ausgibt. Die Variable `i` startet mit dem Wert 0 und solange (engl. "*while*") `i<n` ist, wird `Hallo` ausgegeben und `i` erhöht:

```
#include <iostream>

using namespace std;

int main () {
  const int n = 5;  // wie oft insgesamt
  int       i;      // wie oft bisher

  i = 0;
  while (i<n) {
    cout << "Hallo" << endl;
    i = i+1;
  }
}
```

Die beiden Anweisungen

```
cout << "Hallo" << endl;
i = i+1;
```

stehen zwar nur einmal im Programmtext, sie werden aber mehrfach ausgeführt. Genau gesagt werden sie *solange* wiederholt wie die Bedingung

```
i<n
```

erfüllt ist.

While–Schleifen: von einer Bedingung gesteuerte Wiederholungen

Eine While–Schleife hat die Form

`while (` $<$ *Bedingung* $>$ `)` $<$ *Anweisung* $>$

Die Bedingung steuert dabei die Wiederholung der Anweisung, die – wie im Beispiel oben – selbstverständlich eine zusammengesetzte Anweisung sein kann. Ein Flussdiagramm macht Struktur und Wirkung der While–Anweisung schnell klar (Siehe Abbildung 13):

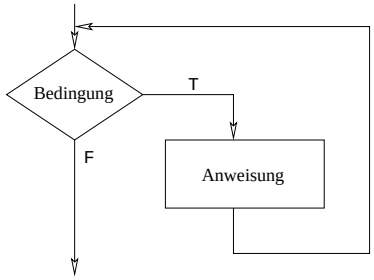


Abbildung 13: While–Schleife

Bei der Ausführung wird zuerst die Bedingung getestet. Falls sie *nicht* zutrifft wird die Schleife *verlassen*. Falls doch wird die Anweisung ausgeführt und es geht danach zurück zur Bedingung. Ist die Bedingung am Anfang nicht erfüllt, dann wird die Schleife übersprungen.

Im nächsten Beispiel werden fünf Zahlen eingelesen und dann ihre Summe ausgegeben:

```
#include <iostream>

using namespace std;

int main () {
  const int n = 5;  // Zahl der Schleifendurchlaeufe insgesamt
  int a,            // eingelesene Zahlen
      s,            // aufsummierte Zahlen
      i;            // Schleifendurchlaeufe bisher

  i = 0;
  s = 0;
  while (i < n) {
    cout << i+1 << "-te Zahl: ";
    cin >> a;
    s = s + a;
    i = i+1;
  }
  cout << "Die Summe ist: " << s << endl;
}
```

Dynamisch bestimmte Zahl der Schleifendurchläufe

Die Zahl der Schleifendurchläufe muss nicht unbedingt beim Programmstart festliegen. Sie kann dynamisch, d.h. während der Laufzeit des Programms, bestimmt werden. Ein Beispiel ist folgendes Programm, das beliebig viele Zahlen aufsummiert. Solange keine Null eingegeben wird, liest das Programm ein und summiert auf:

```
#include <iostream>

using namespace std;

int main () {
  int a,            // eingelesene Zahlen
      s;            // aufsummierte Zahlen

  s = 0;
  cout << "Bitte Zahl eingeben (Stopp mit 0): ";
  cin >> a;

  while (a != 0) {
    s = s + a;
    cout << "Bitte Zahl eingeben (Stopp mit 0): ";
    cin >> a;
  }
  cout << "Die Summe ist: " << s << endl;
}
```

In der Schleife hier kontrolliert die Bedingung nicht mehr wie oben, ob die Zahl der Durchläufe eine vorgegebene Grenze erreicht hat. Sie kontrolliert, ob als letztes eine Null eingegeben wurde. Da zuerst ein Wert eingelesen werden muss, bevor wir prüfen können, ob er gleich Null ist, wurde die erste Leseoperation aus der Schleife herausgezogen. Man beachte auch, dass hier, anders als im letzten Beispiel, innerhalb der Schleife zuerst aufsummiert und dann gelesen wird.

4.2 N Zahlen aufaddieren

Beispiel: Die ersten n ganzen Zahlen aufsummieren

Die Konstruktion korrekter Schleifen ist ein erster wichtiger Schritt auf dem Weg zur systematischen Programmerstellung und mathematische Folgen und Reihen bieten hier ein ausgezeichnetes Übungsmaterial.[10]

[10] Informatiker zeichnen sich gegenüber angelernten Programmierern dadurch aus, dass sie komplexe Programme systematisch konstruieren können und nicht stümperhaft an einem Programm solange "herumdoktern" bis es funktioniert oder sie keine Lust mehr haben.

Wir beginnen mit der Aufsummierung der ersten n Zahlen:

$$s = \sum_{i=1}^{n} i$$

Bei der Konstruktion eines Programms ist es wichtig Variablen und die in ihnen enthaltenen Werte zu unterscheiden. Natürlich besteht zwischen beidem ein Zusammenhang: eine bestimmte Variable enthält einen bestimmten Wert oder soll irgendwann einen bestimmten Wert enthalten. Variablen und Werte sind aber unterschiedliche Dinge. Welche Variablen welche Werte enthalten wird von der Lösungsstrategie – dem *Algorithmus* – des Programms bestimmt.

Das Programm zu unserer kleinen Aufgabe ist:

```
#include <iostream>

using namespace std;

int main () {
  int n,     // Summation bis hierher
      s,     // Summe bisher
      i;     // zuletzt aufsummierte Zahl

  cout << "n = ";
  cin >> n;

  i = 0;
  s = 0;
  while (i < n) {
    i = i+1;
    s = s + i;
  }
  cout << "Summe 1 bis " << n << " = " << s << endl;
}
```

Werte und Variablen in der Summationsaufgabe

Bei der Berechnung der Summe spielen folgende *Werte (!)* eine Rolle:

- n: Die Zahl bis zu der summiert werden soll. Diese Zahl wird eingelesen.
- i: i steht für die Folge der Zahlen $i_0 = 0, i_1 = 1, i_2 = 2, \ldots i_n = n$.
- s: s steht für die Folge der Zahlen $s_0 = 0, s_1 = 1, s_2 = 3, \ldots s_n = \sum_{i=1}^{n} i$.

Es gibt also einen Wert n, viele i–Werte ($i_0, i_1, \ldots$) und viele s–Werte ($s_0, s_1, \ldots$).

Die Werte haben jeweils einen Platz in einer *Variablen (!)*. Manche Variablen enthalten immer den gleichen Wert, andere Variablen enthalten Wertefolgen:

- `n` enthält den Wert n
- `i` enthält im Laufe des Programms die Folge der i–Werte
- `s` enthält im Laufe des Programms die Folge der s–Werte.

`s` enthält erst am Ende des Programms den gesuchten Wert s. Vorher sind in ihm Teilsummen enthalten.

Den sich verändernden Inhalt der Variablen macht man sich am besten in einer Wertverlaufstabelle klar. Hier sind die Werte der Variablen jeweils beim Test der Schleifenbedingung zu finden (Eingabe $n = 5$):

n	i	s
5	0	0
5	1	1
5	2	3
5	3	6
5	4	10
5	5	15

Schleifenbedingung und Reihenfolge der Anweisungen

Bei einer Schleife können kleine harmlos aussehende Änderungen aus einem funktionierenden Programm ein falsches machen. Vertauscht man beispielsweise die beiden Anweisungen in der Schleife

```
#include <iostream>

using namespace std;

int main () {
  int i = 0;
  int s = 0;
  while (i < n) {
    s = s + i;
    i=i+1;
  }
  cout << "Summe der eingebenen Zahlen: " << n << " = " << s << endl;
}
```

dann gibt das Programm ein falsches Ergebnis aus: das letzte i fehlt in der Summe. Mit einer Veränderung der Schleifenbedingung zu `i <= n` kann der Fehler wieder repariert werden. Die Reihenfolge der Anweisungen in der Schleife und die Schleifenbedingung müssen also zusammen passen:

```
OK:                    OK:                  FALSCH:
Kleiner-gleich;        Kleiner;             Kleiner-gleich;
Zuerst addieren,       zuerst erhoehen,     zuerst erhoehen,
dann erhoehen:         dann addieren:       dann addieren:

while (i <= n) {       while (i < n) {      while (i <= n) {
  s = s + i;             i=i+1;               i=i+1;
  i=i+1;                 s = s + i;           s = s + i;
}                      }                    }
```

Einlesen, bis eine Null eingegeben wird

Statt der ersten n Zahlen, können wir auch eingelesene Zahlen addieren. Im folgenden Beispiel werden Zahlen eingelesen und aufaddiert. Die Schleife läuft so lange, bis eine 0 eingeben wird:

```
#include <iostream>

using namespace std;

int main () {
  int x;
  int s = 0;
  cin >> x;
  while ( x != 0 ) {
    s = s+x;
    cin >> x;
  }
  cout << "Summe der eingebenen Zahlen: " << s << endl;
}
```

4.3 Schleifenkontrolle: break und continue

Abbruch mit break

Die `break`–Anweisung ist uns bereits in Zusammenhang mit der `switch`–Anweisung begegegnet: Mit `break` wird die `switch`–Anweisung sofort verlassen. Zum gleichen Zweck kann `break` in einer Schleife benutzt werden. Beispiel:

```
 while ( true ) {
    int x;
    cout << "Bitte eine Zahl eingeben, Stopp mit 0:";
```

```
    cin >> x;
    if ( x == 0 ) break;    // Abbruch, Schleife verlassen
    ... x != 0 verarbeiten ...
}
```

Bei ineinander geschachtelten Schleifen wird nur die innerste verlassen.

```
while ( ... ) {       // Schleife 1
    ...
       while ( ... ) { // Schleife 2
       ...
       break;          // verlaesst Schleife 2, weiter in Schleife 1
       ...
      }
    ...
}
```

Weiter mit continue

Während mit `break` eine Schleife insgesamt abgebrochen wird, beendet `continue` nur den aktuellen Schleifendurchlauf. Beispiel:

```
while ( true ) {
   int x;
   cout << "Bitte eine positive Zahl eingeben:";
   cin >> x;
   if ( x <= 0 ) {
     cout << "Das war wohl nichts!" << endl;
     continue;                  // Abbruch des Durchlaufs, weiter mit Schleife
   }
   ... x > 0 verarbeiten ...
}
```

Wird hier eine negative Zahl oder Null eingegeben, dann wird der Schleifendurchlauf, aber nicht die ganze Schleife abgebrochen. Es geht weiter mit einer erneuten Eingabeaufforderung.

`break` und `continue` können in jeder Art von Schleife benutzt werden, nicht nur in `while`–Schleifen.

4.4 Die For–Schleife

For–Schleife: Eine Folge durchlaufen

Mit einer While–Schleife wird eine Aktion wiederholt solange eine Bedingung zutrifft. Mit der For–Schleife wird eine vorherbestimmte Werte–Folge durchlaufen und für jeden Wert der Schleifenkörper ausgeführt.

Beispielsweise erzeugt die Schleife

```
for (int i = 0; i < 5; ++i)
  cout << "Hallo Nr " << i << endl;
```

die Ausgabe “Hallo Nr 0”, ... “Hallo Nr 4”.

`i` ist hier die sogenannte *Laufvariable*. Sie durchläuft die Werte 0, 1, 2, 3 und 4 und bei jedem Durchlauf wird die Anweisung ausgeführt. Ein etwas komplexeres Beispiel ist:

```
int max,       // bisher gefundes Maximum
int v;
for (int k = 0; k < 10; ++k) {
   cin >> v;
   if ( (v > max) || (k == 0) ) max = v;
}
cout << max << " ist die groesste der eingebenen Zahlen" << endl;
```

Hier werden zehn Zahlen eingelesen und die die größte Zahl dann ausgegeben. Noch etwas komplexer ist:

```
int max,       // bisher gefundes Maximum
    maxIndex;  // Nummer des bisher gefundenen Maximums
for (int k = 1; k <= 10; ++k) {
   int v;
   cin >> v;
   if (k == 1) {
      max = v; maxIndex = 1;
   } else
      if (v > max) {
         max = v; maxIndex = k;
      }
}
cout << max << " die Zahl Nr. " << maxIndex << " ist die groesste" << endl;
```

Hier werden mit Hilfe einer For–Schleife 10 Zahlen eingelesen und die Nummer der größten festgestellt. Wir sehen, dass die Anweisung eine zusammengesetzte sein kann, dass die Laufvariable nicht unbedingt `i` heißen muss und auch nicht unbedingt von 0 an laufen muss.

Allgemeine Form der For–Anweisung

Die For–Anweisung hat folgende allgemeine Form:

$$\texttt{for (} < Init - Anweisung > \texttt{;} < Bedingung > \texttt{;} < Ausdruck > \texttt{)} < Anweisung > \texttt{;}$$

Die Wirkung der For–Anweisung macht am besten ein Flussdiagramm klar (Siehe Abbildung 14).

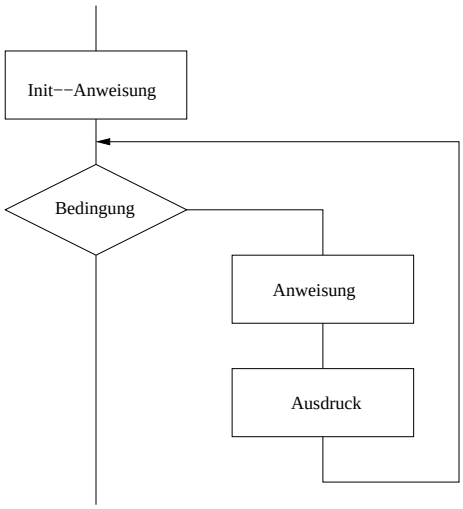


Abbildung 14: For–Anweisung

An ihm erkennt man, dass die For–Schleife eigentlich eine verkleidete While–Schleife ist:

```
for (Init-Anweisung; Bedingung; Ausdruck) Anweisung;
```

ist nichts anderes als

```
Init-Anweisung; while (Bedingung) {Anweisung; Ausdruck;}
```

Die Schleife

```
for (i = 0; i < 10; ++i)
  cout << i;
```

entspricht darum exakt

```
i = 0;
while (i < 10) {
  cout << i;
  ++i;
}
```

Initialisierungsanweisung und Laufvariablen

Eine For–Schleife beginnt mit der *Initialisierungsanweisung*. In

```
for (int i=0; ...) ...
```

ist `int i=0;` die Initialisierungsanweisung. In ihr wird normalerweise die Laufvariable definiert und initialisiert. Eine so *lokal* definierte Laufvariable kann nur innerhalb der Schleife verwendet werden. Folglich ist

```
for (int i = 0; i < 5; ++i)
  cout << "Hallo Nr " << i << endl; // OK
cout << "i = " << i << endl;        // FALSCH i ist hier nicht definiert !
```

fehlerhaft und wird vom Compiler nicht akzeptiert.

Eine For–Schleife kann auch eine "normale" Variable als Laufvariable benutzen. Die Laufvariable wird dann außerhalb der Schleife definiert und in der Initialisierungsanweisung nur noch initialisiert:

```
int i;
...                                  // beliebige Verwendung von i
for (i = 0; i < 5; ++i)              // i ist Laufvariable der Schleife
  cout << "Hallo Nr " << i << endl; // OK
cout << "i = " << i << endl;        // OK, aber Wert ungewiss
...                                  // beliebige Verwendung von i
```

In diesem Fall kann sie natürlich auch außerhalb der Schleife verwendet werden. Man kann sich hier nicht unbedingt darauf verlassen, dass die Variable nach der Schleife einen bestimmten Wert hat. Im Beispiel oben kann `i` den *Abbruchwert* der Schleife (5) haben, das muss aber nicht sein.

Laufvariablen gleichen Namens können in verschiedenen Schleifen lokal definiert werden. Gleichzeitig kann es sogar noch eine Variable mit diesem Namen außerhalb der Schleife geben:

```
float i = 1;               // i Nr. 1
for (int i=20; i<25; ++i) // i Nr. 2
    cout << i << endl;     // i Nr. 2

for (int i=30; i<35; ++i) // i Nr. 3
    cout << i << endl;     // i Nr. 3

cout << i << endl;         // i Nr. 1
```

Hier werden drei verschiedene `i`–s definiert, die sich gegenseitig aber nicht in die Quere kommen.[11]

Der Operator++

In allen Beispielen zur For–Schleife kommt der *Operator++* vor. (Er wird auch *Inkrement–Operator* genannt.) Z.B. `++i`. Das ist hier nur eine Kurzschreibweise für `i = i+1`. Später werden wir uns noch etwas intensiver mit Operatoren beschäftigen, aber hier wird so etwas wie

```
for (int i = 0; i < 10; ++i) ...
```

nur als Kurzform von

```
for (int i = 0; i < 10; i=i+1) ...
```

verwendet. Die genaue Bedeutung von `++i` ist: `i` erhöhen, dann den Wert liefern. Wenn beispielsweise `i` den Wert 5 hat, dann liefert der Ausdruck `++i` den Wert 6 und erhöht nebenbei auch noch `i` auf 6.

Ausdrücke wie `++i` sind auch nicht auf die For–Schleife beschränkt, sie können an jeder Stelle im Programm statt `i=i+1` verwendet werden. Umgekehrt muss in einer For–Schleife nicht unbedingt so etwas wie `++i` auftauchen. Jeder Ausdruck ist erlaubt.[12]

Komplementär zum Inkrementoperator gibt es den Operator`--`, den *Dekrementoperator*:

```
--i
```

entspricht als Anweisung:

[11] Ältere Compiler können mit derartigen Konstruktionen ihre Probleme haben. Sie entsprechen aber dem offiziellen Sprachstandard.
[12] In C++ sind Anweisungen eine Sonderform der Ausdrücke. Normalerweise steht hier auch eine "richtige" Anweisung, z.B. eine Zuweisung.

```
i = i - 1;
```

Neben `++i` gibt es noch die Form `i++`. `++i` ist der *Pre–Inkrementoperator* und `i++` der *Post–Inkrementoperator*. Beide erhöhen den Wert der Variablen in gleicher Weise, aber `i++` hat als Wert den Wert von `i` vor der Erhöhung. Es erhöht *nachdem* der Wert bestimmt wurde (*Post*–Inkrement). Mit `i = 5` hat `i++` beispielsweise den Wert 5 und erhöht `i` auf 6; Der Pre–Inkrementoperator `++i` dagegen erhöht (*Pre*–Inkrement) `i` auf 6 *bevor* 6 als Wert brechnet wird.

`++i`	Pre–Inkrement	i erhöhen (inkrementieren), dann Wert bestimmen
`i++`	Post–Inkrement	Wert bestimmen, dann i erhöhen (inkrementieren)

Der Unterschied macht sich nur bei einer Verwendung als Ausdruck bemerkbar. Als Anweisungen sind beide völlig gleichwertig.[13]

Die Schleifenbedingung

Genau wie die While–Schleife wird auch die For–Schleife durch eine Bedingung gesteuert:

```
for (int i = 0; i < 10; ++i) ...
```

läuft solange wie `i < 10` den Wert `true` hat. Ebenfalls wie bei While wird die Schleife gar nicht erst betreten, wenn die Bedingung am Anfang nicht erfüllt ist. Man darf auch nicht vergessen, dass im Schleifenkörper die Laufvariable verändert und damit die Bedingung beeinflusst werden kann. So wird etwa die Schleife

```
for (int i = 0; i < 10; ++i) i=i-1;
```

niemals terminieren (= enden).

4.5 Die Do-While–Schleife

Beispiel

Die Do-While–Schleife stellt eine weitere Variante der Schleifen in C++ dar. Ihre Wirkung kann leicht an einem Beispiel gesehen werden:

```
do {
  cout << "naechster Wert, Stop mit 0: ";
  cin >> i;
  sum = sum + i;
} while ( i != 0);
```

Hier werden so lange Zahlen eingelesen und aufsummiert, bis eine 0 angetroffen wird. Im Unterschied zur While–Schleife wird bei dieser Form die Bedingung nicht jeweils vor sondern *nach* der Anweisung geprüft. Die Do–While–Schleife setzt man darum immer dann ein, wenn wie hier der Schleifenkörper in jedem Fall mindestens einmal betreten werden muss.

Allgemeine Form

Die Do-While–Schleife hat folgende allgemeine Form:

`do` $<$ *Anweisung* $>$ `while (` $<$ *Bedingung* $>$ `);`

4.6 Schleifenkonstruktion: Zahlenfolgen berechnen und aufaddieren

Beispiel: beliebige Zahlenfolgen aufsummieren

Neben Erfahrung und ein wenig Begabung ist auch etwas Systematik eine nützliche Zutat bei der Konstruktion einer Schleife. Angenommen, wir wollen die Summe

$a_1 + a_2 + a_3 + ...a_n$

[13] In C++ wird allgemein das Pre–Inkrement bevorzugt (++i statt i++): Er ist bei komplizierteren Ausdrücken – die in C nicht möglich sind – wesentlich effizienter und auch leichter zu implementieren. C++ hätte darum eigentlich "++C" genannt werden müssen.

mit

$$a_n = a_{n-1} + c$$

berechnen und prüfen, ob die Gleichung

$$a_1 + a_2 + a_3 + ...a_n = (n(a_1 + a_n))/2$$

für beliebige c, a_1 und n korrekt ist. Für $c = 2$, $n = 6$ und $a_1 = 0$ gilt beispielsweise

$$0 + 2 + 4 + 6 + 8 + 10 = 30$$

und

$$(6 * (0 + 10))/2 = 30$$

Dazu schreiben wir ein Programm, das die Zahlenfolge für beliebige Werte $a_1,\ n$ und c aufsummiert und diese Summe sowie den Formelwert $(n(a_1 + a_n))/2$ ausgibt. Zuerst wird die Folge durch Eingabe von a_1, n und c festgelegt. Dann werden die a_i berechnet und aufsummiert und schließlich das Ergebnis mit dem Wert der Formel verglichen.

Der erste Programmentwurf nach diesen Vorgaben ist:

```
...
int main () {
  int n,     // letzter Index
      c,     // Konstante
      s,     // berechnete Summe
      a_1,   // Anfang: 1-ter Summand
      a_n;   // Ende: letzter Summand

  //a_1, c, n einlesen:
  cout << "a_1 = "; cin  >> a_1;
  cout << "n= ";    cin  >> n;
  cout << "c= ";    cin  >> c;

  //s und a_n berechnen
  //??

  // s und (n*(a_1+a_n))/2 ausgeben:
  cout << "Summiert:  " << s << endl;
  cout <<" Berechnet: " << (n*(a_1+a_n))/2 << endl;
}
```

Dieser Programmentwurf dient in erster Linie dazu, sich über die Funktion der wichtigsten Variablen klar zu werden: Welche Werte werden wohin eingelesen und welche Werte müssen daraus berechnet und in welchen Variablen abgelegt werden.

Summe berechen: der “händische Algorithmus” ist für das Programm nicht geeignet

Es fehlt noch die Berechnung der Summe

$$S = a_1 + a_2 + a_3 + ...a_n$$

und des letzten Summanden

$$a_n = a_1 + c + c + \ldots$$

Bevor eine weitere Zeile Programmcode geschrieben wird, testet man unbedingt sein Problemverständnis, indem mindestens ein Beispiel *per Hand* gerechnet wird. Wir berechnen also als Beispiel die Summe für die Werte $a_1 = 0$, $n = 4$ und $c = 2$:

- Schritt 1: die Summenglieder bestimmen und aufschreiben:

  $0 \quad 0 + 2 = 2 \quad 2 + 2 = 4 \quad 4 + 2 = 6$

- Schritt 2: Summe berechnen:

  $0+0=0 \quad 0+2=2 \quad 2+4=6 \quad 6+6=12$

Dieser "händische Algorithmus" ist für das Programm *nicht* geeignet: Wir haben im Programm keinen Platz um eine beliebig lange Wertefolge abzulegen. In einzelnen Variablen kann zwar jeweils ein Wert abgelegt werden, aber selbst wenn wir 100 Variablen für die Wertefolge vorsehen, kann es sein, dass der Benutzer 101 als Wert für n eingibt.

Summanden und Teilsummen gleichzeitig berechnen

Der Summationsalgorithmus, der für unser Programm geeignet ist, besteht darin, sukzessive neue a_i–Werte und gleichzeitig weitere Teilsummen s_i zu berechnen:

	Schritt 1	Schritt 2	Schritt 3	Schritt 4
Summand a_i:	0	$0+2=2$	$2+2=4$	$4+2=6$
Teilsumme s_i:	$0+0=0$	$0+2=2$	$2+4=6$	$6+6=12$

In jedem Schritt werden genau zwei Werte gebraucht: a_i und s_i. Egal wie groß n ist, also wieviele Schritte ausgeführt werden müssen, wir kommen also mit zwei Variablen aus! Etwas formaler ausgedrückt, haben wir jetzt folgenden Algorithmus angewendet:

- Neue Summanden a_i werden aus alten a_{i-1} durch Addition von c gebildet: $a_i = a_{i-1} + c$
- Neue Teilsummen s_i werden durch Addition des neuen Summanden und der alten Teilsumme gebildet: $s_i = s_{i-1} + a_i$

Diese beiden Berechnungen werden so lange wie notwendig wiederholt. Sie sind die Aufgabe einer Schleife. Im Schleifenkörper müssen also s_i und a_i berechnet werden:

$a_i = a_{i-1} + c$
$s_i = s_{i-1} + a_i$

Wertverlaufstabelle der a_i und s_i

Die a_i und s_i sind Werte–*Folgen*. Die Wertefolgen werden in jeweils einer Variablen abgelegt: Die Folge der a_i in der Variablen `a`, die Folge der s_i in der Variablen `s`. (Nicht vergessen: Variablen und ihre wechselnden Werte sind streng zu unterscheiden!) Wir schreiben zunächst eine Wertverlaufstabelle für `a` und `s` auf. (Werte jeweils vor dem Test der Bedingung):

`a`	`s`
a_1	$s_1 = a_1$
$a_2 = a_1 + c$	$s_2 = a_1 + a_2$
$a_3 = a_1 + c + c$	$s_3 = a_1 + a_2 + a_3$
$a_4 = a_1 + c + c + c$	$s_4 = a_1 + a_2 + a_3 + a_4$
..	..

Die Berechnung des letzten Summanden a_n fehlt noch. Wir können sie uns aber ersparen, da a_n der letzte Wert von `a` ist.

Konstruktion des Schleifenkörpers aus der Wertverlaufstabelle

Aus dieser Tabelle wollen wir jetzt eine Schleife konstruieren. Jede Zeile stellt die Belegung der beiden Variablen zum Zeitpunkt des Tests der Bedingung dar. Der Übergang von einer Zeile zur nächsten beinhaltet darum die Wirkung der Anweisungen in der Schleife. Die Anweisungen sind gesucht. Also ist die Frage, welche Anweisungen führen uns von einer Zeile zur anderen?

Von a_0 zu a_1 und von a_1 zu a_2 etc. kommt man, indem jeweils c zum Inhalt von `a` addiert wird. Zum Inhalt von `s` muss dann dieser neue Wert von `a` addiert werden und man kommt zur nächsten Zeile von `s`:

```
//s und a_n berechnen:
int a = ...;
...
while (...) {
  a = a + c;  //neues a = altes a + c
  s = s + a;  //neues s = altes s + neues a
}
a_n = a;
```

Das “neue a” ist der neue Inhalt der Variablen `a`. In der Tabelle ist es in der neuen Zeile in der `a`–Spalte zu finden. Das “neue s” wird aus dem “alten s” (eine Zeile weiter oben) und dem “neuen a” (gleiche Zeile) berechnet.

Der Schleifenkörper hat also die Aufgabe den Übergang von einer Zeile zur nächsten in der Wertverlaufstabelle zu bewerkstelligen (Siehe Abbildung 15).

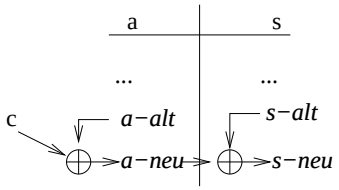


Abbildung 15: Schleifenkörper als Zeilenübergang

Berechnung einer neuen Zeile:

- Zuerst wird a_{neu} brechnet: $a_{neu} \leftarrow a_{alt} + c$
- Daraus dann s_{neu}: $s_{neu} \leftarrow a_{neu} + s_{alt}$

Schleifeninitialisierung

Die Schleife muss mit den korrekten Anfangswerten beginnen. Diese können sofort aus der ersten Zeile der Tabelle entnommen werden:

```
//s und a_n berechnen:
int a
a = a_1; // Belegung der
s = a_1; // ersten Zeile
while (...) {
  a = a + c;
  s = s + a;
}
a_n = a;
```

Schleifenbedingung

Die Bedingung für den Abbruch der Schleife kann nicht der Tabelle entnommen werden. Wir müssen entweder die Zahl der Durchläufe oder den Index der addierten Summanden mitprotokollieren. Entscheidet man sich für den Index des zuletzt addierten Summanden, dann bekommt die Tabelle folgende weitere Spalte:

a	s	i
a_1	$s_1 = a_1$	1
$a_2 = a_1 + c$	$s_2 = a_1 + a_2$	2
$a_3 = a_1 + c + c$	$s_3 = a_1 + a_2 + a_3$	3
$a_4 = a_1 + c + c + c$	$s_4 = a_1 + a_2 + a_3 + a_4$	4
..	..	..

Jede Zeile beschreibt den Zustand des Programms beim Test der Schleifenbedingung. Wir können darum stoppen, wenn `i` den geforderten Wert n erreicht hat:

```
//s und a_n berechnen:
int a = a_1;
int i = 1;
s = a_1;
while (i != n) {
  a = a + c;
  s = s + a;
  i = i + 1;
}
a_n = a;
```

Man beachte die Reihenfolge der Anweisungen innerhalb der Schleife. Es ist wichtig, dass zuerst `a` und dann `s` verändert wird, da der neue Wert von `s` den neuen Wert von `a` benötigt.

Der Vollständigkeit halber noch das gesamte Programm:

```
#include <iostream>

using namespace std;

int main () {
  int n,     // letzter Index
      c,     // Konstante
      s,     // berechnete Summe
      a_1,   // Anfang: 1-ter Summand
      a_n;   // Ende: letzter Summand

  //a_1, c, n einlesen:
  cout << "a_1 = "; cin  >> a_1;
  cout << "n= ";    cin  >> n;
  cout << "c= ";    cin  >> c;

  //s und a_n berechnen
  int a = a_1;
  int i = 1;
  s = a_1;
  while (i != n) {
    a = a + c;
    s = s + a;
    i = i + 1;
  }
  a_n = a;

  // s und (n*(a_1+a_n))/2 ausgeben:
  cout << "Summiert:  " << s << endl;
  cout << "Berechnet: " << (n*(a_1+a_n))/2 << endl;
}
```

Die Grundbestandteile einer Schleife

Schleifen haben generell folgende *Grundbestandteile:*

1. *Die relevanten Variablen:* Die Variablen, die in der Schleife verwendet werden.
2. *Schleifeninitialisierung:* Die relevanten Variablen müssen mit den richtigen Initialwerten (ersten Werten) belegt werden.
3. *Schleifenbedingung:* Wann endet die Schleife?
4. *Schleifenkörper:* Wie werden die relevanten Variablen in einem Schleifendurchlauf verändert?

Die relevanten Variablen sollten immer als erstes festgelegt werden. Natürlich überlegt man auch welche - in der Regel wechselnden (!) – Werte sie enthalten sollen.

Als nächstes sollte der Schleifenkörper konstruiert werden. Dazu überlegt man systematisch, welche alten Werte die Variablen enthalten und wie aus den alten Werten die neuen berechnet werden können. Am besten benutzt man dazu eine Wertverlaufstabelle.

Als nächstes überlegt man welches die richtigen Initialwerte für die Schleife sind. Hat man eine Wertverlaufstabelle, dann nimmt man einfach deren erste Zeile.

Schließlich kann die Schleifenbedingung formuliert werden.

4.7 Rekurrenzformeln berechnen

Rekurrenzformel

Eine Rekurrenzformel definiert eine Funktion mit Hilfe von *Rekursion*, also durch Bezug des zu definierenden auf sich selbst. Ein einfaches Beispiel ist die Definition einer Funktion S:

$$\begin{aligned} S(0) &= 0 \\ S(n) &= S(n-1) + n \end{aligned}$$

Die Interpretation einer solchen Definition ist offensichtlich:

$$S(3) = S(2) + 3 = S(1) + 2 + 3 = S(0) + 1 + 2 + 3 = 0 + 1 + 2 + 3$$

Soll $S(x)$ für für ein beliebiges x berechnet werden, dann muss die Reihe der Werte $S(0), S(1), ...S(x-1)$ jeweils berechnet werden. Alle Berechnungen folgen der Definition und damit dem gleichen Schema. Wiederholungen sind ein Fall für Schleifen. Das Problem wird also mit einer Schleife gelöst.

Rekursion und Iteration

Die Rekursion geht "rückwärts", "von oben" an den gesuchten Wert heran. $S(n)$ wird auf $S(n-1)$, dieses auf $S(n-2)$ etc. zurückgeführt. Eine Schleife dagegen geht "von unten" vor: aus $S(0)$ wird $S(1)$, aus $S(1)$ wird $S(2)$ berechnet etc. bis zu $S(n)$.

Die Art der Wiederholung die mit Schleifen möglich ist, nennt man *Iteration*. Eine Iteration *ist* eine Schleife. Eine Rekurrenzformel beschreibt ebenfalls ein Berechnungsverfahren, das auf Wiederholung beruht. Während die Iteration (Schleife) sich jedoch sozusagen "von vorn nach hinten" zum Ergebnis bewegt, geht die Rekursion "vom Ziel aus zurück zu den Anfängen".

Funktion S iterativ berechnen

In vielen Fällen kann man aus der Rekursion einfach eine Iteration machen und so eine Rekurrenzformel in einer Schleife berechnen. Beispielsweise die Funktion S.

Jede Berechnung von $S(x)$ für ein beliebiges (ganzzahliges) x (größer–gleich 0) beginnt mit $S(0)$ und geht dann über $S(1), S(2), \ldots S(x-1)$ zu $S(x)$. Als Wertverlaufstabelle:

s	i	x
$S(0) = 0$	0	x
$S(1) = 0 + 1$	1	x
$S(2) = 0 + 1 + 2$	2	x
$S(3) = 0 + 1 + 2 + 3$	3	x
..	...	...

Die Rekurrenzformel definiert dabei den Übergang von einem Wert zu nächsten. Daraus kann sofort eine Schleife konstruiert werden:

```
int s = 0;  // S(0)
int i = 0;
while (i != x) {
  i = i + 1;
  s = s + i;
}
```

Vergleich rekusive und iterative Berechnung

Die rekursive Definition beschreibt, wie der gesuchte Wert "von oben nach unten und wieder zurück" berechnet wird: Man geht nach unten bis zur Basis und sammelt dann auf dem Rückweg die Werte auf. Betrachten wir die Berechnung von $S(3)$:

Analysieren		Aufsammeln	
$S(3) = S(2) + 3$	$\downarrow 3$	$\uparrow 6$	4. Wert: $3 + 3$
$S(2) = S(1) + 2$	$\downarrow 2$	$\uparrow 3$	3. Wert: $1 + 2$
$S(1) = S(0) + 1$	$\downarrow 1$	$\uparrow 1$	2. Wert: $0 + 1$
$S(0) = 0$	$\rightarrow 0$	$\uparrow 0$	1. Wert: 0

Das Argument wird solange gemäß der Formel zerlegt, bis die Basis erreicht wird. Bei dieser Analyse muss man sich die Rechenschritte merken, die dann anschließend auszuführen sind. Das Verfahren hat also zwei Phasen: Analyse (Runtergehen) und Aufsammeln (Raufgehen).

Die iterative Berechnung hat dagegen nur eine Richtung und eine Phase; sie sammelt nur auf:

$S(0) = 0$	$\downarrow 0$	1. Wert: 0
$S(1) = S(0) + 1$	$\downarrow 1$	2. Wert: $0 + 1$
$S(2) = S(1) + 2$	$\downarrow 3$	3. Wert: $1 + 2$
$S(3) = S(2) + 3$	$\downarrow 6$	4. Wert: $3 + 3$

Beispiel: Berechnung der Potenzbildung

Ein weiteres Beispiel für eine rekursiv definierte Funktion ist die Potenzbildung:

$$\begin{aligned} x^0 &= 1 \\ x^n &= x^{n-1} * x \end{aligned}$$

Die Potenz kann mit der gleichen Strategie wie die Funktion S von oben iterativ berechnet werden. Wir definieren eine Variable `p` und bilden in ihr die Reihe der Werte $p_0 = x^0,\ p_1 = x^1,\ \ldots\ p^n = x^n$:

`p`	`i`	`n`
$p_0 = 1$	0	n
$p_1 = p_0 * x$	1	n
$p_2 = p_1 * x$	2	n
$p_3 = p_2 * x$	3	n
..	...	...
$p_n = p_{n-1} * x$	n	n

Die erste Zeile definiert die Initialwerte der Schleife, die letzte die Schleifenbedingung und der Schleifenkörper entspricht dem Übergang von einer Zeile zur nächsten.

```
i = 0;
p = 1;
while (i != n) { // p == x^i
  i = i + 1;
  p = p * x;
}
```

Beispiel: Berechnung des GGT

Der größte gemeinsame Teiler (ggt) von zwei ganzen positiven Zahlen lässt sich nach folgender Rekurrenzformel berechnen:

$$ggt(a,b) = \begin{cases} a & a = b \\ ggt(a-b,b) & a > b \\ ggt(a,b-a) & b > a \end{cases}$$

Auch hierzu kann man eine Tabelle angeben. Beispielsweise für die Berechnung von $ggt(120, 48)$:

a	b
120	48
72	48
24	48
24	24

a und b sind die Variablen, sie werden mit der Eingabe initialisiert und die Schleife endet, wenn beide den gleichen Wert haben:

```
// a und b enthalten positive ganze Zahlen a_0, b_0
while (a != b) {
  if (a > b) a = a-b;
  if (b > a) b = b-a;
}
// a enthaelt ggt (a_0, b_0)
```

4.8 Berechnung von e

e als Grenzwert

Die Eulersche Konstante e kann als Grenzwert definiert werden:

$$e = \lim_{n \to \infty} a_n$$

mit

$$a_n = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots \frac{1}{n!}$$

Mit dieser Formel kann der Wert von e mit beliebiger Genauigkeit angenähert werden. Nehmen wir an e sollte mit der Genauigkeit ϵ berechnet werden. Wir müssen dazu nur a_n für steigende n berechnen bis irgendwann einmal $a_n - a_{n-1} < \epsilon$.

Programmentwurf

Ein erster Entwurf ist:

```
const float eps = ...;
float a, // a_n
      d; // a_n - a_n-1
int   n;

? Initialisierung ?
while (d > eps) {
  n = n + 1;
  ? neues a berechnen ?
  ? neues d berechnen ?
}
```

Rekurrenzformel für a_n

Wie man aus der Definition oben sieht kann ein Summand a_n leicht aus a_{n-1} berechnet werden:

$$\begin{aligned} a_0 &= 1 \\ a_n &= a_{n-1} + \frac{1}{n!} \end{aligned}$$

Damit haben wir eine Rekurrenzformel für a_n. Rekurrenzformeln sind immer gut. Man kann sie in der Regel gut in einen Schleifenkörper umsetzen.

Rekurrenzformel für $\frac{1}{n!}$

Die Berechnung von $\frac{1}{n!}$ ist aufwendig und muss für steigende n beständig wiederholt werden. Es stellt sich die Frage, ob $\frac{1}{n!}$ nicht einfacher aus $\frac{1}{(n-1)!}$ berechnet werden kann. Natürlich kann es:

$$\begin{aligned} q_0 &= \frac{1}{1!} = 1 \\ q_n &= \frac{1}{n!} = \frac{1}{(n-1)!} * \frac{1}{n} = q_{n-1} * \frac{1}{n} \end{aligned} \qquad \begin{aligned} a_0 &= 1 \\ a_n &= a_{n-1} + q_n \end{aligned}$$

Damit haben wir eine zweite Rekurrenzformel und können somit q_n aus q_{n-1} und dann a_n aus a_{n-1} berechnen.

Die Schleife

Zur Verdeutlichung des aktuellen Stands der Überlegungen zeigen wir einen Ausschnitt aus der Wertverlaufstabelle:

n	q	a	d	eps
0	1	1	???	ϵ
...	...	...	...	...
$n-1$	$q_{n-1} = \frac{1}{(n-1)!}$	a_{n-1}	$a_{n-1} - a_{n-2}$	ϵ
n	$q_n = q_{n-1} * \frac{1}{n}$	$a_n = a_{n-1} + q_n$	$a_n - a_{n-1}$	ϵ
...	...	...	...	...

Man erkennt unmittelbar, dass die Variable `d` überflüssig ist: ihr Wert entspricht exakt dem von `q`. Damit können wir die Schleife sofort aufschreiben und den Programmentwurf zum vollständigen Programm erweitern:

```
#include <iostream>

using namespace std;

int main () {
  const float eps = 0.00001;
  float a, q;
  int   n;

  q = 1; a = 1; n = 0;
  while (q > eps) {
    n = n + 1;
    q = q/n;
    a = a + q;
  }
  cout << "e = " << a << endl;
}
```

4.9 Die Schleifeninvariante

Schleifeninvariante: Zusicherung in einer Schleife

In Kapitel 2 haben wir "Zusicherung" definiert, als Aussage über die aktuelle Belegung der Variablen, also über den (Programm–) Zustand. Eine *Schleifeninvariante* (kurz *Invariante*) ist eine Zusicherung, die den Zustand am Anfang eines Schleifenkörpers beschreibt. Bei einer Zusicherung versucht man wichtige Informationen über die Variablenbelegung zum Ausdruck zu bringen. Was ist wichtig bei einer Schleife und was sollte darum in der Invariante zum Ausdruck gebracht werden? Erstaunlicherweise ist die Fachwelt sich sicher, dass die wichtige Information über eine Schleife über das Auskunft gibt, was sich *nicht ändert*! Daher hat sie auch ihren Namen: Die *Invariante* beschreibt was *invariant*, also unveränderlich ist. Kurz: Eine Invariante ist eine Zusicherung in einer Schleife, die invariante Eigenschaften der Variablenwerte beschreibt.

Beispiel: GGT

Als Beispiel betrachten wir die Schleife zur Berechnung des GGT (siehe weiter oben).

```
// a und b enthalten positive ganze Zahlen a_0, b_0
while (a != b) {
  if (a > b) a = a-b;
  if (b > a) b = b-a;
}
// a enthaelt ggt (a_0, b_0)
```

Die Variablen `a` und `b` werden mit positiven ganzen Zahlen initialisiert und am Ende der Schleife enthält `a` den gesuchten GGT. Warum ist das so? Wir erkennen auf den ersten Blick die Dynamik der Schleife, also das was sich in ihr bewegt: der kleinere Wert wird wird von größeren abgezogen. Ein entsprechender Kommentar ist völlig überflüssig:

```
// a und b enthalten positive ganze Zahlen a_0, b_0
while (a != b) {
  // (Ueberfuessiger Kommentar zur Programm-DYNAMIK:)
  // Der kleinere Wert wird vom groesseren abgezogen
  if (a > b) a = a-b;
  if (b > a) b = b-a;
}
// a enthaelt ggt (a_0, b_0)
```

Das Offensichtliche muss und soll nicht kommentiert werden. Ein sinnvoller Kommentar würde hier Informationen darüber geben, warum die Subtraktion des Kleineren vom Größeren letztlich zum GGT führt, warum also die Schleife – nach Meinung des Programmautors – funktioniert.

Die GGT–Berechnung der Schleife beruht auf folgender mathematischen Erkenntnis

$$ggt(x,y) = \begin{cases} x & : \quad x = y \\ ggt(x-y,y) & : \quad x > y \\ ggt(x,y-x) & : \quad y > x \end{cases}$$

Etwas einfacher ausgedrückt: der GGT ändert sich nicht, wenn man von der größeren die kleinere Zahl abzieht. Damit haben wir schon das identifiziert, was sich nicht ändert: der GGT. In der Schleife ändert sich der GGT von `a` und `b` nicht, egal welche Werte `a` und `b` auch annehmen.

```
// a und b enthalten positive ganze Zahlen a_0, b_0
while (a != b) {
  // (Sinnvoller Kommentar zu Programm-STATIK, die INVARIANTE:)
  // INV: ggt (a, b) == ggt (a_0, b_0)
  if (a > b) a = a-b;
  if (b > a) b = b-a;
}
// a == b, ggt(a, b) == ggt(a_0, b_0) => a == ggt(a_0, b_0)
```

Beispiel: Ganzzahlige Division

Die Invariante hängt eng mit dem zugrunde liegenden Algorithmus zusammen, sie bringt "die Idee" der Schleife zum Ausdruck und sollte darum schon beim Entwurf der Schleife bedacht werden.

Betrachten wir ein Programm zur ganzzahligen Division mit Rest. Ein Beispiel für eine solche Division ist:

$$15 : 4 \; = \; 3 \, Rest \, 3$$

Ein einfacher Algorithmus ist: Beginne mit einem Rest von 15 (in der Variablen `rest`) und ziehe solange 4 vom Rest ab, bis dieser kleiner als 4 ist. Zähle dabei in einer Variablen `faktor` die Zahl der Subtraktionen. Als Wertverlaufstabelle:

rest	faktor
15	0
11	1
7	2
3	3

Warum ist dieser Algorithmus korrekt? Nun, ganz einfach: Man beginnt mit einem zu großen aber ansonsten korrektem Rest von 15 und erniedrigt ihn bis er kleiner als 4 ist. Gleichzeitig erhöht man den Faktor. Die Beziehung $Rest + Faktor * 4 = 15$ bleibt dabei erhalten.

$Rest$	+	$Faktor$ * 4	=	15
15	+	0 * 4	=	15
11	+	1 * 4	=	15
7	+	2 * 4	=	15
3	+	3 * 4	=	15

Als Schleife mit Invariante:

```
// Berechne die ganzzahlige Division
// a : b     ( a >= 0, b > 0 )
faktor = 0;
rest = a;
while (rest >= b) {  //INV a == faktor * b + rest
   rest = rest - b;
   faktor = faktor + 1;
}
// a == faktor * b + rest  UND  !( rest < b )
```

Die Schleifeninvariante gilt bei Eintritt in die Schleife und am Schleifenende. Am Schleifenende ist ausserdem die Schleifenbedingung nicht mehr erfüllt – die Schleife wäre ja sonst nicht verlassen worden. Beides zusammen ergibt das, was wir wollten: $a = Faktor * b + Rest \wedge Rest < b$.

4.10 Schrittweise Verfeinerung und Geschachtelte Schleifen

Geschachtelte Schleifen

Von geschachtelten Schleifen spricht man, wenn eine Schleife in einer Schleife auftritt. Ein Beispiel ist:

```
#include <iostream>

using namespace std;

int main () {
  int n;
  do {
    cout << "naechster Wert, Stop mit 0: "; cin >> n;
    if (n > 0) {
      int s = 0;
      for (int i = 1; i <= n; ++i)
        s = s + i;
      cout << "Summe i = (1 .. " << n << ") = " << s << endl;
    }
  } while ( n > 0);
}
```

Bei komplexeren Programmen – und geschachtelte Schleifen bringen immer eine gewisse Komplexität mit sich – ist es wichtig die Grobstruktur des Programms zu kennen. Das Beispiel beinhaltet eine äußere Do–While–Schleife und eine innere For–Schleife. Die äußere Schleife gehört zum Programmrahmen, mit dem Zahlen solange eingelesen werden, bis 0 oder eine negative Zahl auftaucht:

```
#include <iostream>

using namespace std;

int main () {
  int n;
  do {
    cout << "naechster Wert, Stop mit 0: "; cin >> n;
    ... Verarbeitung von n ...
  } while ( n > 0);
}
```

Die innere Schleife gehört zu dem Programmteil in dem die eingelesen Werte von n verarbeitet werden:

```
if (n > 0) {
  int s = 0;
  for (int i = 1; i <= n; ++i)
    s = s + i;
  cout << "Summe i = (1 .. " << n << ") = " << s << endl;
}
```

Hier werden einfach alle Zahlen von 1 bis n aufsummiert und die Summe dann ausgegeben.

Programme haben eine hierarchische Struktur. Sie bestehen aus ineinander geschachtelten Teilprogrammen, die jeweils eine bestimmte Aufgabe erfüllen und dazu andere untergordnete Teilprogramme in Anspruch nehmen. Diese Struktur zeigt sich nicht nur bei der Analyse eines Programms. Auch beim Enwurf eines neuen Programms empfiehlt es sich Teilprogamme zu identifizieren und ihnen jeweils einen Auftrag und andere Teilprogramme als Gehilfen zuzuweisen.

Beispiel: Alle Teiler bestimmen

Als nächstes soll ein Programm konstruiert werden, das positive ganze Zahlen einliest und anschließend alle ihre Teiler ausgibt.

Ein Muster für das Einlesen und Verarbeiten von Zahlen haben wir bereits mit dem letzten Beispiel kennengelernt:

```
#include <iostream>

using namespace std;

int main () {
  int n;
  do {
    cout << "naechster Wert, Stop mit 0: "; cin >> n;
    if (n > 0) {
      ... ? Teiler von n bestimmen und ausgeben ? ...
    }
  } while ( n > 0);
}
```

Dieser Programmrahmen kann als Ausgangspunkt für die weitere Entwicklung des Programms dienen.

Schrittweise Verfeinerung: Erster Verfeinerungsschritt

Komplexere Programme, beispielsweise solche mit geschachtelten Schleifen, kann man nicht ohne eine gewisse Systematik entwickeln. Es gibt verschiedene Techniken Programme systematisch zu entwickeln. Sie alle beruhen darauf, dass man das Gesamtproblem “Programm mit den geforderten Fähigkeiten konstruieren” in einfacher lösbare Teilprobleme zerlegt und aus den Teillösungen dann die Gesamtlösung zusammensetzt.

Die *schrittweise Verfeinerung* ist eine der ersten Programmentwicklungsmethoden. Nach ihr werden, ausgehend von einem Grobgerüst des Programms, systematisch immer weitere Teile des Programms immer feiner ausgearbeitet.

Für unser Beispiel der Teiler–Bestimmung existiert ein Grobgerüst. Der nächste Schritt ist die erste *Verfeinerung* dieses Grobgerüsts. Wir ersetzen dazu ein ungeschriebenes Programmstück mit klarer Spezifikation, im Beispiel:

```
... ? Teiler von n bestimmen und ausgeben ? ...
```

durch etwas detailierteren Programmcode, der aber auch wieder ungeschriebene Teilstücke enthalten kann.

Dazu müssen wir überlegen, wie man die Teiler einer Zahl n bestimmen kann. Möglicherweise gibt es eine intelligente Methode alle Teiler einer Zahl festzustellen. Wir kennen sie aber nicht und gehen darum mit *roher Gewalt* (*brute force*)[14] vor und untersuchen einfach alle in Frage kommenden Zahlen. Aus der Zeile

```
... ? Teiler von n bestimmen und ausgeben ? ...
```

wird damit das Programmstück:

[14] Werden ohne Rücksicht auf den Rechenaufwand einfach systematisch alle möglichen Lösungen untersucht, dann spricht man von einem “Algoritmus der rohen Gewalt” oder engl. einem *brute force algorithm*.

```
// n > 0
// Teiler von n bestimmen und ausgeben:
int c = 0;                        // Teiler zaehlen
for (int i = 2; i < n; ++i) {
  if (... ?  n wird von i ohne Rest geteilt ? ...) {
    cout << i << endl;
    ++c;
  }
}
if (c == 0)
  cout << n << " ist eine Primzahl" << endl;
else
  cout << "sind die Teiler von " << n << endl;
```

Hier werden einfach alle Zahlen kleiner `n` darauf hin untersucht, ob sie `n` ohne Rest teilen.

Man beachte, dass dieser Programmcode, als Verfeinerung einer Grobspezifikation, Bestandteil des Gesamtprogramms ist und dass er zwar detailierter als die Ausgangszeile ist, aber immer noch ungeschriebene Teilstücke enthält.

Zweiter Verfeinerungsschritt: Teilbarkeit feststellen

Im nächsten Verfeinerungsschritt muss wieder ein klar spezifiziertes, aber ungeschriebenes Programmstück ausgewählt und durch eine detailiertere Fassung ersetzt werden. Wir nehmen uns vor:

```
... ? n wird von i ohne Rest geteilt ? ...
```

zu verfeinern. Dazu wird ein Programmstück geschrieben, mit dem festgestellt wird, ob `i` ein Teiler von `n` ist. In Ermangelung tiefer zahlentheoretischer Kenntnisse gehen wir wieder mit roher Gewalt vor und ersetzen

```
if (... ? n wird von i ohne Rest geteilt ? ...)
```

durch folgende Verfeinerung:

```
//ist i ein Teiler von n:
int s = i;
while (s < n) // INV: s ist Vielfaches von i
  s = s+i;
// s ist Vielfaches von i UND s >= n
if (s == n)
  ... i ist Teiler von n ...
```

Wenn `i` ein Teiler von `n` ist, dann muss irgendein Vielfaches von `i` gleich `n` sein. Wir addieren also `i` einfach auf, bis wir genau auf `n` landen oder die Summe `n` überschreitet.

Gesamtprogramm: Teiler bestimmen

Die durch schrittweise Verfeinerung erstellten Programmteile werden jetzt einfach zusammengesetzt (Siehe Abbildung 16).

Damit ergibt sich dann folgendes Gesamtprogramm:

```
#include <iostream>

using namespace std;

int main () {
  int n;
  do {
    cout << "naechster Wert, Stop mit 0: "; cin >> n;
    if (n > 0) {

      //-Teiler von n bestimmen und ausgeben:
      int c = 0;
      for (int i = 2; i < n; ++i) {
```

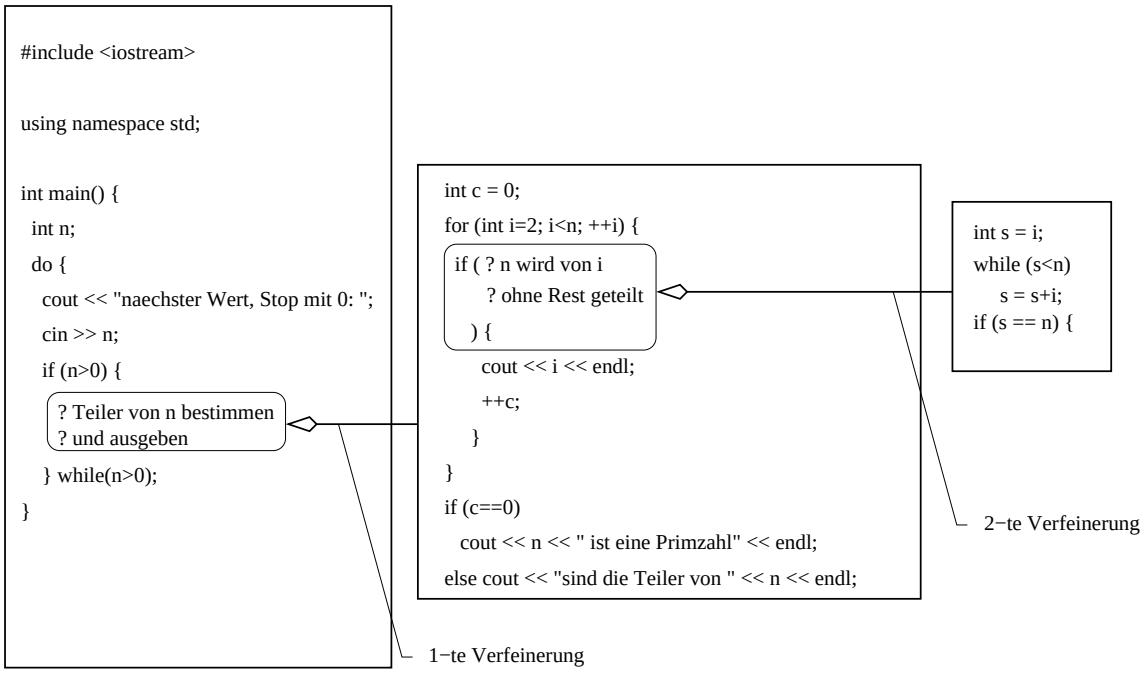


Abbildung 16: Schrittweise Verfeinerung

```
        //-Ist i ein Teiler von n
        int s = i;
        while (s < n)  // INV: s ist Vielfaches von i
          s = s+i;
        if (s == n) {
          cout << i << endl;
          ++c;
        }
      }
      if (c == 0)
        cout << n << " ist eine Primzahl" << endl;
      else
        cout << "sind die Teiler von " << n << endl;
    }
  } while ( n > 0);
}
```

Es ist jetzt sicher klar, dass Programme nicht ohne eine gewisse Systematik entwickelt werden können. Die erste Methode, die wir hier kennengelernt haben, ist die schrittweise Verfeinerung. Sie wird wie viele andere Methoden durch die Sprache C++ mehr oder weniger stark unterstützt. Die Unterstützung der schrittweisen Verfeinerung besteht in den übersichtlichen "schachtelbaren" Konstrukten, um größere Schleifen, Alternativen und zusammengesetzten Anweisungen aus anderen, kleineren Schleifen, Alternativen und zusammengesetzten Anweisungen zusammensetzen zu können.

4.11 Programmtest

Testfall

Bei einem Test wird experimentell festgestellt ob das Programm die gewünschte Ausgabe liefert. Dazu müssen Testfälle definiert werden. Ein *Testfall* besteht aus einer Programmeingabe und der daraufhin erwarteten Programmausgabe. Testfälle für das Programm aus dem letzten Abschnitt bestehen aus Folgen von Eingaben, die mit Null enden und den daraufhin erzeugten Listen von Teilern. Z.B. ("/" steht für einen Zeilenvorschub):

Testfall 1:
Eingabe: `5 6 0`
Ausgabe: `5 ist eine Primzahl / 2 / 3 sind die Teiler von 6`

Testfall 2:

Eingabe: `-3`
Ausgabe: Keine Ausgabe

Testfall 3:
Eingabe: `111`
Ausgabe: `3 / 37 / sind die Teiler von 111`

Die Testfälle sollten natürlich so umfangreich sein, dass man ein gewisses Vertrauen in das Programm gewinnt. Dabei wird die Eingabe "normaler", "extremer" und "falscher" Eingaben in Betracht gezogen. In unserem Beispiel wären das etwa Primzahlen und Zahlen die keine Primzahlen sind, kleine Werte und große Werte, die Null als erste Eingabe und die Eingabe von negativen Werten.

Fehlerstellen identifizieren

Im günstigen Fall besteht das Programm die Testfälle, das heißt zu jeder Eingabe wird die im voraus bestimmte Ausgabe erzeugt. Leider ist der günstige Fall nicht der Normalfall. Auch systematisch entwickelte Programme sind nicht unbedingt fehlerfrei. Es erleichtert allerdings das Testen und Auffinden von eventuellen Fehlern ganz erheblich, wenn man sich etwas bei seinem Quellcode gedacht hat und sich beim Testen auch noch daran erinnert – eventuell mit Hilfe von sinnvollen Kommentaren.

Liefert das Gesamtprogramm nicht das erwartete Ergebnis, dann müssen wir die Fehlerstelle identifizieren, also herausfinden welcher Teilschritt nicht wie geplant funktioniert. Es ist dabei sehr hilfreich, wenn das Programm in klare Teilschritte aufgegliedert wurde. Mit Hilfsausgaben oder `assert`–Anweisungen kann dann leicht geprüft werden, was korrekt funktioniert und was nicht. Als Beispiel statten wir das Programm zur Teiler–Berechnung mit entsprechenden Zwischenausgaben aus:

```
#include <iostream>

using namespace std;

int main () {
  int n;
  do {
    cout << "naechster Wert, Stop mit 0: "; cin >> n;
    if (n > 0) {

      //-Teiler von n bestimmen und ausgeben:
      int c = 0;
      for (int i = 2; i < n; ++i) {

        //-Ist i ein Teiler von n
        int s = i;
        while (s < n) { // INV: s ist Vielfaches von i
          cout << "TEST: " << s << " ist Vielfaches von " << i << endl;
          s = s+i;
        }
        if (s == n) {
          cout << i << endl;
          ++c;
        }
      }

      cout << "TEST: c = " << c << " = Zahl der Teiler von " << n << endl;

      if (c == 0)
        cout << n << " ist eine Primzahl" << endl;
      else
        cout << "sind die Teiler von " << n << endl;
    }
  } while ( n > 0);
}
```

Mit Testausgaben wird hier die Invariante der `while`–Schleife und ihr Berechungsergebnis ausgeben. Beim Programmtest kann dann geprüft werden, ob die ausgegeben Werte so sind wie erwartet. Wenn nicht, dann ist die Fehlerstelle eingegrenzt. Statt der Ausgaben kann man unter Umständen die `assert`–Anweisung einsetzen. Al-

lerdings sollten dann die zu prüfenden Sachververhalte etwas einfacher sein als hier, wo wir beispielsweise die Anzahl der Teiler einer Zahl prüfen müssten.

Debugger

Fehlerstellen im Programm werden auch *Bugs* (Wanzen, Schädlinge) genannt. Ihr Auffinden und Beseitigen nennt man *Degguging* (Entwanzen) oder auch eingedeutscht "Debuggen". Dabei hilft oft ein Hilfs–Programm namens *Debugger*. Ein Debugger kann natürlich keine Fehlerstellen finden oder gar beseitigen, das bleibt Aufgabe der Programmierer. Der Debugger erlaubt es, das Programm Schritt für Schritt durchzugehen und dabei die Werte aller Variablen anzusehen. Man kann sich damit die Hilfsausgaben ersparen. Statt dessen lässt man den Debugger das Programm an definierten Stellen unterbrechen und prüft ob die Variablen den an diesem Punkt erwarteten Wert haben.

Um ein Programm zu "debuggen" muss es so überstzt werden, dass das Maschinenprogramm Informationen über das Quellprogramm enthält. Beispielsweise, um einer Speicherstelle die Variable zuordnen zu können, deren Wert sie enthält, oder, um einem Maschinenbefehl die Anweisung im Quellprogramm zuordnen zu können, aus der sie der Compiler erzeugt hat. Dazu muss das Programm mit der "Debug–Option" übersetzt werden. Angenommen das Programm in der Quelldatei `hallo.cc` enthält einen Fehler und soll mit einem Debugger bearbeitet werden. Wir übersetzen mit der Option Debug–Option `-g`:

```
g++ -g -o hallo hallo.cc
```

und das erzeugte Maschinenprogramm `hallo` kann dann mit einem Debugger bearbeitet werden.

Debugger sind sehr nützliche Hilfsmittel der Programmentwicklung. Jeder Entwickler sollte sich mit ihrem Gebrauch vertraut machen. Sie ersetzen nicht die systematische Programmentwicklung, sie ergänzen sie.

4.12 Übungen

Aufgabe 1

Welchen Wert haben `a` und `b` nach dem Durchlaufen folgender Schleifen, wenn vorher gilt `a` = 3, `b` = 0.

1. `while (a>=b) if (b==0) a--;  // a--; entspricht a = a-1;`
2. `while (a>=b) { a=a+(b-1); b=b+a; }`
3. `while (a>=b) { a=a-b; if (b==0) b++; }`

Aufgabe 2

Manche Schleifen terminieren (enden) nie, andere enden nach einigen Durchläufen und wieder andere werden nicht ein einziges Mal durchlaufen. Die Zahl der Schleifendurchläufe hängt i.A. natürlich von der Belegung der Variablen zu Beginn der Schleife ab.

Bei welchen Variablenbelegungen terminieren folgende Schleifen, bei welchen nicht:

1. `while (a>=5) { a=a-1; do { a = a-1; } while (a>=5); }`
2. `do { a=a-1; while (a<=5) a = a-1; } while (a>=5);`

Aufgabe 3

Schreiben Sie ein Programm das so lange Zahlen einliest, bis zum ersten Mal eine Null eingelesen wird. Anschliessend soll die Summe und der Mittelwert aller eingelesenen Zahlen ausgegeben werden.

Aufgabe 4

Schreiben Sie ein Programm zur Berechnung von

$$\sum_{i=0}^{k} 2 \cdot i$$

Der Wert k soll eingelesen und die Summe in einer Schleife berechnet werden.

Aufgabe 5

Eine Folge von natürlichen Zahlen sei definiert als:

$a_0 = 0$
$a_i = 2 \cdot a_{i-1} + 1$

1. Geben Sie die ersten 5 Glieder der Folge an!
2. Definieren Sie rekursiv eine Funktion f auf natürlichen Zahlen für die gilt:

   $f(x) = a_x$

   (Die Funktion soll mathematisch, mit einem Stift auf Papier, definiert werden, nicht als Funktion in einem Programm.)
3. Schreiben Sie ein Programm das eine natürliche Zahl x einliest und den Wert $f(x)$ ausgibt.
4. Schreiben Sie ein Programm, das eine natürliche Zahl x einliest und den Wert

   $\sum_{i=0}^{x} f(i)$

   berechnet und ausgibt. Bevor Sie mit dem Programm beginnen, sollten Sie ein oder zwei Beispiele mit Stift und Papier berechnen!

Aufgabe 6

Schreiben Sie zwei Programme die π und e^x mit einer vorgegebenen Genauigkeit gemäß folgenden Formeln berechnen:

1. $\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots$
2. $e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$

Die Genauigkeit soll als Konstante im Programm definiert werden, der Wert von x für die zweite Formel wird eingelesen.

Aufgabe 7

Betrachten Sie folgendes Programm zur Berechung der Summe einer Folge:

```
...
int main () {
  int n,     // letzter Index
      c,     // Konstante
      s,     // berechnete Summe
      a_1;   // Anfang: 1-ter Summand

  cout << "a_1 = "; cin  >> a_1;
  cout << "n= ";    cin  >> n;
  cout << "c= ";    cin  >> c;

  int a = a_1;
  int i = 1;
  s = a_1;
  while (i != n) {
    a = a + c;
    s = s + a;
    i = i + 1;
  }

  cout << "s = " << s << endl;
}
```

Die Hilfsvariable `a` ist überflüssig. Statt zuerst `a` zu erhöhen um dann mit dessen neuem Wert `s` zu erhöhen, kann `s` sofort erhöht werden. Wie? Wie sieht die Schleife ohne `a` aus (`a_1` darf natürlich nicht einfach die Rolle von `a` übernehmen)?

Aufgabe 8

Die Behauptung

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

kann man mit vollständiger Induktion beweisen. Man kann sie auch experimentell mit Hilfe eines C++–Programms, das die Summation explizit durchführt, prüfen.

Tun Sie beides! Vergleichen Sie Induktionsverankerung und Schleifeninitialisierung sowie Induktionsschritt und Schleifenkörper.

Aufgabe 9

Betrachten Sie folgende Tabelle:

```
1
3   5
```

```
 7   9  11
13  15  17  19
21  23  25  27  29
...
```

Nach welchem Gesetz wird die Tabelle gebildet?

Schreiben Sie ein Programm, das die Tabelle mit einer eingelesen Zahl von Zeilen ausdruckt. Hinweis: Benutzen Sie geschachtelte Schleifen. Eine äussere Schleife für die Zeilen, eine innere für die Zeilenelemente.

Die Summe der Werte der i–ten Zeile folgt einer bestimmten Regelmässigkeit. Finden Sie diese heraus und überprüfen Sie Ihre Hypothese in ihrem Programm.

Aufgabe 10

Schreiben Sie ein Programm das eine positive Zahl x einliest und deren Quadradwurzel nach dem Verfahren von Heron berechnet:

$$\sqrt{x} = \lim_{n \to \infty} a_n$$

$$\begin{aligned} a_0 &= 1 \\ a_i &= \frac{a_{i-1} + \frac{x}{a_{i-1}}}{2} \end{aligned}$$

Aufgabe 11

Schreiben Sie ein Programm zur Berechnung der Funktion $f(x)$ für beliebiges reelles x:

$$f(x) = \lim_{n \to \infty} 2^n x_n$$

Die Folge der x_n ist dabei definiert als:

$$\begin{aligned} x_0 &= x \\ x_i &= \frac{x_{i-1}}{1 + \sqrt{1 + x_{i-1}^2}} \end{aligned}$$

Der Wert x ist einzulesen. Ihr Programm gibt dann $f(x)$ aus. Die Genauigkeit, mit der der Grenzwert berechnet wird, und die Genauigkeit der Wurzelberechnung sind Programmkonstanten.

Die Funktionen `sqrt` und `pow` *dürfen nicht* verwendet werden! Die Wurzel *muss* mit Hilfe des Verfahrens von Heron selbst berechnet werden. Ebenso *muss* die Potenzbildung selbst programmiert werden. Ausser `fabs` darf keine Funktion in Ihrem Programm vorkommen!

Benutzen Sie die Methode der schrittweisen Verfeinerung zur Konstruktion des Programms.

Welche (recht) bekannte Funktion wird hier berechnet?

Aufgabe 12

1. Schreiben Sie ein Programm, das nur mit Hilfe von Additions–, Subtraktions– und Vergleichsoperationen sowie einer While–Schleife testet, ob eine eingelesene Zahl gerade ist.
2. Schreiben Sie ein Programm, das nur mit Hilfe von Additions–, Subtraktions– und Vergleichsoperationen sowie einer while–Schleife feststellt, ob eine eingelesene positive ganze Zahl t ein Teiler einer anderen eingelesenen positiven ganzen Zahl n ist. Geben Sie die Schleifeninvariante an!

Aufgabe 13

Schreiben Sie ein Programm das die Werte x_1, x_2, p, q und r von der Standardeingabe einliest und in der Datei mit Namen `Dat.txt` die Werte x und $f(x) = px^2 + qx + r$ ausgibt, für 11 gleichmäßig im Bereich $[x_1, x_2]$ verteilte Werte von x. Beispielsweise soll bei der Eingabe `0 2 1 2 3` die Ausgabe

```
  0               3
0.2            3.44
0.4            3.96
0.6            4.56
0.8            5.24
  1               6
1.2            6.84
1.4            7.76
1.6            8.76
1.8            9.84
  2              11
```

erzeugt werden.

Aufgabe 14

1. Was versteht man unter einem Testfall? Was ist ein Programmfehler und was ist eine Fehlerstelle? Wie werden Fehlerstellen noch genannt? Welchem Zweck dient ein Debugger und was leistet er?

2. Definieren Sie Testfälle für das Programm von Aufgabe 12.2.

3. Machen Sie sich mit der Bedienung eines Debuggers vertraut.

4.13 Lösungshinweise

Aufgabe 1

Bitte Vermutungen (theoretisch) anstellen und dann mit Hilfe einfacher Programme (praktisch) prüfen! Tun Sie es wirklich, geben Sie entsprechende kleine Programme ein, man muss auch das Eintippen üben! Geben Sie also beispielsweise das folgende Prograämmchen ein:

```
#include <iostream>
using namespace std;
int main() {
  int a = 3, b = 0
  while (a>=b) if (b==0) a--;
  cout << "a="<<a<<" b="<<b<<endl;
}
```

und vergleichen sie dessen Ausgabe mit dem was Sie als Ausgabe erwarten!

Aufgabe 2

Bitte Vermutungen (theoretisch) anstellen und dann mit Hilfe einfacher Programme (praktisch) prüfen!

Aufgabe 3

Erstellen Sie einen ersten Entwurf des Programms: Welche Eingabe und welche Ausgabe hat das Programm, welche Variablen werden für welche Werte benötigt. Dieser Entwurf könnte wie folgt aussehen:

```
#include <iostream>
using namespace std;
int main () {
  float a,  // eingelesene Zahlen
        s,  // Summe
        m;  // Mittelwert
  int   n;  // Anzahl eingelesener Zahlen
            // fuer die Mittelwertbildung

  ?? Zahlen in a einlesen bis eine 0 eingegeben wird. ??
  ?? Mitz"ahlen wieviele Zahlen eingebene werden.     ??
  ?? (Fuer die Mittelwertbildung)                     ??

  cout << "Es wurden " << n << " Zahlen eingegeben (ohne 0) " << endl
       << "Summe:       " << s << endl
       << "Mittelwert:  " << m << endl;
}
```

Im Gegensatz zu einem Algorithmus “per Hand auf Papier” kann das Programm nicht in zwei Schritten arbeiten und zuerst alle Zahlen einlesen und dann alles aufadieren. Beides – Einlesen und Aufaddieren – muss miteinander verschränkt (vermischt) werden.

Versuchen Sie es bevor Sie weiterlesen!

Das Verschränken von Einlesen und Berechnen der Summe und des Mittelwerts führt uns zu einem etwas weiter entwickelten Programmentwurf:

```
#include <iostream>
using namespace std;
int main () {
  float a,  // eingelesene Zahl
        s,  // Summe bisher
        m;  // Mittelwert bisher
  int   n;  // Anzahl eingelesener Zahlen
            // fuer die Mittelwertbildung

  cout << "Bitte Eingabe (Stop mit 0): ";
```

```
  cin >> a;

  ?? s, m, n, initialisieren ??

  while (??noch keine Null eingelesen??) {
    ?? Wert in a ungleich 0, also:                    ??
    ?? Bei n, der Summe in s und beim Mittelwert in m ??
    ?? die zuletzt eingelesene Zahl bruecksichtigen   ??

    // naechste Zahl einlesen:
    cout << "Bitte Eingabe (Stop mit 0): ";
    cin >> a;
  }

  cout << "Es wurden " << n << " Zahlen eingegeben (ohne 0) " << endl
       << "Summe:       " << s << endl
       << "Mittelwert:  " << m << endl;
}
```

Mit diesem Ansatz sollten Sie zu einer Lösung kommen können. Versuchen Sie es bevor Sie weiterlesen.
Die Lösung ist:

```
#include <iostream>
using namespace std;
int main () {
  float a,  // eingelesene Zahl
        s,  // Summe bisher
        m;  // Mittelwert bisher
  int   n;  // Anzahl eingelesener Zahlen
            // fuer die Mittelwertbildung

  cout << "Bitte Eingabe (Stop mit 0): ";
  cin >> a;

  // Variablen initialisieren
  s = 0; m = 0 ;  n = 0;

  while (a != 0) {
    // a ungleich 0, also:
    // n, s, m an das neue a anpassen
    n = n+1;
    s = s + a;
    m = s/n;

    // naechste Zahl einlesen:
    cout << "Bitte Eingabe (Stop mit 0): ";
    cin >> a;
  }

  cout << "Es wurden " << n << " Zahlen eingegeben (ohne 0) " << endl
       << "Summe:       " << s << endl
       << "Mittelwert:  " << m << endl;
}
```

Aufgabe 4

Testen Sie als erstes Ihr Verständnis der Formel und berechnen Sie beispielsweise $\sum_{i=0}^{5} 2 * i$.

$\sum_{i=0}^{5} 2 * i = 2 * 0 + 2 * 1 + 2 * 2 + 2 * 3 + 2 * 4 + 2 * 4.$

Klären Sie die Fragen: Was soll das Programm einlesen, was soll es ausgeben!

Denken Sie bitte nach bevor Sie weiterlesen.

Es soll einen Wert für k einlesen und dann $\sum_{i=0}^{k} 2*i$ ausgeben. Also wenn Sie 5 eingeben, dann soll das Programm den Wert $\sum_{i=0}^{5} 2 * i$ ausgeben. In welchen Schritten kann man die Summe berechnen?

Denken Sie bitte nach bevor Sie weiterlesen.

Die gesuchte Summe ist das Ende der Folge der Teilsummen

$\sum_{i=0}^{0} 2 * i = 0,$
$\sum_{i=0}^{1} 2 * i = 0 + 2 * 1,$
$\sum_{i=0}^{2} 2 * i = 0 + 2 * 1 + 2 * 2,$
...
$\sum_{i=0}^{k} 2 * i.$

Wie kann aus einer Teilsumme die nächste berechnet werden?

Denken Sie bitte nach bevor Sie weiterlesen.

Aus der $(j-1)$–ten Teilsumme kann die j–te Teilsumme leicht berechnet werden:

$\sum_{i=0}^{j} 2 * i = (\sum_{i=0}^{j-1} 2 * i) + 2 * j$

Wenn die Teilsummen in einer Variablen `s` gespeichert werden, dann entspricht diese Berechnung den Zuweisungen:

```
j = j+1;
s = s + 2*j;
```

Damit haben wir schon die relevanten Variablen und den Schleifenköper gefunden:

```
#include <iostream>
using namespace std;
int main () {
  int   k,  // bis hierher summieren
        s,  // Teilsumme:
            // Summe aller 2*i fuer i=0 bis i=j
        j;  // bis hierher wurde summiert

  ?? k einlesen ??
  ?? Initialisierung von s, j ??
  while (?? ??) {
    // Inhalt von j erhoehen,
    // s entsprechend anpassen:
    j++;
    s = s + 2*j;
  }
  ?? s ausgeben ??
}
```

Der Rest – Initialisierung und Bedingung der Schleife – sollte keine Probleme bereiten.

Aufgabe 5

1. $a_0 = 0, a_1 = 1, a_2 = 3, a_3 = 7, a_4 = 15$
2. Definition von f:

   $f(0) = 0$
   $f(x) = 2 * f(x-1) + 1$

3. Programmentwurf:

```
#include <iostream>
using namespace std;
int main () {
  int   x, // eingelesene Zahl
i, // Nummer Folgeglied
a; // Folgeglied
```

```
  cin >> x;
  while ( x < 0 ) {
    cout << "Natuerliche Zahl eingeben!" << endl;
    cin >> x;
  }

  i = 0;
  a = 0;

  while (i < x) {
    ++i;
    ?? Neues Folgeglied aus altem Folgeglied (liegt in a) ??
    ?? berechnen und in a speichern                       ??
  }
  cout << a << endl;
}
```

4.
```
#include <iostream>
using namespace std;
int main () {
  int   x, i, a, s;

  cin >> x;
  while ( x < 0 ) {
    cout << "Natuerliche Zahl eingeben!" << endl;
    cin >> x;
  }
  i = 0; a = 0; s = 0;
  while (i < x) {
    ++i;
    a = 2*a + 1;
    s = s+a;
  }
  cout << s << endl;
}
```

Aufgabe 6

Die gesuchten Werte in beiden Teilaufgaben sind die letzten Glieder einer Folge von Teilsummen. Die Folge endet, wenn eine "vorgegebene Genauigkeit" erreicht ist. Die Genauigkeit wird als Wert einer Konstanten festgelegt. Sie ist erreicht, wenn zwei Teilsummen sich um weniger als diese Genauigkeit unterscheiden.

1. $\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots$
   Beim Übergang von einer Teilsumme zur nächsten muss

   - ein neuer Summand berechnet werden
   - dieser zur bisherigen Teilsumme mit gewechseltem Vorzeichen addiert werden

   dazu braucht man die Variablen:

   - `s`: Teilsumme bisher
   - `a`: aktueller Sumand
   - `v`: aktuelles Vorzeichen (z.B. als $+1$ bzw. -1)

   In Tabellenform:

`s`	`a`	`v`
1	1	$+1$
$1 - \frac{1}{3}$	$\frac{1}{3}$	-1
$1 - \frac{1}{3} + \frac{1}{5}$	$\frac{1}{5}$	$+1$
...	...	...

Der neue Wert von `a` kann mit einer weiteren Hilfsvariablen `q` berechnet werden. In ihr wird der aktuelle Quotient des Bruchs gehalten:

`s`	`q`	`a`	`v`
1	1	1	$+1$
$1-\frac{1}{3}$	3	$\frac{1}{3}$	-1
$1-\frac{1}{3}+\frac{1}{5}$	5	$\frac{1}{5}$	$+1$
...	...	...	...

Der Übergang von einer Zeile zur nächsten – der Schleifenkörper – ist jetzt ganz einfach:

```
??Initialisierung??
while (??Bedingung??) {
  q = q+2;
  v = v * (-1);
  a = (1/q) * v;
  s = s+a;
}
```

In der Schleifenbedingung muss geprüft werden, ob der neue Wert sich um mehr als die geforderte Genauigkeit von dem alten unterscheidet. Dazu könnte die Funktion `fabs(a)` nützlich sein, die den Absolutbetrag eines `float`–Wertes `a` berechnet und in der Mathematik–Bibliothek zu finden ist.

2. Ein Programm zur Berechnung von
   $e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$
   kann mit der Tabellenmethode ebenfalls leicht entwickelt werden.

Aufgabe 7

Eine äquivalente Schleife (mit mehr Rechenaufwand) ist:

```
...
int a = a_1;
int i = 1;
s = a_1;
while (i != n) {
  s = s + i*c + a_1;
  i = i + 1;
}
...
```

Aufgabe 8

Induktionsverankerung: $\sum_{i=0}^{0} 2^i = 2^0 = 1 = 2^{0+1} - 1$

Schleifeninitialisierung: `k = 0; s = 1;`

Induktionsschritt:

- Hypothese: $\sum_{i=0}^{k-1} 2^i = 2^k - 1$
- Schritt: $\sum_{i=0}^{k} 2^i = (\sum_{i=0}^{k-1} 2^i) + 2^k = 2^k - 1 + 2^k = 2 * (2^k) - 1 = 2^{k+1} - 1$

Schleifenkörper: `while (??) { k++; s = s +` 2^k`; }`

2^k kann leicht aus 2^{k-1} berechnet werden. Wenn die Wertefolge der 2^k in einer Variablen `p` festgehalten wird, dann wird der Schleifenkörper einfach zu:

```
  while (??) {
    k++;
    p = p*2;
    s = s+p;
  }
```

Die Schleifenbedingung ist offensichtlich. Am Ende der Schleife wird die Summe nach der Formel berechnet und der Wert von `s` und der Formelwert ausgegeben:

```
#include <iostream>
#include <cmath>
using namespace std;
int main () {
  int s, p, k, n;
  // n = ??   k = ??   s = ??  p = ??
  while (??) {
    k++;
    p = p*2;
    s = s+p;
  }
  cout << "Summe: " << s << " Formel: " << pow (2,n+1)-1 << endl;
}
```

Aufgabe 9

Das Programm besteht in seiner einfachsten Form aus zwei ineinander geschachtelten `for`–Schleifen. Eine äussere, die über die Zeilen und eine innere, die über die Spalten läuft.

```
  ??
  for (z=0; z<=t; z++) {
    ??
    for (s=1; s<=z; s++) {
      ??
    }
    ??
  }
```

`z` ist die aktuelle Zeilen– und `s` die aktuelle Spaltennummer. Der erste Wert einer Zeile kann leicht berechnet werden: $n_z = n_{z-1} + 2 * z$. Damit wird die Schleife zu:

```
  n_z = 1;
  for (z=0; z<=t; z++) {
    n_s = n_z;
    for (s=1; s<=z; s++) {
      ??
    }
    ??
    n_z = n_z + z*2;
  }
}
```

In der inneren Schleife werden die Spaltenelemente ausgegeben. Sie erhöhen sich einfach immer um 2:

```
  n_z = 1;
  for (z=0; z<=t; z++) {
    n_s = n_z;
    for (s=1; s<=z; s++) {
      cout.width(4);       // sorgt dafuer, dass alle Zahlen
      cout << n_s << " ";  // 4 Stellen einnehmen
      n_s = n_s + 2;
```

```
    }
    cout << endl;
    n_z = n_z + z*2;
  }
```

Jetzt muss nur noch `t` eingelesen werden sowie die Zeilensummen berechnet und ausgegeben werden. Beides sollte keine Probleme bereiten.

Die entsprechenden Werte geben dann einen Hinweis auf die mathematische Funktion, die hinter den Zeilensummen steckt – Sie sollten diese Funktion erkennen, sie ist einfach und wohlbekannt!

Aufgabe 10

Kein Lösungshinweis. Arbeiten Sie am Rechner, bis Sie eine Lösung erarbeitet haben!

Aufgabe 11

Kein Lösungshinweis. Arbeiten Sie am Rechner, bis Sie eine Lösung erarbeitet haben!

Aufgabe 12

1. Lösungsidee:

   - 0 ist gerade, 1 ist ungerade;
   - $n > 1$: n ist so gerade/ungerade wie n-2.

   Man kann darum von einer Zahl 2 abziehen ohne dass dies die Eigenschaft gerade oder ungerade zu sein verändert. Die Schleife kann darum darum eine Zahl verkleinern und dabei die Eigenschaft erhalten, die gleiche Geradheit/Ungeradheit wie eine andere zu haben.

   ```
   cin >> n;
   k = n;
   while (k>1) { // INV: n ist gerade/ungerade gdw.[15] k gerade/ungerade ist
     ... k geeignet erniedrigen ...
   }
   if (k == 0) {
      ... n ist gerade ...
   } else {
      ... n ist ungerade ...
   }
   ```

2. Lösungsidee:

   - t ist ein Teiler von $n \in \{0, ...t-1\}$, gdw. $n = 0$.
   - Für $n \geq t$ gilt: t ist ein Teiler von n, gdw. t ein Teiler von $n - t$ ist.

Aufgabe 13

```
#include <iostream>
#include <fstream>

using namespace std;

int main () {
  double x_1, x_2;
  double p, q, r;

  ofstream ausgabe("Dat.txt");

  cin >> x_1 >> x_2 >> p >> q >> r;
```

[15] gdw. = genau dann wenn

```
  if ( x_2 <= x_1) {
    cerr << "flasche Eingabe" << endl;
    return 1;
  }

  double intervall;
  intervall = (x_2 - x_1) / 10;

  for ( double x = x_1; x < x_2; x = x + intervall) {
    ausgabe.width(10);
    ausgabe.precision(4);
    ausgabe << x;
    ausgabe.width(17);
    ausgabe.precision(12);
    ausgabe << p*x*x + q*x + r << endl;
  }
}
```

Aufgabe 14

Kein Lösungshinweis.

5 Einfache Datentypen

5.1 Was ist ein Datentyp

Datentypen: Wertarten und Variablenarten

`int`, `char`, etc. sind *Datentypen* (kurz *Typen*, engl. *(Data) Types*). Sie beschreiben Werte, die möglichen Operationen auf den Werten und die Struktur der Variablen die diese Werte aufnehmen können. Man unterscheidet elementare (einfache, unstrukturierte) und nicht elementare (strukturierte) Datentypen. Die *elementaren Datentypen* sind:

- Typen für ganze Zahlen: `int`, `long`, `short`
  Ganze Zahlen werden je nach Maschine und Compiler in Speicherplätzen unterschiedlicher Länge abgespeichert. Z.B. `short` in 2 Bytes (16 Bits), `int` und `long` in 4 Bytes (32 Bits). Die Zahlen werden dabei in einer speziellen Bitcodierung abgelegt. Weder die Größe des Speicherplatzes noch die Art der Codierung wird von der Sprache festgelegt.

- Einzelne Zeichen: `char`
  Zeichen werden (fast immer) in einem Byte (8 Bits) abgespeichert. Die Codierung der Zeichen in Bits entspricht (fast immer) dem ASCII–Code.

- Wahrheitswerte: `bool`
  Wahrheitswerte werden meist in einem Byte abgelegt. Die Codierung entspricht der von Zahlen, wobei "falsch" (`false`) genau wie eine `int` 0 dargestellt wird. Jede andere Bit–Kombination bedeutet "Wahr". `true` wird wie eine 1 gespeichert.

- Typen für gebrochene Zahlen: `float`, `double`, `long double`
  Gebrochene Zahlen werden in 4 bis 12 Bytes abgelegt. Auch hier gilt, dass weder die Größe des Speicherplatzes noch die Art der Codierung von der Sprache festgelegt wird.

Die genaue Art der Zahldarstellung (Bitmuster, Größe) wird von den Möglichkeiten der Rechner–Hardware und dem verwendeten Compiler festgelegt. Die genannten Datentypen sind vordefiniert und elementar. Ein *vordefinierter Datentyp* wird sozusagen gleich mit der Sprache mitgeliefert. Neben den vordefinierten können in einem Programm auch eigene – selbstdefinierte – Datentypen eingeführt werden. Ein *einfacher Datentyp* hat im Gegensatz zu einem strukturierten Datentyp keine innere Struktur. Der Datentyp `string` etwa ist nicht einfach, da man eine Zeichenkette in ihre Elemente zerlegen kann.

Elementare Datentypen im Rechner: Hardware bestimmt das Format von Bitsequenzen

Im Rechner gibt es nur Bits. Jeder Wert wird durch eine Bitsequenz dargestellt. Ob eine bestimmte Bitsequenz eine Zahl, ein Zeichen, ein Bild oder einen Ton darstellt, ist reine Interpretationssache. Die Interpretation ist allerdings nicht völlig willkürlich. Elementare Operationen, Addition, Subtraktion, etc., werden von Hardware–Komponenten ausgeführt. Diese verarbeiten Bitsequenzen mit einer festgelegten Länge und natürlich auch festgelegter Interpretation der Bits. Die Addition ganzer Zahlen etwa wird von einer Komponente ausgeführt, die zwei Bitsequenzen festgelegter Länge verarbeitet und dabei natürlich auch die Bits in "fest verdrahteter" Art interpretiert. Es ist darum sehr sinnvoll `int`–Zahlen in Größe und Codierung so auszurichten, dass sie von der Hardware direkt verarbeitet werden können. Entsprechendes gilt für Gleitkommazahlen und Zeichen (die ein– und ausgegeben werden können).

Die interne Darstellung eines Wertes hängt darum ab vom Wert, seinem Datentyp und der Maschine, auf der das Programm läuft. Beispielsweise kann der Wert 6 folgendermaßen als Bitsequenz im Speicher eines PCs abgelegt sein:

`char c = 6` wird in einem Byte gespeichert als `00110110`

`int i = 6` wird in 4 Bytes gespeichert als
`00000110 00000000 00000000 00000000`

`double f = 6` wird in 8 Bytes gespeichert als:
`00000000 00000000 00000000 00000000 00000000 00000000 00011000 01000000`

Das sind natürlich systemabhängige Beispiele. Auf einem anderen System kann es auch etwas anders aussehen.

Byteordnung: Bits im Speicher und in der CPU

Die Bitsequenz zur Darstellung von `int i = 6` wird eventuell etwas verwundern. 6 als 32–stellige Binärzahl ist:

```
00000000000000000000000000000110
```

In dieser Form taucht die 6 auch tatsächlich in einem der Register der CPU auf, wenn der Wert verarbeitet werden soll. Die Daten werden aber nicht unbedingt in der Form im Speicher abgelegt, in der sie in der CPU verarbeitet werden. Die 32–stellige Binärzahl, die zu einem Int–Wert gehört, belegt (in einem 32–Bit–Rechner) genau ein Register. Im Speicher werden alle Werte als Bytes von jeweils 8 Bits gespeichert. Wenn ein Wert verarbeitet werden soll, dann muss er – Byte für Byte – vom Speicher in die CPU geladen werden. Die Art dieses Ladens wird durch die *Byteordnung* bestimmt und ist nicht bei allen CPUs gleich. Bei Intel–CPUs wird ein Register "von rechts nach links" mit den Bytes aus dem Speicher gefüllt. Das erste Byte im Speicher enthält darum die 8 Bits des Wertes mit der geringsten Wertigkeit. Man sagt Intel "hat die Byteordnung *Least significant Byte First"* (Siehe Abbildung 17).

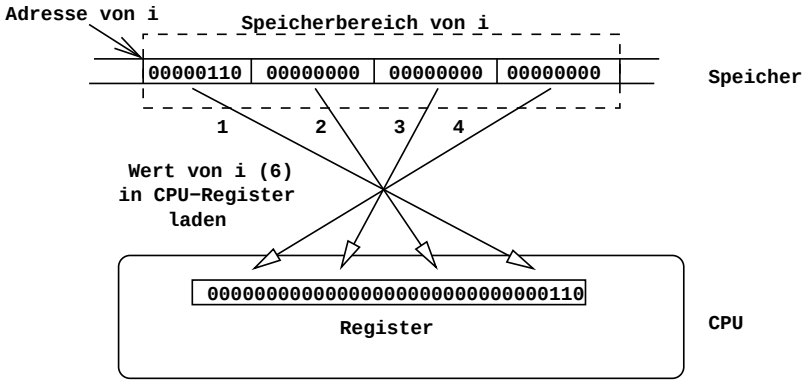


Abbildung 17: Byteordnung *Least Significant Byte First*

Die CPUs anderer Hersteller haben die Byteordnung *Most Significant Byte first*. Das heißt, dass das erste Byte im Speicher (das mit der niedrigsten Adresse) die Bits mit der höchsten Wertigkeit enthält. Wird ein Wert aus dem Speicher in ein CPU–Register geladen, dann wird das Register von links nach rechts gefüllt.

Datentypen ohne Hardwareunterstützung: Software bestimmt das Format von Bitsequenzen

Datentypen, die nicht direkt von der Hardware unterstützt werden, benötigen spezielle Software die ihre Operationen ausführt. Die notwendige Software kann einfach sein. Ein `short`–Wert beispielsweise ist meist kürzer als die von der Rechnerarithmetik verarbeitete Bitsequenz. In dem Fall muss per Software vor einer Addition der Wert mit führenden Nullen aufgefüllt und so auf die richtige Länge gebracht werden. Oft sind die Aktionen die nötig sind, um die Operationen eines Datentyps auf die Möglichkeiten der Hardware abzubilden, aber wesentlich komplexer. Man denke etwa an die +–Operation in

```
string s = "Hallo " + " wer " + "da?"
```

Die Hardware eines Rechners hat also auch ein einfaches Typkonzept. Es definiert das Speicherformat der elementaren Datentypen (`int`, `char` und `float` oder `double`) und bildet damit die Basis des gesamten Typkonzepts der Sprache, das aber insgesamt weit über diese elementaren Grundlagen hinausgeht. Die Sprache definiert das Typkonzept und die Compiler bilden es auf die Möglichkeiten der Hardware ab.

Datentypen: Informationen und Anweisungen an den Compiler

Typinformationen in einem Programm sind Informationen die der Compiler während der Übersetzung überprüft und beachtet. Ein Beispiel ist:

```
int    i;         // schaffe Platz fuer einen int--Wert
float  f, g;      // schaffe Platz fuer 2 float Werte
double d = 2.0;   // schaffe Platz fuer einen double-Wert,
                  // erzeuge Bitmuster fuer double-2
i = j;            // kopiere die richtige Zahl an Bits
i = i+i;          // addiere mit der Ganzzahl--Hardware
```

```
f = f+g;          // addiere mit der Gleitkomma--Hardware
f = i;            // konvertiere von der Ganzzahldarstellung in die
                  // Gleitkomma-Darstellung;
f = d.length();   // pass auf, dass so etwas nicht passiert
```

Eine Variablendefinition wie

```
int i;
```

wird in Maschinencode übersetzt, der der Variablen Platz im Speicher verschafft. Dazu muss der Platzbedarf bekannt sein. In

```
double d = 2.0;
```

wird der Variablen `d` der Fließkomma–Wert 2 zugewiesen. Dazu muss bekannt sein, welches Bitmuster einer Fließkomma 2 mit der `double`–Länge zugeordnet ist. Bei einer Zuweisung einer Variablen an eine andere muss der Compiler wissen wieviele Bytes zu bewegen sind und den entsprechenden Maschinencode erzeugen. Bei einer Addition muss er aus dem Typ der Argumente schließen welche Additions–Hardware zu verwenden ist und einen entsprechenden Maschinenbefehl absetzen. Bei einer Zuweisung von Variablen unterschiedlichen Typs, z.B. in

```
f = i;
```

wird geprüft, ob das überhaupt erlaubt ist und wenn ja – wie hier – wird *konvertiert*: Der Compiler erzeugt Maschinenbefehle die später zur Laufzeit des Programms die Bitsequenz in `i` – die einen `int`–Wert darstellt – in eine Bitsequenz transformiert, die den entsprechenden `float`–Wert darstellt. Bei

```
d.length()
```

muss der Compiler schließlich eine Fehlermeldung ausgeben, da `d` kein `string` ist und darum keine `length`–Methode hat.

Insgesamt ist der Compiler dafür zuständig, dass die abstrakten Konzepte des Programms in die Möglichkeiten des Rechners umgesetzt werden. Dabei beachtet er:

- Die Größe von Bitmustern im Speicher (wieviele Bits nimmt ein Wert in Anspruch).
- Die Bedeutung von Bitmustern (Zahl, Zeichen, welche Codierung etc.).
- Die Regeln zur Verarbeitung der Bitmuster (wie wird addiert, wie die Länge bestimmt, ...).
- Welche Operationen erlaubt sind, welche nicht (Programmfehler aufdecken).

5.2 Datentypen im Programm

Werte, Literale, Operationen

Im Programm ist ein Datentyp:

- eine Menge von *Werten*,
- eine Menge von *Literalen* (auch *literale Konstanten* genannt) zur Bezeichnung von Werten,
- eine Menge von möglichen *Operationen* auf den Werten,
- eine Eigenschaft von Variablen; z.B. `float`–Variablen können nur `float`–Werte aufnehmen.

Ein *Literal* ist so etwas wie `1.5`, `true` oder `'a'`: Ein Stückchen Programmtext das in jedem Programm einen ganz bestimmten festen Wert bezeichnet. *Namen* wird dagegen im Programm eine Bedeutung zugewiesen. Ob `a12` in einem Programm etwas bedeutet, und wenn ja was, das wird vom Programm bestimmt. Ein Name wie `a12` kann mal dies, mal das und mal gar nichts bedeuten. Das Literal `1.5` bedeutet dagegen immer und in jedem Programm das gleiche.

Werte, Bitmuster und Literale sind zu unterscheiden. Werte sind etwas Abstraktes, das durch Bitsequenzen mit einer bestimmten Interpretation dargestellt wird. Bitmuster werden von Hard– oder Software verarbeitet, dabei ist es letztlich unerheblich welchen Wert sie darstellen sollen. Bitmuster und damit die durch sie dargestellten Werte existieren zur Laufzeit in der Hardware der Maschine.

Literale sind Bestandteile des Quellprogramms: kleine Textstücke mit einer festen und in allen Programmen unveränderlichen Bedeutung. Sie sind nur Text. Zur Laufzeit, in der Maschine, existieren sie nicht.

Der mathematische Wert 10 wird sowohl durch das `int`–Literal `10` als auch durch das `float`–Literal `10.0` dargestellt. Im Speicher werden die Werte aber als völlig unterschiedliche Bitmuster erscheinen.

Wozu Datentypen

Datentypen dienen:

- Der Effizienz der Programme:
  Die Hardware des Rechners verarbeitet Bitsequenzen mit einigen Operationen direkt und damit schnell. Typischerweise werden Ganzzahl– und Fließkommazahl–Operationen unterstützt. Die Werte müssen dazu in der richtigen Form (Codierung als Bitsequenzen) vorliegen. Das Speicherformat von Werten wird durch ihren (Daten–) Typ festgelegt.
- Der Entlastung des Programmautors:
  - Der Compiler bestimmt Bitmuster,
    Z.B.: `1, "1"` und `1.0` werden korrekt in die völlig unterschiedlichen Bitmuster einer `int`–Zahl, einer Zeichenkette, einer `float`–Zahl umgesetzt
  - Der Compiler wählt richtige Maschinen–Anweisungen
    Z.B.: bei `i = 2*i;` wird die richtige Zahl von Bits vom Speicherplatz der Variablen `i` in ein Register geladen; das richtige Bitmuster für die Zahl 2 in `int`–Form mit der richtigen (`int`–) Additions–Hardware addiert. etc.
- Der Typsicherheit, d.h. dem Schutz des Programmierers
  Eigentlich ist alles ein Bitmuster, da kann es leicht zu Verwechslungen kommen. Programmfehler werden vom Compiler als Typverletzungen entdeckt, z.B. (`int`–Addition auf `float`–Werte etc.)

Werte unterschiedlichen Typs mit passenden Operationen sind *virtuell*, es gibt so etwas nicht "wirklich". Sie sind ein Angebot der *virtuellen Maschine* "C++". *Real* im Rechner gibt es nur Bitmuster als einzigen (wirklichen) Typ.

Die Weiterentwicklung der Programmiersprachen besteht zum Großteil auf einer Verfeinerung des Typkonzepts. C++ unterscheidet sich auch von C vor allem durch sein weiter entwickeltes Typkonzept.

5.3 Integrale Datentypen

Integrale Datentypen

Die *integralen Datentypen* sind `int` und seine engere und weitere Verwandtschaft: `short`, `int`, `long`, `char` und `bool`. `int`, `long`, `short` und `char` gibt es dazu noch in den Alternativen `signed` und `unsigned`. Die integralen Datentypen insgesamt sind also:

- `short` (= `signed short`), `unsigned short`
- `int` (= `signed int`), `unsigned int`
- `long` (= `signed long`), `unsigned long`
- `char` (= `signed char`), `unsigned char`
- `bool`

Mit oder ohne Vorzeichen: `signed` und `unsigned`

Bei allen integralen Typen außer `bool` kann das äußerst linke Bit entweder als Bestandteil des Wertes (`unsigned` ...) oder als Vorzeichen (`signed` ...) interpretiert werden. Beispielsweise erzeugt das Programm

```
#include <iostream>
using namespace std;
int main () {
  short           x = -1; // == signed short x = -1;
  unsigned  short y = x;
  cout << "x: " << x << endl;
  cout << "y: " << y << endl;
}
```

die Ausgabe:

```
x: -1
y: 65535
```

Die Bitfolge `1111 1111 1111 1111` (zwei Bytes voll mit Einsen) bedeutet als `signed short` die Zahl -1. (Die Zahlen werden im 2er Komplement dargestellt, die erste Eins zeigt negative Werte an.) Als `unsigned short` bedeutet die gleiche Bitfolge 65535.

`int`–Literale

Die Literale der integralen Typen gibt es in drei Varianten:

- `int`– Literale: z.B.: `-1`, `1`, `2`, ...
- `char`– Literale: z.B.: `'a'`, `'b'`, ... `'1'`, ... `'!'`, ...
- `bool`– Literale: `true`, `false`

`int`–Literale sind ganze Zahlen in der üblichen Dezimalnotation. Oktale und hexadezimale Notation ist ebenfalls möglich. Durch eine vorangestellte "`0`" wird das Literal als Oktalzahl und durch vorangestelltes "`0x`" als Hexadezimalzahl interpretiert. So bezeichnen die drei Literale:

```
26
032
0x1a
```

jeweils den gleichen Wert 26. Alle `int`–Literale repräsentieren Werte vom Typ `int`. Durch Anhängen eines `L` (oder `l`) und/oder eines `U` (oder `u`) kann man explizit Werte vom Typ `long` und/oder `unsigned` bezeichnen:

```
26U      // 26 vom Typ unsinged int
0x1aL    // 26 vom Typ long
032UL    // 26 vom Typ unsigned long
```

In der Regel ist es nicht notwendig `U` und `L` zu verwenden. Die Werte werden automatisch in das richtige Format konvertiert:

```
unsigned long x = 26; // automatische Konversion 26 -> 26UL
```

`char`–Literale

Mit `char`–Literalen können druckbare und nicht–druckbare Zeichen angegeben werden. Bei druckbaren Zeichen wird das Zeichen in Hochkommas gesetzt, Beispiele sind:

`'a'`, `'X'`, `'+'`, `' '` (Leerzeichen), `'1'`, `'-'` ...

Der ASCII–Standard definiert (üblicherweise) die Bitkombination die mit einem Zeichen verbunden ist. Beispielsweise erzeugt das Programm

```
#include <iostream>
using namespace std;
int main () {
  char  c1 = 'A',
        c2 = '0';
  int   i1 = c1;  // OK
  int   i2 = c2;
  cout << "c1: "   << c1
       << "  i1: " << i1 << endl;
  cout << "c2: "   << c2
       << "  i2: " << i2 << endl;
}
```

die Ausgabe

```
c1: A  i1: 65
c2: 0  i2: 48
```

Dem Zeichen `'A'` ist im ASCII–Code die gleiche Bitkombination zugeordnet wie der `int`–Zahl `65` und dem Zeichen '0' ist die gleiche Bitkombination zugeordnet wie der ganzen Zahl `48`. Diese Codierung wird vom ASCII–Code definiert und ist eine sehr alte Konvention zur Speicherung von Zeichen:

Bitsequenz	interpretiert als char	interpretiert als int
0011 0000	'0'	48
0011 0001	'1'	49
0011 0010	'2'	50
...	...	...
0100 0001	'A'	65
0100 0010	'B'	66
0100 0011	'C'	67
...	...	...

Der ASCII–Code definiert genaugenommen Bitkombinantionen der Länge 7. Diese werden stets in einem Byte (8 Bits) mit einer führenden Null abgespeichert. `'A'` hat den "ASCII–Code" `65` oder binär `100 0001`. Beide werden in einem Byte als `0100 0001` abgespeichert.

Interessanterweise definiert C++ auch einige Literale für *nicht-druckbare* Zeichen:

- `'\n'` Zeilenvorschub
- `'\t'` horizontaler Tabulator
- `'\v'` vertikaler Tabulator
- `'\b'` Zeichenrücklauf (*Backspace*)
- `'\r'` Zeilenrücklauf (*carriage return*)
- `'\f'` Seitenvorschub
- `'\a'` Alarm

In oktaler Notation kann auch direkt die Codierung eines Zeichens angegeben werden. Beispielsweise sind

```
char c = '\60';
char c = '0'; und
char c = 48;
```

völlig äquivalent. Zeichen die eine Sonderfunktion bei Literalen erfüllen wie \, `'` oder `"` können als *Escape Sequenz* – also mit vorgestelltem \ – angegeben werden. Z.B.:

- `'\\'`
- `'\''`

`bool`–Literale

Im Vergleich zu den `int` und `char`–Literalen sind die `bool`–Literale geradezu trivial. Es gibt nämlich nur zwei:

- `true`
- `false`

`false` wird dabei wie eine `int`–0 und `true` wie eine `int`–1 abgespeichert. Umgekehrt bedeutet jede Bitsequenz, die nicht als `int`–0 interpretiert wird auch "wahr".

Operationen auf integralen Datentypen

Auf Werten des Datentyps `int` und seinen Verwandten sind die üblichen arithmetischen Operationen definiert. Sie werden immer mehr oder weniger direkt von der Hardware des Rechners unterstützt. D.h. wenn `i` einen integralen Datentyp hat, dann wird eine Addition wie in `i = i + i;` in einen Maschinenbefehl übersetzt und von der Hardware ausgeführt. Die Operationen sind:

- `+ , -, *` Addition, Subtraktion, Multiplikation
- `/` ganzzahlige Division

- `%` Modulo–Operation

Addition, Subtraktion und Multiplikation folgen den üblichen Gesetzen. Allerdings kann es zu einem *Überlauf* kommen. Wenn die Größe des Speicherplatzes nicht ausreicht, um das Ergebnis einer Operation aufzunehmen, dann werden die anfallenden überzähligen Bits einfach ignoriert bzw. überschreiben das Vorzeichen-Bit. Beispielsweise erzeugt das Programm

```
#include <iostream.h>
using namespace std;
int main () {
  short k=2, i=1, j=0; // k > i > j

  while (i > 0) { ++i; ++j; ++k; }

  cout << "j (= i-1) =  " << j << endl;
  cout << "i         =  " << i << endl;
  cout << "k (= i+1) =  " << k << endl;
}
```

ohne Warnung die Ausgabe:

```
j (= i-1) =  32767
i         =  -32768
k (= i+1) =  -32767
```

Bei der ganzzahligen Division `/` wird das Ergebnis auf die nächste ganze Zahl abgeschnitten. Z.B.:

`5/3` $= 1$
`(-5)/3` $= -1$
`5/(-3)` $= -1$.

Die Modulo–Operation berechnet den Rest bei ganzzahliger Division, z.B. `7 % 3` = 1; Die Modulo–Operation ist (in C++) nur für positive Argumente definiert. Das heißt nicht dass z.B. `7 % (-3)` kein Ergebnis hat, sondern dass das Ergebnis Maschinen– und/oder Compiler–abhängig ist und nicht von der Sprache definiert wird. Programme in denen `%` auf negative Operanden angewendet wird, sind darum nicht portabel.

Werte integraler Datentypen

Die integralen Datentypen sind aufs Engste miteinander verwandt. Ihnen allen ist gemeinsam, dass sie intern wie ganze Zahlen in einer einheitlichen Darstellung behandelt werden. Man kann also nicht nur Werte eines `int`–Typs sondern auch `char`– und `bool`–Werte mit arithmetischen Operationen bearbeiten. Das Programm

```
#include <iostream>
using namespace std;
int main () {
  int i = 32;
  for (char c = 32; c < 127; ++c, ++i)
    cout << i << ": " << c << endl;
}
```

gibt beispielsweise alle druckbaren ASCII–Zeichen mit ihrem Integer–Code aus.

Der ASCII–Code ist ein 7–Bit Code. Die Obergrenze ist darum 127. Das Zeichen mit dem Code 127 ist aber genau wie alle mit einem Code kleiner 32 nicht druckbar. Man kann natürlich auch nicht–druckbare Zeichen ausgeben, also solche denen nach ASCII kein druckbares Zeichen zugeordnet ist. Das Resultat ist dann maschinenabhängig:

```
#include <iostream>
using namespace std;
int main () {
  int i = 32;
  for (unsigned char c = 32; c < 128; ++c, ++i)
    cout << i << ": " << c << endl;
}
```

Hier ist die Laufvariable `c` vom Typ `unsigned char`: 128 liegt außerhalb des (7-Bit) Bereichs von `char` (-128 .. + 127). Ein `char` kann darum nicht sinnvoll mit 128 verglichen werden.

Der Wertebereich der verschiedenen int–Varianten wird in der Definition von C++ offen gelassen. Die einzige Forderung ist, dass jeder char– auch ein short–, jeder short– auch ein int–Wert und jeder int– auch ein long–Wert sein muss. Die tatsächlichen Grenzen der Wertebereiche können mit folgendem Programm festgestellt werden:

```
#include <iostream>
#include <climits>

using namespace std;

int main () {
  cout << "Wertebereich der integralen Typen" << endl;
  cout << "char, unsigned char, short, unsigned short,";
  cout << "int, unsigned int, long und unsigned long" << endl;
  cout << endl;
  cout << "CHAR  : " << CHAR_MIN  << "\t\t...\t" << CHAR_MAX  << endl;
  cout << "UCHAR : " << "0"       << "\t\t...\t" << UCHAR_MAX << endl;
  cout << "SHORT : " << SHRT_MIN  << "\t\t...\t" << SHRT_MAX  << endl;
  cout << "USHORT: " << "0"       << "\t\t...\t" << USHRT_MAX << endl;
  cout << "INT   : " << INT_MIN   << "\t...\t"   << INT_MAX   << endl;
  cout << "UINT  : " << "0"       << "\t\t...\t" << UINT_MAX  << endl;
  cout << "LONG  : " << LONG_MIN  << "\t...\t"   << LONG_MAX  << endl;
  cout << "ULONG : " << "0"       << "\t\t...\t" << ULONG_MAX << endl;
}
```

Eine mögliche Ausgabe dieses Programms ist:

```
Wertebereich der integralen Typen
char, unsigned char, int, unsigned intlong und unsigned long

CHAR  : -128            ...     127
UCHAR : 0               ...     255
SHORT : -32768          ...     32767
USHORT: 0               ...     65535
INT   : -2147483648     ...     2147483647
UINT  : 0               ...     4294967295
LONG  : -2147483648     ...     2147483647
ULONG : 0               ...     4294967295
```

5.4 Bitoperatoren und Bitvektoren

Ein Bitvektor ist eine Folge von 0 und 1, die als binäre Daten direkt abgespeichert werden. Bitvektoren erlauben eine kompakte Speicherung von binären Daten wie z.B. Zustandsinformationen (an–aus, etc.).

C++ hat zwei Datentypen um binäre Daten in kompakter Form als Bitvektoren zu speichern: Boolesche Vektoren (Typ vector<bool>) mit unbestimmter und veränderlicher Länge, sowie "Bitsets" als Bitvektoren fester Länge (z.B. bitset<8>). Beide stellen zusammengesetzte Datentypen dar, wir werden sie darum an anderer Stelle behandeln.

Integrale Typen als Bitvektoren

Zum C–Erbe von C++ gehört die Fähigkeit integrale Datentypen als *Bitvektoren* interpretieren zu können. Diese Methode ist recht schwierig zu handhaben und wenig portabel.[16] Sie bietet aber die Möglichkeit "direkt auf die Bits" einer Variablen zu sehen.

Als Beispiel betrachten wir folgendes Programm, das einen char–Wert einliest und dann seine Bitrepräsentation ausgibt:

```
#include <iostream>

using namespace std;
```

[16] Man bezeichnet ein Programm als portabel, wenn es ohne Modifikationen von einem Rechner auf einen beliebigen anderen gebracht werden kann und dort das gleiche Verhalten zeigt.

```
int main () {
  char          c;
  unsigned char shifter;

  cin >> c;
  cout << "interne Darstellung von " << c << " = ";

  for (shifter = 0x80;            // Initialmuster: 10000000
       shifter > 0;               // 1 noch nicht rausgeschoben
       shifter = shifter>>1) {    // Shift: 1 um eine Position verschieben

    if ( c & shifter ) cout << "1";
    else               cout << "0";
  }

  cout << endl;
}
```

Die Variable `shifter` enthält am Beginn der Schleife das Bitmuster `10000000`. Nach jedem Schleifendurchlauf wird mit:

```
shifter = shifter>>1
```

die `1` um eine Position nach rechts verschoben. Der Shift–Operator `>>` verschiebt bitweise um die angegebene Zahl von Stellen nach rechts. Dabei werden von hinten Nullen nachgezogen. Mit dem bitweisen Und (`&`) in

```
c & shifter
```

wird getestet, ob `c` an der Position, an der in `shifter` die `1` gerade steht, ebenfalls eine `1` enthält. In der Schleife wird also jede Bitposition von `c` abgeprüft.

Bitoperationen

Die Bitoperationen sind (Bindungsstärke abnehmend):

- `~` Bitweises NICHT, jedes Bit wird umgekehrt.
- `>>` Shift nach rechts, Nullen werden nachgezogen.
- `<<` Shift nach links, Nullen werden nachgezogen.
- `&` Bitweises UND.
- `|` Bitweises ODER.
- `^` Bitweises XOR.

Bei den bitweisen Operationen wird jedes Bit der Operanden positionsweise verknüpft. Z.B.:

```
1100001 & 0000001 = 0000001
1100001 | 0000001 = 1100001
1100001 ^ 0000001 = 1100000
```

Die bitweisen Operationen können auf alle integralen Datentypen angewendet werden. `~`, `&` und `|` sollten nicht mit den logischen Operatoren `!`, `&&` und `||` verwechselt werden.

Bitoperationen und Byteordnung

Die Bitoperationen beachten die Byteordnung der Maschine, auf der sie operieren. Auf einer Intel–Maschine wird beispielsweise zuerst das niederwertige und dann das höherwertige Byte abgespeichert. Eine `short int 3`, abgelegt in zwei Bytes, ist darum tatsächlich als

```
00000011 00000000
```

im Speicher zu finden. Die Bit–Operationen werden aber in Registern ausgeführt und dort liegen die Werte in der richtigen Byteordnung. `00000011 00000000` wird also auf einem Intel Rechner als `0000000000000011` aus dem Speicher in ein Register geladen und umgekehrt wieder abgespeichert.

Die Bitoperationen behandeln den Wert also in einem Register der CPU in dem die 3, mit vertauschten Bytes, auftaucht als:

```
0000000000000011.
```

D.h. `3 >> 1` ergibt wie erwartet 1, auch wenn die 3 als `00000011 00000000` im Speicher steht.

5.5 Der Datentyp Float

Die verschiedenen Gleitpunkt–Typen

Die Gleitpunkttypen sind

- `float`
- `double`
- `long double`

Im Gegensatz zu den integralen Typen unterscheiden sich die Float–Typen nicht nur in der Größe des Wertebereichs, sondern auch in der *Genauigkeit* der Darstellung. Die exakten Werte des jeweiligen Rechners kann man sich mit folgendem Programm ausgeben lassen:

```
#include <iostream>
#include <cfloat>

using namespace std;

int main () {
  cout << "Wertebereich der Float Typen" << endl << endl;
  cout << "Float" << endl;
  cout << "  Genaugkeit    : " << FLT_DIG << " Dezimalstellen" << endl;
  cout << "  Bereich       : " << FLT_MIN << "..." << FLT_MAX << endl;
  cout << "  Schritt       : " << FLT_EPSILON << endl;

  cout << "Double" << endl;
  cout << "  Genaugkeit    : " << DBL_DIG << " Dezimalstellen" << endl;
  cout << "  Bereich       : " << DBL_MIN << "..." << DBL_MAX << endl;
  cout << "  Schritt       : " << DBL_EPSILON << endl;

  cout << "Long Double" << endl;
  cout << "  Genaugkeit    : " << LDBL_DIG << " Dezimalstellen" << endl;
  cout << "  Bereich       : " << LDBL_MIN << "..." << LDBL_MAX << endl;
  cout << "  Schritt       : " << LDBL_EPSILON << endl;
}
```

Die Genauigkeit gibt an, auf wieviele Dezimalstellen genau jeder Wert dargestellt wird. “Schritt” ist die kleinste in dem jeweiligen Typ darstellbare Zahl, die größer als 1 ist (d.h. es ist die kleinste Zahl mit 1.0 + Schritt != 1.0). Die Genauigkeit ist bei `float`–Typen allerdings nicht im ganzen Wertebereich gleich. Sie nimmt mit größeren Zahlen ab.

Die Werte sind maschinenabhängig. Eine mögliche Ausgabe ist:

```
Wertebereich der Float Typen
Float
  Genaugkeit    : 6 Dezimalstellen
  Bereich       : 1.17549e-38...3.40282e+38
  Schritt       : 1.19209e-07
Double
  Genaugkeit    : 15 Dezimalstellen
  Bereich       : 2.22507e-308...1.79769e+308
  Schritt       : 2.22045e-16
Long Double
  Genaugkeit    : 18 Dezimalstellen
  Bereich       : 3.3621e-4932...1.18973e+4932
  Schritt       : 1.0842e-19
```

Float–Literale

Float–Werte können auf zwei Arten durch Literale direkt angegeben werden: mit und ohne Skalierungsfaktor. `3.1415926` ist ein Float–Literal ohne und `5.1E2` eins mit *Skalierungsfaktor*. Der Skalierungsfaktor `E2` in `5.1E2` steht für 10^2. Der Skalierungsfaktor kann auch negativ sein. `-2E-2` beispielsweise steht für den Float–Wert $-0,02$.

Float–Operationen

Auf Float–Werten sind die üblichen arithmetischen Operationen definiert (+,−,⋆,/). Man beachte, dass Int– und Float–Operationen etwas völlig verschiedenes sind. Nicht nur ist `1/2` = 0 und `1.0/2.0` = `0.5`; auch die anderen Operationen werden intern unterschiedlich abgewickelt.

Der Vorteil der `float`–Zahlen gegenüber den `int`–Zahlen ist der größere Wertebereich, der auch gebrochene Zahlen umfasst. Dem stehen als Nachteile die aufwendige und langsamere Hardware zur Ausführung der Operationen, der größere Speicherbedarf und die Ungenauigkeit gegenüber.

5.6 Konversionen

Typ–Mix

Treffen Werte oder Variablen unterschiedlichen Typs in einem Ausdruck oder in einer Zuweisung zusammen, dann spricht man von einem Mix der Typen. In manchen Fällen ist ein solcher Mix erlaubt und die Werte werden konvertiert, in anderen nicht. Bei manchen Konversionen werden Bitsequenzen in andere transformiert, bei anderen nicht.

Konversionen zwischen integralen Typen

Am einfachsten hat es der Compiler bei Konversionen zwischen integralen Typen. Beispiele erlaubter Konversionen sind:

```
char   c = false; // bool -> char
int    i = '0';   // char -> int
bool   b = 48;    // int  -> bool
```

Bei Konversionen zwischen integralen Typen wird die interne Darstellung (die Bits) *nicht* verändert. `false` ist nur eine andere Art `0` zu sagen und `'0'` ist das gleiche wie `48`. Die Konversion besteht hier darin, dass den Bits "ein neues Mäntelchen" umgehängt wird. Das einzige was an Konversion wirklich gemacht wird ist, dass bei unterschiedlicher Größe mit Nullen aufgefüllt, bzw. führende Stellen weggestrichen werden.

```
long   l = 2097; // int   -> long  : auffuellen
short  s = 2097; // int   -> short : abschneiden
char   c = s;    // short -> char  : abschneiden (c = '1' = 49)
```

Konversionen zwischen Int– und Float–Typen

Bei den Konversionen zwischen Int– und Float–Typen, den *arithmetischen Konversionen,* muss die interne Repräsentation des Wertes umgewandelt werden. `1` ist ein `int`–Literal und bezeichnet darum eine Bitsequenz, die "in der Int–Welt" 1 bedeutet. In der "Float–Welt" bedeutet diese Bitsequenz ganz sicher *nicht* 1. Möglicherweise steht sie für `1.4013e-45` oder etwas ähnliches. Die Zuweisung

```
float f = 1; // int -> float
```

ist trotzdem erlaubt und führt auch zu einem sinnvollen Ergebnis: Aus der internen Darstellung der Int–Zahl 1 wird die interne Darstellung der äquivalenten Float–Zahl 1.0 erzeugt.

Konversionen nach oben

Die Konversion von Int nach Float ist immer möglich, denn zu jeder Int–Zahl gibt es eine äquivalente Float–Zahl. Umgekehrt gilt das nicht, darum wird eine Konversion in dieser Richtung vom Compiler mit einer Warnung quittiert:

```
int i = 1.0; // WARNUNG: float -> int
```

Die Tatsache, dass in diesem Fall der Float–Zahl 1.0 exakt die Int–Zahl 1 entspricht interessiert ihn dabei nicht weiter. Die notwendige detaillierte Analyse der Werte findet nicht statt, der Compiler beachtet nur den Typ der Werte, nicht die Werte selbst. Generell sind "Konversionen nach oben" hin zu einer höheren Genauigkeit immer erlaubt:

`char` → `short` → `int` → `long int` → `float` → `double` → `long double`

Implizite Konversionen in Ausdrücken

Diese Konversionen sind *implizit*: Treffen zwei arithmetische Ausdrücke mit unterschiedlichem Typ aufeinander, dann wird – ohne dass man es explizit angeben muss – der "schwächere" Ausdruck in den Typ des "stärkeren" konvertiert. Beispiel:

```
int   i=1;
float a=1.0;
a = (i + 2) * a; // OK:      Int-+, Float-*;
i = (i + 2) * a; // WARNUNG: Float -> Int
```

In der ersten Zuweisung

```
a = (i + 2) * a;
```

treffen bei "+" zwei Int–Werte zusammen also wird eine Int–Addition ausgeführt. Bei "`*`" kommen ein Int– und ein Float–Wert zusammen. Der Int–Wert wird implizit in einen Float–Wert konvertiert und dann die Float–Multiplikation ausgeführt. Die zweite Zuweisung

```
i = (i + 2) * a;
```

führt zu einer Warnung des Compilers, da hier der Float–Wert der rechten Seite "rückwärts" in einen Int–Wert konvertiert werden muss.

Man beachte, dass immer nur die Argumente und niemals das Ergebnis einer Operation über die Konversion entscheiden. So wird beispielsweise in

```
float a = 1/2; // => a = 0.0 !!
```

der Variablen `a` der Wert 0.0 (!) zugewiesen: bei der Division "`/`" treffen zwei ganzzahlige Werte zusammen. Es wird *nicht* konvertiert und eine Int–Division mit Ergebnis 0 (Int) ausgeführt. Diese 0 wird dann nach Float konvertiert und `a` zugewiesen.

Explizite Konversion (Cast)

Gelegentlich will man nicht auf die impliziten Konversionen allein angewiesen sein, sondern sie *explizit* erzwingen. Eine explizite Konversion wird auch oft *Cast* genannt. Sollen etwa zwei Int–Werte mit der Float–Division dividiert werden, dann muss mindestens ein Operand explizit nach Float konvertiert werden:

```
int   i = 1, j = 2;
float a;
a = i / j;          // => a = 0.0
a = float(i) / j;   // => a = 0.5
```

Mit dem Ausdruck `float (i)` wird der Int–Wert von `i` in den äquivalenten Float–Wert konvertiert. Damit treffen bei "`/`" ein Float– und ein Int–Wert zusammen. Der Int–Wert (von `j`) wird implizit nach Float konvertiert und eine Float–Division ausgeführt.

Konversion mit Konstruktor

Die verwendete Konstruktion der Form

$<Typ>(<Ausdruck>)$

ist eine *Konversion mit einem Konstruktor*. In `float(i)` ist `float` ein *Konstruktor*. Ein Konstruktor trägt den Namen eines Typs und konstruiert einen Wert von diesem Typ. Für alle vordefinierten Typen kann der Konstruktor benutzt werden, um Konversionen in diesen Typ zu erzwingen. Das geht auch ohne Warnung "rückwärts" von hoher nach geringerer Genauigkeit (*narrowing conversion*):

```
int   i;
float a;
...
i = int ((i + 2) * a); // OHNE WARNUNG: Float -> Int (Narrowing)
```

Konversion mit `static_cast`

Konstruktoren sind ein allgemeiner und sehr flexibler Mechanismus um Werte eines bestimmten Typs zu erzeugen. Nur für die in der Sprache vordefinierten Typen ist festgelegt, dass und wie sie als Konversionsoperatoren funktionieren. Die allgemeine explizite Konversionsoperation ist `static_cast`. Beispielsweise ist

```
static_cast<int> ((i + 2) * a);
```

äquivalent zur Konversion oben mit `int (...)`. Die allgemeine Form des `static_cast` ist:

`static_cast<<` Typ `>> (<` $Ausdruck$ `>)`

Auch bei der Konversion mit `static_cast` wird versucht ein sinnvolles dem Argument entsprechendes Ergebnis zu erzeugen. Bei Konversionen zwischen Float und Int ist das mit einer Transformation von Bitsequenzen verbunden. Da die Konversion stets sinnvolle Ergebnisse produzieren soll, kann man auch mit `static_cast` nicht zwischen beliebigen Typen konvertieren. Es sind genau gesagt nur die Konversionen explizit möglich, die auch implizit – ohne und mit Warnung – möglich sind.

5.7 Zeichenketten und Zahlen

Beispiel: Umwandlung einer Zeichenkette in eine Zahl

An einem etwas größeren Beispiel zeigen wir das Zusammenwirken der bisher behandelten Sprachmittel. Wir wollen ein Programm konstruieren, das eine Folge von Zeichen einliest und diese als Zahl im üblichen Dezimalsystem interpretiert, d.h. das Programm soll aus einer Zeichenfolge eine entsprechende Float–Zahl konstruieren.

Zunächst machen wir uns klar, dass eine eingegebene Zeichenfolge völlig unterschiedlich behandelt wird, je nachdem welchen Typ die Variable hat, in die eingelesen wird. Auch wenn jeweils `123` eingegeben wird, in den Variablen `s`, `i` und `f` werden völlig andere Werte (Bitkombinationen) nach den Eingabeanweisungen

```
string s; cin >> s;
int i; cin >> i;
float f; cin >> f;
```

zu finden sein:

- in `s` kann man neben Kontrollinformationen drei Zeichen `'1'`, `'2'` und `'3'` finden (in ASCII–Codierung `00110001`, `00110010` und `00110011`);
- in `i` wird man die Binärdarstellung der Zahl 123 finden (`0...001111011`) und
- in `f` ist die Float–Codierung der Zahl 123 zu finden (`01000010111101100...0`).

Die unterschiedlichen Bitsequenzen sind das Ergebnis unterschiedlicher Aktivitäten der Eingabefunktion. In jedem Fall werden die gleichen drei Zeichen `1`, `2` und `3` von der Tastatur gelesen, sie werden nur intern unterschiedlich weiter verarbeitet.

Ziffer–Zeichen in Zahlen umwandeln

Die ASCII–Codierungen der Ziffer–Zeichen `'0'`, `'1'`, `'2'`, ... sind aufsteigend geordnet:

- Code von `'0'` = `0011 0000` = 48
- Code von `'1'` = `0011 0001` = 49
- Code von `'2'` = `0011 0010` = 50
- etc.

Eine einzelne Ziffer kann darum leicht von einem Zeichen in eine Zahl umgewandelt werden. Der Zahlwert eines Zeichens in der Variablen `c` wird als dessen Distanz zum Code des Zeichens `'0'` bestimmt:

```
int j; j = c - '0';
```

Auf die Weise kann der Zahlwert jedes Zeichens in einer Zeichenkette `s` bestimmt werden:

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

using namespace std;

int main () {
  string s;

  cin >> s;
  for (int i=0; i < s.length(); ++i) {
    int j;
    j = s.at(i) - '0';  // Zeichen -> Ziffer
    cout << j;
  }
  cout << endl;
}
```

Wertigkeit von Vorkommastellen beachten

Aus dem Wert der einzelnen Ziffern kann der Wert ganzer Zahlen berechnet werden. Jede Ziffer muss dabei mit ihrer Wertigkeit multipliziert werden:

$$123 = 1 * 10^2 + 2 * 10^1 + 3 * 10^0$$

Die Wertigkeit ergibt sich aus der Position der Ziffer innerhalb der Zeichenkette:

$$123 = 1 * 10^{l-1} + 2 * 10^{l-2} + 3 * 10^{l-3}$$

wobei l die Länge der Zeichenkette ist. Zunächst konstruieren wir ein Programm, das jede Ziffer mit ihrer Wertigkeit multipliziert und dann ausgibt:

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

using namespace std;

int main () {
  string s;

  cin >> s;
  int l = s.length();
  for (int i=0; i < l; ++i) {
    float j;
    j = (s.at(i) - '0') * pow (10.0, l-(i+1)); // Ziffer * Wertigkeit -> Zahl
    cout << j << endl;
  }
}
```

Ein Programm das die Zeichenkette in eine entsprechende Zahl umwandelt, muss die Zahlwerte nur noch aufaddieren:

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

using namespace std;

int main () {
  string s;
```

```
  cin >> s;
  int l = s.length();
  float f = 0;
  for (int i=0; i < l; ++i) {
    float j;
    j = (s.at(i) - '0') * pow (10.0, l-(i+1));
    f = f + j;
  }
  cout << f << endl;
}
```

Verbesserter Algorithmus

Der Einsatz der ebenso mächtigen wie aufwendigen `pow`–Funktion, nur um eine Sequenz von 10–er Potenzen zu erzeugen, ist etwas übertrieben. Das geht einfacher, wenn wir nicht gleich mit der richtigen 10–er Potenz multiplizieren, sondern das bisherige Ergebnis durch Multiplikation mit 10 korrigieren, wenn eine neue Stelle in Betracht gezogen wird:

$$1234 \; = \; ((1 * 10 + 2) * 10 + 3) * 10 + 4$$

Als Programm:

```
...
cin >> s;
float f = 0;
for (int i=0; i < s.length(); ++i) {
  float j;
  j = s.at(i) - '0';
  f = f*10 + j;
}
cout << f << endl;
...
```

Nachkomma–Stellen verarbeiten

Die Stellen hinter dem Komma haben (genau wie die vor dem Komma) von links nach rechts eine immer kleinere Wertigkeit. Im Gegensatz zu den Vorkomma–Stellen haben wir es hier aber mit steigenden (negativen) Potenzen zu tun:

$$0,123 \; = \; 1 * 10^{-1} + 2 * 10^{-2} + 3 * 10^{-3}$$

Das macht die Bearbeitung der Nachkomma–Stellen eher einfacher:

```
...
float p = 10; // Wertigkeit: 1/p
for (int i=..erste Position hinter dem Komma..; i < s.length(); ++i) {
  float j;
  j = s.at(i) - '0';
  f = f + j*(1/p);
  p = p*10;
}
cout << f << endl;
...
```

Vor– und Nachkomma–Stellen verarbeiten

Ein Programm, das sowohl die Vorkomma–, als auch die Nachkomma–Stellen verarbeitet, könnte zuerst nach der Position des Kommas suchen, dann alle Zeichen davor mit dem Vorkomma–Algorithmus und alle Stellen danach mit dem Nachkomma–Algorithmus bearbeiten:

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

using namespace std;
```

```
int main () {
  string s;
  int komma_pos; // Position des Kommas

  cin >> s;

  komma_pos = s.length(); // Annahme es gibt kein Komma

  for (int i=0; i<s.length(); ++i) {
    if (s.at(i) == ',')
      komma_pos = i;
  }
  // komma_pos ist die Position des Kommas, falls ein Komma vorkommt
  // ansonsten ist es s.length()

  float f = 0;

  // Vorkomma-Stellen:
  for (int i=0; i < komma_pos; ++i) {
    float j;
    j = s.at(i) - '0';
    f = f*10 + j;
  }
  //Nachkomma-Stellen:
  float p = 10;
  for (int i=komma_pos+1; i < s.length(); ++i) {
    float j;
    j = s.at(i) - '0';
    f = f + j*(1/p);
    p = p*10;
  }
  cout << f << endl;
}
```

Vor– und Nackomma–Stellen in einer Schleife

Die Suche nach dem Komma kann “im Fluge” erfolgen. Wir durchlaufen die Zeichenkette mit dem Vorkomma–Algorithmus so lange bis wir auf ein Komma treffen, dann wird zum Nachkomma–Algorithmus gewechselt:

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

using namespace std;

int main () {
  float    p;   //nach Komma Wertigkeit
  string   s;   //eingelesene Zeichenkette
  float    f;   //float-Wert von s
  bool     vK;  //aktueller Zustand: vor oder hinter Komma

  cout << "string: ";
  cin >> s;

  f = 0;
  p = 10.0;  // NachKomma-Wertigkeit: ein 10-tel
  vK = true; // wir sind vor dem Komma
  for (int i=0; i<s.length(); ++i) {
    switch (s.at(i)) {
    case '.':
    case ',': vK = false; break; // Wechsel von Vorkomma auf Nachkomma

    case '0': case '1': case '2':
    case '3': case '4': case '5':
    case '6': case '7': case '8':
```

```
    case '9':
      switch (vK) {
      case true:                // wir sind vor dem Komma
        f = 10*f + s.at(i) - '0';
        break;
      case false:               // wir sind hinter dem Komma
        f = f + (s.at(i) - '0')/p;
        p = p*10;
        break;
      }
      break;
    default:
      cout << "Eingabe-Fehler " << s.at(i)
           << " wird ignoriert" << endl;
      break;
    }
  }
  cout << "Wert von " << s << " = " << f <<endl;
}
```

Nebenbei werden hier auch noch die Zeichen daraufhin geprüft, ob sie Bestandteil einer Dezimalzahl sein können.

5.8 Aufzählungstypen: `enum`

Aufzählungstypen sind selbst definierte Typen

Aufzählungstypen sind hier das erste Beispiel für einen *selbst definierten* Typ. Im folgenden Beispiel wird der Typ `Farbe` im Programm und nur für dieses Programm definiert:

```
#include <iostream>

using namespace std;

int main () {
  enum Farbe {rot, gelb, gruen, blau};  // neuer Typ
  Farbe klecks = gelb;                  // Variablendefinition
                                        // mit Initialisierung;
  klecks = gruen;                       // gruen ist ein Wert des Typs
  cout << "The klecks is ";
  switch (klecks) {
  case rot:   cout << "red"    << endl; break;
  case gelb:  cout << "yellow" << endl; break;
  case gruen: cout << "green"  << endl; break;
  case blau:  cout << "blue"   << endl; break;
  }
}
```

Hier wird der Typ `Farbe` definiert. Dieser Typ hat vier Werte mit den Namen `rot`, `gelb`, `gruen` und `blau`. `klecks` ist eine Variable mit dem neuen Typ. Ihr wird zuerst der Wert `gelb` und dann `gruen` zugewiesen. In der `switch`–Anweisung wird ihr Wert schließlich abgefragt und ausgegeben.

Allgemeine Form

Die Definition eines Aufzählungstyps hat die allgemeine Form:

`enum` $< Bezeichner >$ { $< Bezeichner_1 >, < Bezeichner_2 >, ...$ };

Hiermit wird ein Typ mit dem Namen $< Bezeichner >$ definiert. Zu diesem Typ gehören die Werte mit den Namen $Bezeichner_1$, $Bezeichner_2$, ...

Werte eines Aufzählungstyps

Die Werte eines `enum`–Typs werden intern wie `int`–Werte dargestellt. Die Codierungsregel ist sehr einfach: $< Bezeichner_1 >$ wird als 0 codiert, $< Bezeichner_2 >$ als 1, etc. Aufzählungstypen gehören damit zu den integralen Typen, deren Werte ja immer `int`–s unter unterschiedlichen Blickwinkeln sind.

Die Darstellung der `enum`–Werte kann bei Bedarf auch explizit kontrolliert werden:

```
enum Tag {Montag = 1, Dienstag, Mittwoch, Donnerstag,
          Freitag, Samstag, Sonntag};
```

Die Codierung der Wochentage beginnt hier mit 1 statt mit 0. Wird einem Wert explizit eine Code–Zahl zugeordnet, dann erhalten die folgenden die nächsten Zahlen als Code. Die Codierung kann völlig willkürlich gewählt werden, negative Werte, Lücken und sogar Wiederholungen beinhalten. In

```
enum E {e1 = -2, e2, e3 = 10, e4, e5 = -1, e6};
```

haben die Werte die Codierung `-2, -1, 10, 11, -1, 0`. Die einfache Form der Definition

`enum` $Bezeichner$ `{` $Bezeichner_1, Bezeichner_2, ...$ `};`

ist äquivalent zu

`enum` $Bezeichner$ `{` $Bezeichner_1$ `= 0,` $Bezeichner_2$ `= 1, ... };`

Literale, Namen und Werte

Die Werte eines `enum`–Typs haben Namen, es gibt *keine Literale* für die Werte! Literale sind Zeichenfolgen deren Bedeutung in der Sprache fest verdrahtet ist. Namen dagegen sind Bezeichner deren Bedeutung vom Programmierer frei gewählt werden kann. Beispielsweise ist in

```
const float pi = 3.1415;
```

`pi` ein Name für einen Float–Wert. Man hätte genausogut mit

```
enum pi { pi1, pi2, pi3};
```

`pi` als Namen für einen neuen Typ einführen können. Dagegen ist `3.1415` ein Float–Literal, das immer und in allen Programmen eine ganz bestimmte Float–Zahl bezeichnet.

Der *Wertebereich* eines `enum`–Typs erstreckt sich von Null bis zur nächst größeren 2er–Potenz minus 1, die größer ist als alle angebenen Werte. Enthält der `enum`–Typ negative Werte dann geht der Wertebereich bis zur kleinsten negativen 2er–Potenz plus eins. Beispiele sind:

`enum A { aa, bb };`	Wertebereich: 0 bis 1
`enum B { aa = 2, bb = 9 };`	Wertebereich: 0 bis 15
`enum C { aa = -2, bb = 0, cc = 5};`	Wertebereich: -7 bis 7
`enum D { aa = -3, bb = 0, cc = 1};`	Wertebereich: -3 bis 3

Achtung: Der Wertebereich wird bei `int`–Konversion *nicht* (unbedingt) geprüft!

```
klecks = Farbe(88);
```

wird vom Compiler darum anstandslos akzeptiert und später dann auch ausgeführt.

Der Compiler unterscheidet streng zwischen Aufzählungstypen

Verschiedene `enum`–Typen werden vom Compiler unterschieden. Auch wenn die Wertebereiche zweier Aufzählungstypen völlig gleich sind, verweigert der Compiler jeden Typmix. Dabei ist es auch egal, ob die geforderte Konversion implizit oder explizit ist:

```
enum Hund  {fifi, hektor,  waldi};
enum Katze {susi, schnurr, mauz};
Katze k;
Hund  h;
k = susi;                        // OK
h = fifi;                        // OK
k = h;                           // WARNUNG/FEHLER: enum-Konversion
h = mauz;                        // WARNUNG/FEHLER: enum-Konversion
k = static_cast<Katze> (waldi); // FEHLER (trotz expliziter
                                 // Konversion): enum-Konversion
```

Der Compiler ist liberal beim Mix von Aufzählungstypen und anderen integralen Typen

Während der Compiler bei den enum–Typen untereinander sehr streng darauf achtet, dass alles schön getrennt bleibt, betrachtet er die Vermischung von enum–Typen mit anderen integralen Typen mit der gleichen multikulturellen Liberalität, mit der er generell die Vermischung der integralen Typen untereinander akzeptiert. Konversionen *aus* einem enum–Typ *heraus* werden anstandslos hingenommen und implizit ausgeführt. Konversionen *in* einen enum–Typ *hinein* dagegen werden mit Warnungen oder Fehlermeldungen bedacht. (Das entspricht dem Verhalten bei der Konversion zwischen *int* und *float*):

```
enum Hund  {fifi, hektor,  waldi};
enum Katze {susi, schnurr, mauz};

Katze k = mauz;                // OK
int   i = susi + fifi + k;     // OK, Katze -> int, Hund -> int

k = 1;                         // WARNUNG: int -> Katze
k = Katze (1);                 // OK,      int -> Katze
k = static_cast<Katze> (1);    // OK,      das gleiche
k = susi + schnurr;            // WARNUNG: int -> Katze
k = int(susi) + int(schnurr); // WARNUNG: int -> Katze
k = static_cast<Katze> (
       susi + fifi + k);           // OK, explizit in den enum-Typ
k = static_cast<Katze> (waldi);    // FEHLER enum -> enum
k = static_cast<Katze> (waldi+1); // OK     enum -> int -> enum
```

In der Zuweisung

```
k = int(susi) + int(schnurr);
```

ist die entscheidende Konversion die von int in den Typ Katze. Sie ist implizit und wird darum mit einer Warnung bedacht. Die explizite Konversion von susi und schnurr nach int ist hier völlig überflüssig, sie würde auch ohne Warnung implizit ausgeführt.

Verwendung von Aufzählungstypen

Die wichtigste Anwendung von Aufzählungstypen besteht darin, dass man mit ihnen int–Werten schöne – d.h. aussagekräftige – Namen geben kann. Ein Programm, das mit Wochentagen arbeitet, kann selbstverständlich so geschrieben werden, dass 0 für “Sonntag”, 1 für “Montag”, und so weiter steht. Die Typdefinition

```
enum Tag {Sonntag, Montag, Dienstag, Mittwoch,
          Donnerstag, Freitag, Samstag, Sonntag};
```

macht diese Konvention jedoch explizit im Programm sichtbar. Das macht das Programm nicht nur besser lesbar:

```
if (t == Sonntag) ...
```

statt

```
if (t == 0) ...
```

Es hilft auch Fehler zu vermeiden. Wer weiß am nächsten Tag, oder gar bei einer kommenden Umstellung der Jahreszahlen auf fünfstellige Werte, mit welchem Wochentag die Woche beginnt und ob der erste Wert Codewert 0 oder 1 war.

Dadurch, dass der Compiler so streng darüber wacht, dass Aufzählungstypen nicht mit anderen vermischt werden, kann er auch viele versehentliche Programmierfehler aufdecken.[17]

Beispiel: Aufzählungstyp als Zustand des Programms

Im folgenden Beispiel soll eine Zeichenkette in float–Zahl umgewandelt werden. Das Programm verarbeitet ein Zeichen nach dem anderen. Die Verarbeitung eines Zeichens ist aber unterschiedlich, je nachdem, ob das Zeichen vor oder hinter dem Komma zu finden ist. Wir führen darum eine Varibale z wie “aktueller Zustand” ein, deren Wert angibt, ob das Komma schon gelesen wurde oder nicht.

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

[17] Ältere Compiler haben eventuell Probleme mit der korrekten Behandlung von Aufzählungstypen.

```
using namespace std;

int main () {
  enum Z {vorK, nachK};  // Zustand:
                         // vor oder hinter Komma
  int      p;            // Nach-Komma Wertigkeit
  string   s;
  float    v;
  Z        z;

  cout << "string: ";   cin >> s;

  v = 0; p = 10; z = vorK;
  for (int i=0; i<s.length(); ++i) {
    switch (s.at(i)) {
    case '.': case ',':
      z = nachK; break;

    case '0': case '1': case '2': case '3':
    case '4': case '5': case '6': case '7':
    case '8': case '9':
      switch (z) {
      case vorK:
        v = 10*v + s.at(i) - '0';
        break;
      case nachK:
        v = v + (float(s.at(i) - '0')/p);
        p = p*10;
        break;
      }
      break;

    default:
      cout << "Eingabe-Fehler\n"; break;
    }
  }
  cout << "Wert von " << s << " = " << v <<endl;
}
```

5.9 Namen für Typen: `typedef`

`typedef` gibt Typen einen neuen Namen

Mit `typedef` kann einem Typ ein Name gegeben werden. Der Typ selbst kann dabei namenlos sein – dazu werden wir später Beispiele kennenlernen – oder bereits einen Namen haben. Im folgenden Beispiel wird den Typen `short`, `string` und `float` jeweils ein weiterer neuer Name gegeben.

```
typedef short       s_int;
typedef string      Name;
typedef float       Kilo;

s_int  i;
Name   author;
Kilo   aepfel;
```

Die neuen Namen `s_int`, `Name` und `Kilo` sind Synonyme der alten. Man kann `typedef` verwenden, wenn man keinen Gefallen am offiziellen Namen findet – `s_int` statt `short` –, oder auch um die Lese– und Änderbarkeit des Programms zu verbessern. So ist `Kilo` eine etwas aussagekräftigere Mengenbezeichnung als `float`. Mit der Definition

```
typedef string Name;
```

kann ein Wechsel in der Darstellung von Namen mit der Änderung dieser einen Textstelle bewerkstelligt werden.

Allgemeine Form einer typedef Definition

Die Form einer typedef *Definition* ist:

typedef $< Typ > < Bezeichner >$;

Mit ihr wird $< Bezeichner >$ als Name (Synonym) für $< Typ >$ eingeführt. Es ist *kein* neuer Typ. Der Compiler wird also keinerlei Warnungen oder Fehlermeldungen erzeugen, wenn Altes und Neues vermischt werden:

```
typedef float       Kilo;
typedef float       Gramm;
typedef float       Birnen;

Kilo   aepfel = 5.2;
Birnen b1     = 12.65;
Gramm  gold   = 0.2;
gold = aepfel + b1;     // OK keine Fehlermeldung!
```

Im Gegensatz zu den enum–Typen ist also typedef kein besonders gutes Mittel um Verwechslungen von Variablen und Werten mit unterschiedlicher Bedeutung aber gleicher Codierung zu vermeiden. Sein Einsatz wird trotzdem sehr empfohlen, da die Lesbarkeit eines Programms mit diesem Konstrukt wesentlich verbessert werden kann.

5.10 Übungen

Aufgabe 1

1. Erläutern Sie kurz den Unterschied zwischen Typen und Variablen am Beispiel “Hund” und “Fifi”.
2. Was ist falsch an:

```
typedef float Gewicht;
Gewicht = 12.5;
```

3. Wieviele Namen werden in folgendem Programmstück mit welcher Bedeutung definiert:

```
enum Ton { doh, reh, mih, fah, soh, lah, tih };
typedef Ton Klang;
Ton     t1, t2;
Klang   k = fah;
t1 = reh;
k = t1;
```

4. Was ist falsch an:

```
enum Ton { do, re, mi };
```

Aufgabe 2

1. Die rationalen Zahlen enthalten als Teilmenge die ganzen Zahlen. Ist der Datentyp `int` eine Teilmenge des Datentyps `float`?
2. In alten Zeiten wurden Rechner verkauft, deren Hardware nicht in der Lage war Gleitkommazahlen zu verarbeiten. Können auch auf einem solchen Rechner beliebige C++–Programme ablaufen?
3. Was ist falsch an: `string s = 'a'; char c = "d";`?
4. Welchen Wert bezeichnen die Literale `10`, `010`, `0x10`, `'1'`?
5. Wenn das Programm

```
#include <iostream>
using namespace std;
int main () {
  short           x = -1; // == signed short x = -1;
  unsigned  short y = x;
  cout << "x: " << x << endl;
  cout << "y: " << y << endl;
}
```

   die Ausgabe

```
x: -1
y: 65535
```

   erzeugt, wie viele Bytes hat dann ein `short`?

6. Das Programm

```
#include <iostream>
using namespace std;
int main () {
  int i = 32;
  for (char c = 32; c < 128; ++c, ++i)
    cout << i << ": " << c << endl;
}
```

   enthält eine Endlosschleife. Warum, wie kann dieser Fehler korrigiert werden?

7. Welche Werte werden genauer dargestellt `int`– oder `float`–Werte?
8. Was passiert, wenn man zu der größten Integer–Zahl eins addiert?
9. Was passiert, wenn man durch Null dividiert?
10. Korrekt oder nicht: `enum primzahl {2, 3, 5, 7, 11};`
11. Welche Fehler enthält: `enum farbe {rot, gelb, nachts}; Farbe gelb; int rot;`
12. `a` sei vom Typ `float`, und `i, j` vom Typ `int`. Es gelte `i == 1` und `j == 2` Bestimmen Sie den Wert von `a` nach folgenden Zuweisungen.
    - `a = float(i)/float(j);`
    - `a = float(i)/j;`
    - `a = i/float(j);`
    - `a = i/j;`
    - `a = j/2;`
    - `a = 1/2 * j;`
    - `a = float(1/2 * j);`
    - `a = float(1/2) * j;`
    - `a = float(1)/2 * j;`
    - `a = 1/float(2) * j;`
13. Welchen Wert hat `(true && (30 % 7)) / 5` ? Ist dies überhaupt ein legaler Ausdruck in C++?
14. Welche Fehler enthält folgendes Programmfragment:

```
enum AA {aa, bb, cc};
enum XX {xx, yy, zz};
AA a = aa;
XX x = xx;

a = int (x);
a = x;
a = x + 1;
a = int(x) + 1;
a = AA(int(x) + 1);
```

Aufgabe 3

Schreiben Sie ein Programm, das alle ASCII–Zeichen mit ihrer Codierung ausdruckt.

Aufgabe 4

Schreiben Sie ein Programm, das die Wahrheitstafel des folgenden Ausdrucks ausgibt (alle Argumente mit entsprechendem Wert des Gesamtausdrucks):

$P \rightarrow ((Q \rightarrow P) \wedge R)$

Ihr Programm soll die unterschiedlichen Werte für P, Q, R in `for`–Schleifen durchlaufen.

Aufgabe 5

Konversionen können implizit (I) oder explizit (E) sein. Bei einer Konversion kann der konvertierte Wert *inhaltlich* erhalten bleiben (Wert vor und nach der Konversion "gleich") (WE) oder verändert (WV) werden. Bei einer Konversion kann die interne Bit–Repräsentation erhalten bleiben (RE) oder verändert werden (RV). Eine Konversion kann also mit drei Attributen klassifiziert werden:

- Entweder *implizit (I)*, oder *explizit (E)*.

- Entweder *Wert–erhaltend (WE)* oder *Wert verändernd (WV)*.
- Entweder *Repräsentation–erhaltend (RE)* oder *Repräsentation–verändernd (RV)*.

Geben Sie Beispiele für folgende Konversionen an:

1. I WE RV
2. E WE RE

Aufgabe 6

1. Statt eines Aufzählungstyps wie

   ```
   enum Farbe { rot, gelb, gruen, blau};
   ```

   kann man auch Konstanten verwenden. Z.B.:

   ```
   typedef   int   Farbe;
   const Farbe rot   = 0;
   const Farbe gelb  = 1;
   const Farbe gruen = 2;
   const Farbe blau  = 3;
   ```

   welche *Nachteile* hat das?

2. Zu den Aufzählungstypen gibt es keine speziellen Literale. Warum nicht?

Aufgabe 7

Schreiben Sie ein Programm, das eine ganze Zahl einliest, in einer Variablen `i` vom Typ `int` abspeichert und dann das Bitmuster in `i` ausgibt.
Hinweis: die Funktion `sizeof` liefert die Zahl der Bytes, die von Variablen eines bestimmten Typs belegt werden. Z.B. `sizeof(int)` liefert die Zahl der Bytes, die eine Int–Variable belegt. Gesucht ist die "logische" Bitfolge im Speicher, d.h. die Byteordnung der jeweiligen Maschine kann ignoriert werden.
Beispiel:
Eingabe: `15` Ausgabe: `00000000000000000000000000001111`
Eingabe: `-16` Ausgabe: `11111111111111111111111111110000`

Aufgabe 8

Schreiben Sie ein Programm, das eine ganze Zahl b mit $2 \leq b \leq 16$ sowie eine Zeichenkette s einliest, s dann als positive ganze oder gebrochene Zahl zur *Basis b* interpretiert und den entsprechenden Wert als Zahl zur Basis 10 ausgibt. Falsche Eingaben sind abzufangen und zurückzuweisen. Beispiel:
Eingabe: `16` und `4AB` Ausgabe: `1195` (Denn $1195 = 4 \cdot 16^2 + 10 \cdot 16^1 + 11 \cdot 16^0$)
Eingabe: `7` und `66.6` Ausgabe: `48.8571`
Eingabe: `8` und `8` Ausgabe: `EINGABE-FEHLER!`

5.11 Lösungshinweise

Aufgabe 1

1. Typen sind Informationen darüber, wie Dinge aussehen, bzw. wie sie zu erzeugen sind. Variablen sind die entsprechenden Dinge. “Hund” ist der Typ und “Fifi” ist die Variable.
2. `Gewicht` ist ein Typ. Einem Typ kann kein Wert zugewiesen werden.
3. Es werden definiert:
   - 2 Typen mit den Namen: `Ton` und `Klang`;
   - 7 Werte mit den Namen: `doh`, `reh`, `mih`, `fah`, `soh`, `lah`, `tih`;
   - 3 Variablen mit den Namen: `t1`, `t2`, `k`.
4. `do` ist ein Schlüsselwort, also ein Wort mit einer festgelgten unveränderlichen Bedeutung. Es kann darum weder als Variablen–, Typ– oder Wertnamen verwendet werden.

Aufgabe 2

1. Der Datentyp `int` ist *keine* Teilmenge des Datentyps `float`.
2. Ja, die Gleitkommazahlen werden dann rein in Software realisiert.
3. Die Anweisungen in

   ```
   string s = 'a'; char c = "d";
   ```

   sind nicht erlaubt. Ein einzelnes Zeichen ist kein Sonderfall einer Zeichenkette und umgekehrt ist auch eine einelementige Zeichenkette kein Zeichen. Beachten Sie unterschiedlichen Anführungszeichen!
4. Die Literale bedeuten:
   - `10`: Ganzzahl (Datentyp `int`) 10
   - `010`: Ganzzahl (Datentyp `int`) oktale Notation 8
   - `0x10`: Ganzzahl (Datentyp `int`) hexadezimale Notation 16
   - `'1'`: Zeichen 1 Ganzahl “in Zeichennotation” (Datentyp `char`) 49
5. $2^{16}-1 = 65535$ d.h 16 Bits = 2 Bytes sind voll mit 1–en. Der Übergang von `signed` zu `unsigned` in `y = x` verändert nicht die Bits im Speicher, sondern nur deren Interpretation. Um etwas als 65535 interpretieren zu können werden mindestens 16 Bits benötigt. Im 2er Komplement ist eine -1 durch “alles 1” dargestellt. Wenn “alles 1” 65535 ergibt, dann müssen es 16 Stellen mit 1–en sein. Mehr als zwei Bytes voller 1–en hätten zur Ausgabe einer größeren Zahl geführt. Ein `short` belegt auf diesem Rechner also zwei Bytes.
6. In der Schleife

   ```
   for (char c = 32; c < 128; ++c, ++i) ...
   ```

   terminiert nicht, da 128 binär 8–stellig ist ($128 = 2^7$ = `10000000`). Das erste Bit bei `char` (= `signed char`) bewirkt jedoch dass die dargestellte Zahl negativ ist. Im 2–er Komplement des Rechners bedeutet eine 1 gefolgt von $n-1$ 0–en den Wert -2^{n-1}. In unserem Fall mit einer 1 gefolgt von 7 Nullen also $-2^{8-1} = -2^7 = -128$. Das ist die kleinste darstellbare Zahl im Typ `signed char`.

   `++c` in der Schleife führt darum von 127 (= `01111111`) zu -128 (= `10000000`), ohne +128 jemals zu erreichen. Mit `c < 128` wird in der Schleifenbedingung zwar

   ```
   signed char 1000 0000 < signed int 0..0 1000 0000
   ```

   geprüft, der Vergleich liefert aber immer `true`, da die Typen der Werte dabei mit in Betracht gezogen werden.
7. `int`–Werte werden entweder gar nicht oder völlig exakt dargestellt. `float`–Werte dagegen werden meist nur angenähert.
8. Was passiert, wenn man zu der größten Integer–Zahl eins addiert?
   Das kann man mit einem einfachen Programm testen:

```
#include <iostream>
#include <climits>
using namespace std;
int main () {
  int large = INT_MAX;

  cout << "Maximale int-Zahl    :" << large << endl;
  large++;
  cout << "Maximale int-Zahl + 1:" << large << endl;
}
```

9. Was passiert, wenn man durch Null dividiert?
   Auch das lässt sich einfach testen. Tun Sie das!

10. Nicht korrekt: Zahlen sind Literale und keine Bezeichner, sie sind darum nicht als `enum`–Wertnamen erlaubt.

11. `enum farbe {rot, gelb, nachts}; Farbe gelb; int rot;`
    Fehler: `Farbe` ist nicht definiert, Gross– und Klein–Schreibung beachten!

    Fehler: `int rot;` ist nicht erlaubt `rot` ist schon als Wert des `enum`–Typs vergeben. Für `gelb` gilt das Gleiche.

12. `i == 1, j == 2`

    - `a = float(i)/float(j);` a: 0.5
    - `a = float(i)/j;` a: 0.5
    - `a = i/float(j);` a: 0.5
    - `a = i/j;` a: 0.0
    - `a = j/2;` a: 1.0
    - `a = 1/2 * j;` a: $0 * 2.0 = 0.0$
    - `a = float(1/2 * j);` a: 0.0
    - `a = float(1/2) * j;` a: 0.0
    - `a = float(1)/2 * j;` a: $1.0/2 * 2.0 = 1.0/2.0 * 2.0 = 1.0$
    - `a = 1/float(2) * j;` a: 1.0

13. `(true && (30 % 7)) / 5` sollte auch auf Ihrem System korrekt sein und den Wert "0" liefern. Berechnen Sie bitte die Teilausdrücke und versuchen Sie dieses Ergebnis nachzuvollziehen!

14. ```
    enum AA {aa, bb, cc};
    enum XX {xx, yy, zz};
    AA a = aa;
    XX x = xx;

    a = int (x);     // FEHLER: int(x) ist OK, aber
                     //         keine implizite Konv. von int nach AA!
    a = x;           // FEHLER: keine implizite Konv. von XX nach AA!
    a = x + 1;       // FEHLER: implizite Konv. von X nach int
                     //         vor der Addition ist OK, aber
                     //         keine implizite Konv. von int nach AA!
    a = int(x) + 1; // FEHLER: int(x) ist ueberfluessig,
                     //         keine implizite Konv. von int nach AA!
    a = AA(int(x) + 1);
    ```

    Achtung C trennt `int` und `enum`–Typen sowie `enum`–Typen untereinander weniger streng als C++. Nicht jeder C++–Compiler – speziell, wenn er etwas betagt ist – hält sich an die Regeln von C++!

Aufgabe 3

Hier ist zu bedenken, dass ASCII ein 7–Bit Code ist. Ihr Programm muss also alle Kombinationen von 7 Bits (0 – 127) durchlaufen und den entsprechenden `int` und `char`–Wert ausgeben. Beispielsweise so:
    ```

```
#include <iostream>
using namespace std;
int main () {
  char c;

  c = 0;
  cout << int(c) << " " << c << endl;
  do {
    c++;
    cout << int(c) << " " << c << endl;
  } while (c != 127);
}
```

Etwas umständlich um Vergleiche mit 128 zu vermeiden.

Aufgabe 4

Schleifen über Boole–sche Werte laufen zu lassen ist etwas schwierig, da der Datentyp nur zwei verschiedene Werte hat.
`for (P=false ; P<=true; P++)` ist eine Endlos–Schleife!
`for (P=false ; P<true; P++)` ist zuwenig!

Die Schleife sollte darum über zwei `int`–Werte laufen, die dann konvertiert werden.

Aufgabe 5

1. Implizit, Wert–erhaltend und Repräsentation–verändernd:
   ```
   float f = 1;
   ```
2. Explizit und Repräsentation–erhaltend:
   ```
   enum Ton {so, re, mi, fa, la, ti};
   enum Farbe {rot, gelb, blau, gruen};

   Ton t = ti;
   Farbe f;

   f = static_cast<Farbe> (static_cast<int> (t));
   ```

   Hier kann man sich allerdings streiten, ob diese Konversion Wert–erhaltend oder Wert–verändernd ist. Was ist ein “Wert”? Eine Farbe kann kein Ton sein, andererseits sind die Farben und Töne ja nichts anderes als verkleidete `int`–Werte, die durch die Konversion nicht verändert werden.

Aufgabe 6

- Im Gegensatz zu Konstanten wacht der Compiler über die “Reinheit” der Enum–Typen. Eine Int–Konstante kann problemlos (irrtümlich !) mit anderen Int–Werten mit anderer Bedeutung gemischt werden. Beispiel:

  ```
  typedef   int   Farbe;
  const Farbe rot   = 0;
  const Farbe gelb  = 1;
  const Farbe gruen = 2;
  const Farbe blau  = 3;

  typedef   int   Tonart;
  const Tonart dur   = 0;
  const Tonart moll  = 1;

  Tonart a; Farbe b;
  ...
  a = b;  // Wahrscheinlich ein Programmfehler
  ...     // aber keine Fehlermeldung des Compilers!
  ```

Bei Aufzählungstypen erzeugt der Compiler Warnungen oder Fehler.

- Zu den Aufzählungstypen kann es keine speziellen Literale geben. Literale sind fest für die Sprache insgesamt und für immer definiert. Aufzählungstypen können jedoch nach Belieben im Programm eingeführt werden.

Aufgabe 7

Kein Lösungshinweis.

Aufgabe 8

Kein Lösungshinweis.

6 Felder und Verbunde

6.1 Felder sind strukturierte Variablen

Definition eines Felds

Aus der Mathematik sind Vektoren und Matrizen als strukturierte Werte bekannt. Ein Vektor beispielsweise ist strukturiert, weil er aus Teilwerten – seinen Komponenten – besteht. Der Vektor $\vec{a}$ mit

$$\vec{a} = \begin{pmatrix} 5 \\ 2 \end{pmatrix}$$

hat beispielsweise die Komponenten $a_1 = 5$ und $a_2 = 2$.

In C++ kann ebenfalls mit Vektoren, Matrizen, oder ganz allgemein mit Kollektionen von Werten gearbeitet werden. Man spricht dann von einem *Feld* oder engl. einem *Array*. Ein Feld `a` mit zwei Int–Werten wird mit der Variablendefinition

`int a[2];` → erzeugt das Feld → a 

definiert. `a` ist der *Feldanme*, `int` der Typ der Komponenten und `2` die Größe (Zahl der Elemente). Mit dieser Definition wird eine Variable `a` eingeführt, die dann im weiteren Programm genutzt werden kann. Die erste Komponente hat den *Index* `0`, die zweite hat den Index `1`. Die Definition wie

```
int a[2];
```

sagt also, dass das Feld `a` zwei Komponenten hat. Die allgemeine Form einer Feld-Definition ist:

<Typ> *Name* `[`*<Größe>*`];`

Zugriff auf Feldelemente

Die Komponenten des Felds `a` können in Zuweisungen belegt werden und ihr Wert kann in Ausdrücken festgestellt werden, z.B.:

```
a[0] = 5;          // Zuweisung an erste Komponente von a
a[1] = a[0]-3;     // Zuweisung an zweite Komponente von a
```

Hier wird die erste Komponente (Index `0`) auf 5 und die zweite (Index `1`) auf 2 ($= 5 - 3$) gesetzt:

a: 

`a[0]` ist die erste und `a[1]` die *zweite* Komponente von `a`.

Ein vollständiges kleines Programmbeispiel mit Feldern ist:

```
int main () {
  // Definition von zwei Feldern
  int a[2];   // a hat 2 Komponenten
  int b[3];   // b hat 3 Komponenten

  a[0] = 1;
  a[1] = 2;
  b[0] = a[0] + 1;    // b[0] hat jetzt den Wert 2
  b[1] = a[0] + a[1]; // b[1] hat jetzt den Wert 3
  b[2] = b[0] + b[1]; // b[2] hat jetzt den Wert 5

  // b mit Schleife belegen (b[0]=0, b[1]=1, b[2]=2)
  for ( int i=0; i<3; ++i)
    b[i] = i;
}
```

Ein häufiger Fehler besteht darin, dass über die Feldgrenze hinaus gegriffen wird:

```
int a[10];
...
a[10] = ...  // FEHLER: a[10] ist nicht mehr in a!
```

Achtung, der Compiler bemerkt solche Fehler nicht!

Felder sind strukturierte Variablen

Feldkomponenten können wie ganz normale Variablen verwendet werden – es sind ganz normale Variablen. Auf den Wert eines Feldes als ganzes kann dagegen nicht zugegriffen werden. Das Feld selbst ist darum keine normale Variable. Felder sind *strukturierte Variablen*. Die wichtigste Konsequenz dieser Tatsache ist, dass Felder nicht als ganzes zugewiesen oder verglichen werden können. Will man also zwei Felder `a` und `b` vergleichen, dann muss jedes aller Elemente von `a` und `b` jeweils extra verglichen werden. Mit der Zuweisung verhält es sich genauso. Ein etwas ausführlicheres Beispiel ist:

```
#include <iostream>

using namespace std;

int main () {
  int a[2] = {5, 2};  // a hat 2 Komponenten und zwei Initialwerte
  int b[2];           // b hat zwei Komponenten

  b[0] = a[0];                 // |- Zuweisung an b (statt a = b)
  b[1] = a[1];                 // |

  bool gleich = true;          // |
  for (int i = 0; i < 2; ++i)  // |
    if (a[i] != b[i])          // |- Vergleich von a und b (statt if(a == b))
      gleich = false;          // |
                               // |
  if (gleich)                  // |
    cout << "a und b sind gleich" << endl;
}
```

In diesem Beispiel werden zwei Felder `a` und `b` definiert. `a` wird initialisiert, dann wird der Wert von `a` an `b` zugewiesen und schließlich wird geprüft ob beide gleich sind. Da es den Wert von `a` als einzelnes Objekt nicht gibt, müssen Zuweisung und Vergleich komponentenweise erfolgen.[18]

Beispiel, Programm mit einem Feld

Das folgende Programm gibt die geraden Zahlen von 0 bis 18 aus.

```
#include <iostream>

using namespace std;

int main () {
  int a[10];

  for (int i=0; i<10; ++i){
    a[i] = 2*i;
  }
  for (int i=0; i<10; ++i){
    cout << a[i] << " ";
  }
  cout << endl;
}
```

Das Feld wird hier zweimal mit einer For–Schleife durchlaufen. In der ersten Schleife wird es mit Werten belegt. In der zweiten werden diese ausgegeben. Die beiden Schleifen sind typisch für die Bearbeitung eines Feldes. Mit

```
Typ a[Größe];
...
for (int i = 0; i < Größe; ++i)
... a[i] ...
```

wird ein Feld angelegt und systematisch alle Feldelemente bearbeitet.

[18] Manche Compiler (–Schreiber) denken sie müssten "besser" als der Sprachstandard sein und erlauben die Zuweisung ganzer Felder. Diese Möglichkeit sollte jedoch niemals ausgenutzt werden.

Feld–Typen und Feld–Variablen

Mit einer Definition wie `int a[2];` wird eine Variable `a` mit ihren Teilvariablen `a[0]` und `a[1]` definiert. `a[1]` und `a[0]` sind vom Typ `int`. `a` selbst hat den Typ "Feld von zwei Int–Variablen". `a` ist eine Variable und hat auch einen Typ. Dieser Typ hat aber keinen Namen – es ist ein *anonymer Datentyp*. Mit Hilfe von `typedef` kann man Feldtypen einen Namen geben. Beispiel:

```
typedef int A[2];   // A ist der Typ der Int-Felder mit zwei Komponenten
A  a;               // a hat den Typ A: es ist ein Int-Feld mit zwei Komponenten
```

Man sieht: durch das Voranstellen von `typedef` wird aus einer Variablen– eine Typdefinition. Der so definierte Typ kann dann in weiteren Variablendefinitionen verwendet werden. Als weiteres Beispiel definieren wir zwei Int–Felder `a1` und `a2` mit jeweils zwei Komponenten auf drei unterschiedliche Arten:

```
// Mit typedef          // mit anonymen Typen   // auch moeglich
typedef int A[2];       int a1[2];              int a1[2],
A  a1, a2;              int a2[2];                  a2[2];
```

Natürlich ist die Definitionen von Feldtypen nicht auf Int–Felder beschränkt. Mit

```
typedef float F[10];
```

wird `F` als der Name des Typs der Felder mit 10 `float`–Komponenten definiert.

Nach einer weitverbreiteten Programmierkonvention beginnen die Namen von Typen immer mit einem Großbuchstaben. Das macht die Programme besser lesbar, hat aber sonst keine Konsequenzen. Wir empfehlen die Einhaltung dieser Konvention. Programme, in denen die Bezeichner ohne jede Systematik vergeben werden, machen einen sehr unprofessionellen Eindruck. Selbstverständlich muss man sich nicht unbedingt an diese Konvention halten und so können Feldtypen auch Kleinbuchstaben als Namen haben:

```
typedef  int a[10];  // a ist der Typ der Int-Felder mit 10 Komponenten
a        b, C;       // b und C sind Felder vom Typ a
```

Zusammengefasst:

- `int a[10];` `a` ist eine Variable mit anonymem Typ
- `int a[2], b[4];` `a` und `b` sind Variablen mit anonymem Typ
- `typedef int A[10];` `A` ist ein Typ
- `typedef int A[10];` `A` ist ein Typ,
  `A a1, a2;` `a1` und `a2` sind Variablen vom Typ `A`.

6.2 Indizierte Ausdrücke, L– und R–Werte

Indizierter Ausdruck

Ein *indizierter Ausdruck* ist ein Ausdruck in dem ein Index vorkommt. Die Zuweisung

```
a[2] = i + b[i];
```

enthält zwei indizierte Ausdrücke `a[2]` und `b[i]`. Mit ihr soll der dritten Komponente von `a` die Summe des aktuellen Wertes von `i` und des aktuellen Wertes der i–ten Komponente von `b` zugewiesen werden. (Genau genommen ist es der Wert der (Wert–von–`i`)–ten Komponente von `b`.)

Auswertung indizierter Ausdrücke

Indizierte Ausdrücke werden stets von innen nach außen ausgewertet.

- Beispiel `a[i]` :
  - Zuerst wird der Wert von `i` bestimmt, angenommen etwa zu i,
  - dann wird `a[` i `]` bestimmt: das Ergebnis ist eine Teilvariable (ein Element) von `a`
  - Findet sich der Ausdruck links vom Zuweisungszeichen, dann ist die Auswertung beendet.

  - Rechts vom Zuweisungszeichen geht es noch einen Schritt weiter mit der Bestimmung des aktuellen Wertes von `a[`i`]`.

- Beispiel `a[i]+i` :

  Dieser Ausdruck kann nur rechts vom Zuweisungszeichen erscheinen. Als erstes wird der Wert von `i` bestimmt (z.B. i). Damit wird dann `a[`i`]` identifiziert und der Wert dieser Teilvariablen von `a` bestimmt (z.B. k). Der Wert des Gesamtausdrucks ist dann $k + i$.

- Beispiel `a[i+i]` :

  Dieser Ausdruck kann links und rechts vom Zuweisungszeichen erscheinen.

  Links bezeichnet er die Teilvariable (das Element) `a[`k`]`, dabei sei $k = i + i$ und i der Wert von `i`, rechts bezeichnet er den Wert den diese Teilvariable aktuell hat.

l–Wert, r–Wert von ganzen Feldern

Ausdrücke bedeuten links und rechts vom Zuweisungszeichen etwas Unterschiedliches: Links bezeichnen sie eine Variable und rechts einen Wert. Um dies unterscheiden zu können haben wir weiter oben die Begriffe *l–Wert* (linksseitiger Wert) und *r–Wert* (rechtsseitiger Wert) eingeführt. Manche Ausdrücke haben nur einen r–Wert (z.B. `i+2`) andere dagegen haben einen l– und einen r–Wert. Das alles gilt natürlich auch für indizierte Ausdrücke. Ein indizierter Ausdruck wie etwa a[i] hat einen l–Wert (es ist eine Variable ...) und einen r–Wert (... die einen Wert hat).

Der *Index* muss natürlich – so wie die rechte Seite einer Zuweisung – ein r–Wert sein. `i` in `a[i]` wird darum genauso berechnet wie rechts vom Zuweisungszeichen, etwa in `...= i;`

Felder sind keine normalen Variablen. Sie haben “als Ganzes” keinen Wert. Die Konsequenz dieser Tatsache ist, dass ihnen nicht mit einer (einzigen) Zuweisung ein Wert zugewiesen werden kann und dass sie nicht (mit einer Vergleichsoperation) verglichen werden können. Diese Eigenschaften der Felder stecken in ihren “besonderen” l– und r–Werten:

- Ein Feld hat *keinen l–Wert*
- Ein Feld hat *keinen normalen r–Wert*. Felder haben zwar einen r–Wert, aber das ist *nicht* der Inhalt der Feldvariablen.[19]

Da sie ohne einen l–Wert sind, kann es keine Zuweisungen an Felder geben. Der r–Wert eines Feldes ist nicht dessen Inhalt, darum kann man Felder nicht als ganzes vergleichen:

```
int a[10];
int b[10];
...
a = ...         // =  ist VERBOTEN: Feld a hat keinen l-Wert
...
if (a == b) {   // == ist erlaubt, aber KEIN Vergleich
                // des Inhalts von a und b:
  ...           // der r-Wert von Feldvariablen ist NICHT deren Inhalt
}
```

Der Zuweisungsoperator = ist also auf Feldern gar nicht und der Vergleichsoperator == nicht so definiert, dass ganze Felder verglichen werden..

Beispiele

Wir wollen einige Beispiele für Zuweisungen und Ausdrücke mit indizierten Variablen (Variablen mit einem Index) betrachten:

Angenommen `i` habe den Wert 5 und `a[`i`]`, für i = 0 ... 9, den Wert $(i + 1)\ modulo\ 10$. Also:

i: | 5 |

a:

1	2	3	4	5	6	7	8	9	0

[19] Der r–Wert eines Feldes ist ein Zeiger auf seine erste Komponente.

Jetzt werden einige Anweisungen nacheinander ausgeführt. Die Variablen `a` und `i` durchlaufen dann die entsprechenden Werte (wir geben jeweils zuerst die Anweisung und danach die Variablenbelegung an, die diese Anweisung erzeugt):

1. Ausgangssituation:

   i: | 5 |

   a:

1	2	3	4	5	6	7	8	9	0

2. `i = a[1];`

   i: | 2 |

   a:

1	2	3	4	5	6	7	8	9	0

3. `i = a[i];`

   i: | 3 |

   a:

1	2	3	4	5	6	7	8	9	0

4. `i = a[i]+1;`

   i: | 5 |

   a:

1	2	3	4	5	6	7	8	9	0

5. `i = a[i+1];`

   i: | 7 |

   a:

1	2	3	4	5	6	7	8	9	0

6. `a[i] = a[a[i]+1];`

   i: | 7 |

   a:

1	2	3	4	5	6	7	0	9	0

7. `a[a[i]+1] = a[a[i+1]-1];`

   i: | 7 |

   a:

1	9	3	4	5	6	7	0	9	0

8. `i = a[a[i]] + a[a[i]+7] + a[a[i-1]];`

   i: | 1 |

   a:

1	9	3	4	5	6	7	0	9	0

(Bitte nachvollziehen!)

Beispiel: Fakultät

Mit folgendem Programm kann die Fakultätsfunktion tabelliert werden:

```
...
int main () {
  const int n = 10;
  int f[n], // f[i] soll i! enthalten
      i;

  f[0] = 1; i = 1;
  while (i<n) {
    f[i] = f[i-1]*i;
    i = i+1;
  }
}
```

Das Programm berechnet in den Variablen `f[`i`]` für ansteigendes i den Wert $i!$. Die While–Schleife kann durch eine äquivalente For–Schleife ersetzt werden:

```
...
  f[0] = 1;
  for (int i = 1; i < n; ++i)
    f[i] = f[i-1]*i;
...
```

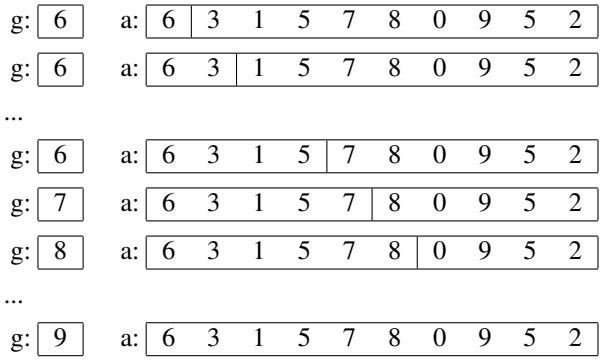
6.3 Suche in einem Feld, Feldinitialisierung, Programmtest

Suche in einem Feld

Ein bestimmter Wert s kann in einem Feld recht einfach gesucht werden. Man vergleicht einfach jedes Feldelement mit s. Nehmen wir an, in Feld `a` der Größe `n` soll der Wert s (in `s`) gesucht werden. Bei dieser Suche interessieren wir uns für den Index. D.h. es soll jeder Index i ausgegeben werden für den `a[`i`]` = s ist. Das Feld wird mit der Standard–For–Schleife durchlaufen:

```
for (int i = 0; i < n; ++i) // Fuer jeden Index i
  if (a[i] == s)            // Vergleiche a[i] mit s
    cout << i << endl;      // und gib i eventuell aus
```

Im folgenden Beispiel soll das größte Element in einem Feld `a[n]` gesucht werden. Die Strategie zur Suche besteht darin, dass ein immer größerer Bereich des Feldes untersucht wird und in einer Variablen `g` stets der bisher gefundene größte Wert zu finden ist:



Mit einer Schleife

```
int g = a[0];
for (int i=1; i<10; ++i) // g enthaelt den bisher groessten Wert
  if ( a[i] > g )
    g = a[i];
```

kann diese Strategie realisiert werden. Die Laufvariable `i` gibt dabei die Grenze zwischen dem bereits untersuchten und dem noch zu untersuchenden Bereich von `a` an. `a[0...i-1]` ist untersucht. `a[i...n]` muss noch untersucht werden.

Da aber nicht der Wert, sondern der Index des größten gesucht ist, ändern wir die Schleife zu:

```
int gi = 0;
for (int i=1; i<10; ++i) // a[gi] enthaelt den bisher groessten Wert
  if (a[i] > a[gi])
    gi = i;
```

als Gesamtprogramm:

```
#include <iostream>

using namespace std;

int main () {
  const int n = 10;
  int    a[n] = {0,9,1,6,4,6,8,2,7,6};

  int gi = 0;
  for (int i=1; i<n; ++i)
```

```
    if (a[i] > a[gi])
      gi = i;

  cout << "Der Index des groessten Wertes " << a[gi] << " ist " << gi << endl;
}
```

Feldinitialisierung

Initialisierungen wie etwa

```
int a[10] = {0,9,1,6,4,6,8,2,7,6};
```

sind die einzigen Stellen, an denen eine Zuweisung an ein Feld erlaubt ist.

Bei der Initialisierung wird in geschweiften Klammern eine Folge von Werten angegeben. Die Folge sollte der Zahl der Feldelemente entsprechen. Sie darf nicht größer sein (Fehler, oder Warnung des Compilers) als die Zahl der Feldelemente. Wenn sie kleiner ist, wird der Rest mit 0 aufgefüllt:

```
float a[4] = {-2.0, 0.5};
```

ist das Gleiche wie

```
float a[4] = {-2.0, 0.5, 0.0, 0.0};
```

Programmtest

Ein Test ist die experimentelle Kontrolle der Korrektheit eines Programms. Dazu wird die Programmausgabe unter verschiedenen Bedingungen (Eingaben) untersucht. Tests werden in Form von Testfällen formuliert (siehe Kapitel 3). Ein *Testfall ist immer nur eine "Stichprobe"*, trotzdem kann man systematisch testen. Man testet das Programm mit Testfällen, die "repräsentativ" für alle möglichen Bedingungen sind.

In unserem Beispiel kann der Plan für das systematische Testen so aussehen, dass die Testfälle folgende Situationen abdecken:

1. In unsortiertem Feld suchen
2. In sortiertem Feld suchen
3. Suchen mit mit kleinstem / größtem Wert in der Mitte
4. Suchen mit kleinstem / größtem Wert im ersten/letzten Feldelement

Wenn das Programm die entsprechenden Testfälle bestanden hat, kann man einigermaßen sicher sein, dass es korrekt ist.

6.4 Sortieren, Schleifeninvariante

Eine Sortierstrategie

Ein Feld von Zeichen soll aufsteigend sortiert werden. Wir verwenden die gleiche Strategie wie bei der Suche. Ein Anfangsbereich des Feldes ist sortiert und enthält alle kleinsten Elemente des Feldes. Dieser Bereich wird systematisch vergrößert. Am Anfang enthält der Bereich kein Element und am Ende alle Feldelemente. Die Vergrößerung des Bereichs besteht darin, dass das kleinste Element im unbearbeiteten Teil gesucht und mit dem ersten Element im unbearbeiteten Teil vertauscht wird (der sortierte Bereich vergrößert sich somit um dieses Element):

Anfang:	a:	6	3	1	5	7	8	0	9	5	2
0 und 6 werden vertauscht:	a:	**6**	3	1	5	7	8	**0**	9	5	2
1 und 3 werden vertauscht:	a:	0	**3**	**1**	5	7	8	6	9	5	2
2 und 3 werden vertauscht:	a:	0	1	**3**	5	7	8	6	9	5	**2**
3 und 5 werden vertauscht:	a:	0	1	2	**5**	7	8	6	9	5	**3**
5 und 7 werden vertauscht:	a:	0	1	2	3	**7**	8	6	9	**5**	5
5 und 8 werden vertauscht:	a:	0	1	2	3	5	**8**	6	9	7	**5**

nichts wird vertauscht:	a:	0	1	2	3	5	5	**6**	9	7	8
7 und 9 werden vertauscht:	a:	0	1	2	3	5	5	6	**9**	**7**	8
8 und 9 werden vertauscht:	a:	0	1	2	3	5	5	6	7	**9**	**8**
nichts wird vertauscht:	a:	0	1	2	3	5	5	6	7	8	**9**
fertig:	a:	0	1	2	3	5	5	6	7	8	9

Der Schleifenkörper

Für die Vergrößerung des bearbeiteten Teils des Feldes ist eine Schleife verantwortlich. Jeder Durchlauf (Betreten des Schleifenkörpers) hat eine Aufgabe und operiert unter bestimmten Voraussetzungen. Die Aufgabe besteht auch darin die Voraussetzungen für den nächsten Durchlauf zu schaffen:

- Voraussetzung: Feld ist bis `i` sortiert: `a[0..i-1]` ist sortiert und enthält die `i`–kleinsten Elemente des Feldes.
- Teilaufgabe eines Schleifendurchlaufs: `i` um eins erhöhen und dabei die Eigenschaft erhalten, dass `a[0..i-1]` sortiert ist und die `i`–kleinsten Elemente enthält.
- Lösung der Teilaufgabe: Im unsortierten Teil das kleinste Element suchen und mit dem ersten im unsortierten Bereich (`a[i]`) vertauschen!

Entwurf der Schleife:

```
for (int i=0; i<n; i++) {

  //a[0..i-1] ist aufsteigend sortiert und enthaelt kein
  // Element das groesser ist als irgend eines aus a[i..n-1]

  m =  Index des kleinsten in a[i..n-1]
  tausche Inhalt von a[m] und a[i]
}
```

Die Suche nach dem Kleinsten kann mit dem Algorithmus aus dem letzten Abschnitt erfolgen. Das Vertauschen ist trivial:

```
t    = a[i];
a[i] = a[m];
a[m] = t;
```

Das Sortierprogramm

Das Programm insgesamt kann aus diesen Komponenten leicht zusammengesetzt werden:

```
...
int main () {
  const int n = 10;
  int a[n] = {0,9,1,6,4,6,8,2,7,6},
      m, t;

  for (int i=0; i<n; i++) {
    //INVARIANTE:
    //a[0..i-1] ist aufsteigend sortiert und enthaelt kein
    //Element das groesser ist als eines aus a[i..n-1]

    //suche Index des kleinsten in a[i..n-1]:
    m = i;
    for (int j=i+1; j<n; ++j)
      if (a[j]<a[m])
        m = j;
    //m enthaelt Index des kleinsten in a[i..n-1]
```

```
      //tausche Inhalt von a[i] und a[m]:
      t    = a[i];
      a[i] = a[m];
      a[m] = t;
    }

    // a[0..i-1] ist aufsteigend sortiert, i = n,
    // also: a ist aufsteigend sortiert !
  }
```

Schleifeninvariante

Die *Schleifeninvariante* (siehe Kapitel 3) ist eine Aussage über die Voraussetzungen unter denen jeder Durchlauf der Schleife startet und die dieser Durchlauf darum auch wieder herstellen muss – nach diesem, kommt ja gleich der nächste Durchlauf. Sie beschreibt die Voraussetzung und gleichzeitig die Wirkung des Schleifenkörpers. Das ist etwas, das bei aller durch die Schleife erzeugten Veränderung unverändert ("invariant") bleibt: die Essenz der Schleife.

In unserem Sortier–Beispiel hier ist die Invariante die Tatsache, dass `a[0..i-1]` – bei sich erhöhendem `i` – sortiert bleibt und die `i`–kleinsten Werte des Feldes enthält. Bei der Suche oben war die Invariante, dass `gi` – bei sich erhöhendem `i` – den Index des größten in `a[0..i-1]` enthält:

```
int gi = 0;
for (int i=1; i<n; ++i) // INV: a[gi] ist der groesste in a[0..i-1]
  if (a[i] > a[gi])
    gi = i;
```

6.5 Zweidimensionale Strukturen

Vektoren und Matrizen

Ein Vektor ist eine Folge von Zahlen. Vektoren können durch Felder oder Verbunde dargestellt werden. Üblicherweise nimmt man Felder, da bei ihnen Indexberechnungen möglich sind.

Matrizen sind zweidimensionale Strukturen. Als solche können sie nicht im Speicher abgelegt werden. Der Speicher ist eine Folge von Bytes und damit eindimensional. Zweidimensionale Strukturen können aber immer auf eindimensionale abgebildet werden.

Eine Matrix kann auf zwei Arten eindimensional dargestellt, also als Kombination von Vektoren verstanden werden:

- Entweder sieht man sie als Folge von Spalten, oder
- als Folge von Zeilen.

In beiden Fällen ist die Matrix ein Vektor von Vektoren. Die Matrix

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix}$$

wird also entweder als

- $((a_{11}, a_{12}, a_{13}), (a_{21}, a_{22}, a_{23}))$ (Zeilenform, Standard), oder als
- $((a_{11}, a_{21}), (a_{12}, a_{22}), (a_{13}, a_{23}))$ (Spaltenform, nicht üblich)

interpretiert. Die erste Variante ist die Zeilenform (Matrix = Vektor von Zeilen–Vektoren), die zweite ist die Spaltenform (Matrix = Vektor von Spalten–Vektoren).

Standarddarstellung einer Matrix

Für die Sprache C++, und fast alle anderen Programmiersprachen, ist festgelgt, dass Matrizen (zweidimensionale Strukturen) in der *Zeilenform* gespeichert werden, also als Vektor von Zeilen–Vektoren. Jede Matrix ist damit ein Feld von Zeilen und jede Zeile ein Feld von Werten. Der Typ einer Matrix kann entsprechend definiert werden.

Beispielsweise kann die 2×3 Matrix von oben wie folgt definiert werden:

```
typedef float Zeile[3];     // Zeile = Feld von Werten
typedef Zeile Matrix[2];    // Matrix = Feld von Zeilen
Matrix a;
```

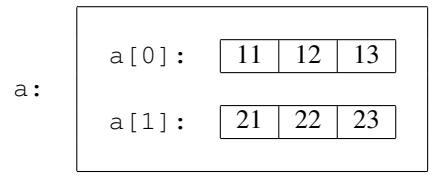
Eine `Matrix` besteht aus zwei Komponenten vom Typ `Zeile`. Jede `Zeile` besteht aus drei `float`s. `a` vom Typ `Matrix` ist ein zweidimensionales Feld. Also ein Feld, das aus Feldern besteht. Im folgenden gehen wir stets von einer Zeilendarstellung aus.

Elementzugriff

Der Zugriff auf das Matrixelement $a_{i,j}$ wird (bei Zeilendarstellung !) zu `a[i][j]`: Element `j` von Element `i` von `a`. In `a[i]` – einer Variablen vom Typ `Zeile` – wird auf das Element Nr. `j` zugegriffen. (Man beachte immer, dass in C++ *jeder Indexbereich mit Null beginnt* !) Die Matrix

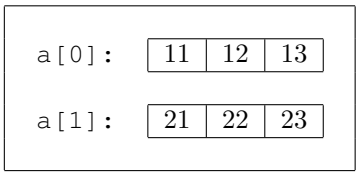
$$\begin{pmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \end{pmatrix}$$

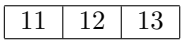
wird mit den Deklarationen von oben folgendermaßen in `a` gespeichert:



`a[i][j]` ist also als Element Nr `j` in der Zeile Nr. `i` zu interpretieren. `a[0][1]` beispielsweise ist Element Nr. `1` von `a[0]`:

`a[0][1]`





$= 12$

Beispiel: Determinante berechnen

Im folgenden Beispiel berechnen wir die Determinante einer 2×2 Matrix:

```
#include <iostream>

using namespace std;

typedef float Zeile[2];
typedef Zeile Matrix[2];

int main () {
  Matrix m;

  //einlesen:
```

```
  for (int i=0; i<2; ++i)
    for (int j = 0; j<2; ++j)
      cin >> m[i][j];

  //Determinante bestimmen
  cout << m[0][0]*m[1][1]
          - m[0][1] * m[1][0]
       << endl;;
}
```

Anonyme Typen für zweidimensionale Felder

Felder können ohne vorherige Definition eines Typs angelegt werden. Der Typ ist dann anonym. Das eindimensionale Feld `x` kann beispielsweise mit einem anonymen Typ definiert werden:

```
int x[10];
```

Das Gleiche gilt auch für zweidimensionale Felder. Statt

```
typedef float Zeile[3];
typedef Zeile Matrix[2];
Matrix a;
```

schreibt man kürzer:

```
typedef float Zeile[3];
Zeile a[2];
```

Damit wird `a` als Variable (mit namenlosem Typ) definiert, die aus zwei Komponenten vom Typ Zeile besteht. Das Spiel kann man weitertreiben und dem Typ `Zeile` seinen Namen nehmen:

```
float a[2][3];
```

Jetzt wird `a` als Variable (mit namenlosem Typ) definiert, die aus zwei (2) Komponenten besteht (die ebenfalls namenlose Typ haben). Jede Komponente besteht wiederum selbst aus drei (3) Elementen vom Typ `float`. Man leist die Definition also von `a` weg: `a` hat 2 Komponenten, jede Komponente hat 3 Komponenten und diese haben den Typ `float`.

```
//Explizte Typdefinitionen   // Matrix anonym          // Zeile und Matrix
                                                      // anonym
typedef float Zeile[3];      typedef float Zeile[3];
typedef Zeile Matrix[2];
Matrix a;                    Zeile   a[2];             float  a[2][3];
```

Generell wird empfohlen mit expliziten Typdefinitionen zu arbeiten. Der leicht erhöhte Schreibaufwand wird in der Regel durch die verbesserte Lesbarkeit der Programme belohnt.

Beispiel: Matrix–Multiplikation

Bekanntlich kann man zwei Matrizen $\mathbf{A}$ und $\mathbf{B}$ miteinander multiplizieren, wenn die Spaltenzahl der ersten gleich der Zeilenzahl der zweiten ist. ($\mathbf{A}$ ist eine $M \times K$ und $\mathbf{B}$ eine $K \times N$ Matrix.) Ein Element $c_{i,j}$ der Ergebnismatrix ist das Produkt einer Zeile und einer Spalte (Mathematische Notation: Indizes starten mit 1):

$c_{i,j} = \mathbf{A}_i * \mathbf{B}^j = \sum_{k=1}^{K} a_{i,k} * b_{k,j}$

Die Multiplikation einer 2×3 und mit einer 3×2 Matrix ergibt eine 2×2 Matrix (C++ Notation: Indizes starten mit 0):

```
int  main () {
  int a[2][3] = {{1,2,3},{2,4,6}};
  int b[3][2] = {{0,1},{1,2},{2,3}};
  int c[2][2];

  //c=a*b:

  for (int i=0; i<2; ++i)
    for (int j=0; j<2; ++j){
      int s = 0;
```

```
        for (int k=0; k<3; ++k)
          s = s + a[i][k] * b[k][j];
        c[i][j] = s;
      }
  }
```

6.6 Beispiel: Pascalsches Dreieck

Das Pascalsche Dreieck

Das Pascalsche Dreieck hat die Form:

```
          1
        1   1
      1   2   1
    1   3   3   1
  1   4   6   4   1
        ...
```

Das Bildungsgesetz ist wohl offensichtlich. Man kann die Zeilen auch linksbündig hinschreiben.

```
1
1  1
1  2  1
1  3  3  1
1  4  6  4  1
...
```

Dabei erhält man eine Folge von Zeilen, deren Länge stetig wächst. Man kann natürlich die “fehlenden” Werte als Null annehmen um eine Folge von Zeilen fester Länge zu erhalten:

```
1  0  0  0  0  0  0  0  0
1  1  0  0  0  0  0  0  0
1  2  1  0  0  0  0  0  0
1  3  3  1  0  0  0  0  0
1  4  6  4  1  0  0  0  0
...
```

Das Dreieck als Inhalt einer Matrix

Das Pascalsche Dreieck kann leicht als Inhalt eines zweidimensionalen Feldes berechnet werden. Jede Zeile wird durch einfache Additionen aus der darüberliegenden berechnet. Die erste Zeile benötigt natürlich eine Sonderbehandlung, sie hat ja keinen Vorgänger.

Innerhalb einer Zeile berechnet sich jedes Element als Summe der Elemente direkt und links über ihm. Auch hier benötigt das erste Element eine Sonderbehandlung: links über ihm gibt es ja nichts.

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

  // erste Zeile setzen
  x[0][0] = 1; // Erstes Element, Sonderbehandlung
  for (int j = 1; j < 10; ++j)
    x[0][j] = 0;

  // weitere Zeilen aus ihren Vorgaengern berechnen
  for (int i = 1; i < 10; ++i) {
    x[i][0] = 1; // Erstes Element, Sonderbehandlung
    for (int j = 1; j < 10; ++j)
      x[i][j] = x[i-1][j-1] + x[i-1][j];
  }

  //Matrix ausgeben:
  for (int i = 0; i < 10; ++i) {
```

```
    for (int j = 0; j < 10; ++j) {
      cout.width(4);  //4 Zeichen pro Element ausgeben
      cout << x[i][j];
    }
    cout << endl;
  }

}
```

Das Dreieck zeilenweise als Feldinhalt

Da jede Zeile nur von der direkt darüber liegenden abhängt, benötigt man eigentlich kein zweidimensionales Feld. Die Zeilen können als wechselnde Wertebelegungen eines eindimensionalen Feldes `a` in einer Schleife berechnet werden:

```
const int L = ...;
int a[L];

// erste Zeile setzen
a[0]=1;
for (int j=1; j<L; j++)
    a[j]=0;

// weitere Zeilen berechnen
for (int i=1; i<L; i++) {
    // INV: a enthaelt Zeile Nr. i-1
    ?? neue Werte von a (Zeile Nr. i) aus den alten berechnen ??
    // a enthaelt Zeile Nr. i
}
```

Die Berechnung einer neuen Zeile a_i aus einer alten a_{i-1} scheint nicht sehr schwierig zu sein: der neue Wert an Position j ist offensichtlich die Summe aus seinem Vorgänger an gleicher Position und dem alten linken Nachbarn:

$a_{i,j} = a_{i-1,j-1} + a_{i-1,j}$

Z.B.: alte Belegung von a (Zeile Nummer 3 des Pascalschen Dreiecks: a_3):

a:	$a_{3,0} = 1$	$a_{3,1} = 3$	$a_{3,2} = 3$	$a_{3,3} = 1$

Daraus wird die neue Belegung von a konstruiert (Zeile Nummer 4 des Pascalschen Dreiecks: a_4):

a:	$a_{4,0} = 1$	$a_{4,1} = 4$	$a_{4,2} = 6$	$a_{4,3} = 4$	$a_{4,4} = 1$

Da die Folge $a_{i,0}, a_{i,1}, \ldots$ der aktuelle Wert von `a` ist, liegt es nahe, den Schleifenkörper (Übergang von Zeile $i-1$ zu Zeile i) wie folgt zu formulieren:

```
for (int i=1; i<L; i++) {
    // INV: a enthaelt Zeile Nr. i-1
    a[0]=1;
    for (int j=1; j<i; j++)
        a[j] = a[j-1] + a[j];
}
```

Das Problem dabei ist, dass der alte Wert von `a[j-1]` für die Berechnung des neuen Werts von `a[j]` gebraucht wird, aber "gerade eben" – bei der Berechnung des neuen `a[j-1]` – überschrieben wurde.

Für dieses Problem gibt es eine einfache Lösung: Das Feld wird nicht von links nach rechts, sondern von rechts nach links mit neuen Werten gefüllt:

```
for (int i=1; i<L; i++) {
    // INV: a enthält Zeile Nr. i-1
    a[0]=1;
    for (int j=i; j>0; j--)
        // INV: a[j+1..i] is neu, a[1..j] ist alt
        a[j] = a[j-1] + a[j];
    // a enthält Zeile Nr. i

    // Ausgabe der Zeile i
```

```
    for (int j=0; j<L; j++) { cout.width(4); cout << a [j]; } cout << endl;
}
```

6.7 Beispiel: Gauss–Elimination

Das Gauss–Verfahren

Die Gauss–Elimination (Gauss–Verfahren) ist eine Methode zur Lösung eines linearen Gleichungssystems $\mathbf{A}\vec{x} = \vec{b}$ (Vergl. Mathematik–Vorlesung):

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \cdots & & & \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \cdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \cdots \\ b_n \end{pmatrix}$$

Auch wenn in C++ jeder Indexbereich mit 0 beginnt, so bleiben wir doch in mathematischen Ausdrücken bei der in der Mathematik üblichen 1 als erstem Index. $a_{1,1}$ wird natürlich in `a[0][0]` gespeichert.

Die Matrix $(\mathbf{A}, \vec{b})$:

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} & b_1 \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} & b_2 \\ \cdots & & & & \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} & b_n \end{pmatrix}$$

wird dabei systematisch so transformiert, dass $\mathbf{A}$ in eine obere Dreiecksmatrix überführt wird. Anschließend können die Unbekannten dann "von unten nach oben" bestimmt werden.

Bei der Transformation von $(\mathbf{A}, \vec{b})$ wird im ersten Schritt ein geeignetes (jeweils unterschiedliches) Vielfaches der ersten Gleichung von den anderen abgezogen, derart, dass die Koeffizienten von x_1 in diesen Gleichungen verschwinden. Am Ende dieses Schritts wird also x_1 nur noch in der ersten Gleichung vorkommen. Das ist natürlich nur dann möglich, wenn $a_{1,1} \neq 0$ ist. Dies kann man gegebenenfalls durch Vertauschen der Gleichungen erreichen.

Im zweiten Schritt wird mit Hilfe der zweiten Gleichung x_2 in der dritten und den weiteren Gleichungen eliminiert.

Im dritten Schritt wird mit Hilfe der dritten Gleichung x_3 in der vierten und den weiteren Gleichungen eliminiert. Und so weiter, bis schließlich in der letzten Gleichung nur noch x_n vorkommt. Als Ergebnis dieser Schritte ergibt sich:

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} & b_1 \\ 0 & a_{2,2} & \cdots & a_{2,n} & b_2 \\ \cdots & & & & \\ 0 & 0 & \cdots & a_{n,n} & b_n \end{pmatrix}$$

Ein Algorithmus zur Gauss–Elimination

Führe die Schritte i = 1 .. n durch, in Schritt i:

1. Suche nach einer geeigneten nächsten Gleichung:
   Bestimme ein $r \geq i$ mit $a_{r,i} \neq 0$, falls nicht vorhanden: Abbruch, keine (eindeutige) Lösung möglich.
2. Mache sie zur i–ten Gleichung:
   Tausche die Zeilen a_i und a_r (jetzt ist $a_{i,i} \neq 0$).
3. Eliminiere in den folgenden Gleichungen x_i:
   Führe Schritte j = $(i+1)$ .. n durch:
   a) Setze f = $a_{j,i}/a_{i,i}$
   b) Setze $a_{j,m} = a_{j,m} - f * a_{i,m}$ für m = i .. n

c) Setze $b_j = b_j - f * b_i$

Jetzt ist die obere Dreiecksform erreicht und die Gleichungen können von unten nach oben gelöst werden:

Löse die Gleichungen: Für i = n ... 1

1. setze z = b_i
2. Setze z = z - $x_j * a_{i,j}$ für $j = (i + 1)$ .. n
3. setze x_i = $z/a_{i,i}$

Ein Programm zur Gauss–Elimination

Im folgenden Programm haben wir darauf verzichtet ein $a_{r,i} \neq 0$ zu suchen. Wir gehen einfach davon aus, dass $a_{i,i} \neq 0$ ist. Eine entsprechende Erweiterung des Programms bleibt dem Leser überlassen. Auf eine Angabe der Ein– und Ausgabeoperationen wurde zur Vereinfachung ebenfalls verzichtet.

```
#include <iostream>

using namespace std;

const int N = 3;

typedef float Zeile[N];
typedef Zeile Matrix[N];
typedef float Vektor[N];

Matrix a;
Vektor b;
Vektor x;

int main () {
  float f;

  // Eingabe von a und b ...

  //In Dreiecksform bringen:
  for (int i = 0; i < N; ++i)
    for (int j = i+1; j < N; ++j) {
      f = a[j][i]/a[i][i];
      for (int m = i; m<N; ++m)
        a[j][m] = a[j][m] - f*a[i][m];
      b[j] = b[j] - f*b[i];
    }

  //Rueckrechnung (x-Werte berechnen):
  for (int i = N-1; i>=0; --i) {
    float z = b[i];
    for (int j=i+1; j<N; ++j)
      z = z - x[j]*a[i][j];
    x[i] = z/a[i][i];
  }

  // Ausgabe von x ...
}
```

6.8 Verbunde (struct–Typen)

Verbund

Verbunde (auch structs, Strukturen oder engl. *Records* genannt) sind wie Felder strukturierte Variablen, also Variablen, die aus "Untervariablen" als Komponenten zusammengesetzt sind. Felder bestehen aus Komponenten, die alle den gleichen Typ haben müssen und jeweils mit einem Index in Form eines Int–Wert angesprochen werden. *Verbunde* sind zusammengesetzte Variablen deren Komponenten

- *Namen* haben und
- von *unterschiedlichem Typ* sein können.

Statt, wie bei Feldern mit einem Index, wird auf die Komponente eines Verbunds also mit einem Namen zugegriffen und im Gegensatz zu den Komponenten eines Feldes dürfen die Komponenten eines Verbunds unterschiedliche Typen haben.[20]

Als Beispiel definieren wir einen Vektor mit zwei Elementen einmal als Feld und einmal als Verbund:

```
// Vektoren als Felder           // Vektoren als Verbunde
//                               //
typedef   int VektorF[2];        struct   VektorV { int x; int y; };  // Typ
VektorF   v1;                    VektorV  v2;                         // Variable
v1[0] = 1;                       v2.x = 1;                            // Zugriff
v1[1] = 2;                       v2.y = 2;
```

Das Feld `v1` und der Verbund `v2` haben jeweils zwei Int–Variablen als Komponenten. Der einzige Unterschied zwischen beiden ist die Art des Zugriffs: mit einem Index bei `v1`, mit den Namen `x` und `y` bei `v2`.(Siehe Abbildung 18).

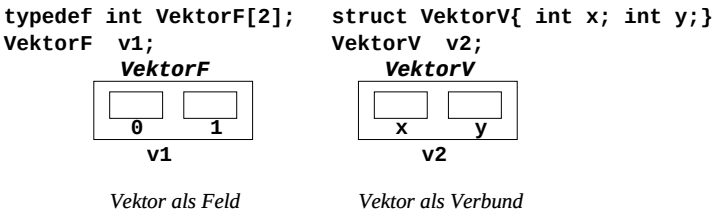


Abbildung 18: Feld und Verbund

Verbunde benutzt man, wenn man beim Zugriff Namen gegenüber Indizes bevorzugt, oder wenn nicht alle Komponenten vom selben Typ sind. Im folgenden Beispiel modellieren wir "Hunde" als Verbunde. Ein Hund hat einen Namen und eine Rasse. Name und Rasse sind unterschiedliche Typen, Hunde als Felder darzustellen ist darum nicht möglich:

```
#include <string>

using namespace std;

int main () {
  enum Hunderasse {Dalmatiner, Terrier, Dackel, Boxer};  // Typ Hunderasse

  struct Hund {            // Definition eines Verbund-Typs Hund
    string     name;       // Hunde haben Namen: Werte vom Typ string
    Hunderasse rasse;      // Hunde haben eine Rasse: Werte vom Typ Hunderasse
  };

  Hund w,                          // Definition einer Variablen
       hektor = {"Max", Terrier}; // Variable mit Initialisierung

  w.name  = "Waldi";               // Zugriff auf Komponenten.
  w.rasse = Dackel;
}
```

Hier wird der Typ `Hund` und zwei Variablen `w` und `hektor` von diesem Typ definiert. `hektor` erhält seinen Wert in einer Initialisierung und `w` mit komponentenweisen Zuweisungen.

Definition eines Verbundtyps

Verbundtypen werden mit Definitionen der Form:

[20] Die Komponenten eines Verbundes werden auch oft *Feld* genannt! Hier wird allerdings die Bezeichnung "(Verbund–) Komponente" oder "–Element" bevorzugt.

struct $< Name >$ { $< Typ - 1 >< Name - 1 >; ... < Typ - n >< Name - n >;$ };

eingeführt.

- $< Name >$ ist der Name des neu definierten Verbund–Typs (und nicht der einer Variablen)
- $< Typ - 1 >$ bis $< Typ - n >$ sind die Typen der Komponenten, auf die mit
- $< Name - 1 >$ bis $< Name - n >$ zugegriffen werden kann.

Das ist also eine Typdefinition. Zum Vergleich mit Feldern: Die Definition

<Typ> a[*<Größe>*];

ist eine Variablendefinition! Zur Definition eines Feld–*Typs* muss typedef verwendet werden.

Verbundinitialisierung

Genau wie ein Feld kann auch eine Verbundvariable bei ihrer Definition gleich mit einem Wert initialisiert werden:

```
Hund hektor = { "Max", Terrier };
```

Wie bei Feldern kann die Initialisierung auch unvollständig sein. Die fehlenden Werte werden dann mit 0 ergänzt. Es wird generell nicht empfohlen diese Eigenschaft in einem Programm auszunutzen.

Punkt–Operator: Zugriff auf eine Verbundkomponente

Auf die Komponenten einer Verbundvariablen wird mit dem *Punkt–Operator* zugegriffen. Mit

$< Variable >.< Name >$

wird auf die Komponente mit dem Namen $< Name >$ innerhalb der (Verbund–) Variablen $< Variable >$ zugegriffen.

Mit dem folgenden Beispiel vergleichen wir den Komponentenzugriff bei Feldern und Verbunden. Es wird ein Feld va und ein Verbund vs benutzt, um jeweils einen Vektor mit zwei Komponenten zu modellieren:

```
...
int main () {
  struct VektorV {
    int  x;
    int  y;
  };

  typedef int VektorF[2];

  VektorV  vs = {1, 2}; // Verbund-Initialisierung
  VektorF  va = {1, 2}; // Feld-Initialisierung

  vs.x = 3;             // Zugriff ueber feste Namen
  vs.y = 3;             // Schleife nicht moeglich

  for (int i = 0; i < 2; ++i)
    va[i] = 3;          // Zugriff ueber berechenbaren Index

  // Zugriff auf vs in einer Schleife nicht moeglich !
}
```

Der Zugriff auf die Komponenten von vs ist mit vs.x (x–Komponente von vs) und vs.y (y–Komponente von vs) "informativer" als der Feldzugriff. Dafür kann der Index dynamisch berechnet werden. Felder können darum mit Schleifen durchlaufen werden, Verbunde nicht.

Felder sind flexibler beim Zugriff auf die Elemente. Verbundelemente können beliebige Typen haben. Sie sind flexibler in der Struktur.

Felder und Verbunde

Felder und Verbunde können alternativ und auch gemeinsam genutzt werden, um bestimmte Daten im Programm zu modellieren. Die Beschäftigten einer Firma kann man etwa folgendermaßen darstellen:

```
// Arten von Angestellten:
enum Gruppe {
  Sekretariat, Programmierer, Manager, Boss
};

// Typdefinition: Daten zu jede(r/m) Angestellten:
struct Person {
  string   vorname;
  string   zuname;
  int      gehalt;
  Gruppe   pos;
};

// Variablendefinition:  Daten zu zwei bestimmten Angestellten:
Person  p = {"Peter", "Hacker", 60000, Programmierer},
        q = {"Petra", "Meier", 120000, Manager};

// Ein Feld von Verbunden: Daten zu vielen Angestellten:
Person  angestellte[20];

...
//Gehalt anpassen:
for (int i=0; i<20; ++i) // fuer jede(n) Angestellte(n)
  switch (angestellte[i].gruppe) {
  case Boss          :
    angestellte[i].gehalt = angestellte[i].gehalt * 3;
    break;
  case Manager       :
    angestellte[i].gehalt = angestellte[i].gehalt * 2;
    break;
  case Programmierer :
    angestellte[i].gehalt = angestellte[i].gehalt * 2;
    break;
  case Sekretariat   :
    ++angestellte[i].gehalt;
    break;
  }
...
```

Mit der Definition

```
Person angestellte[20];
```

wird im Speicher Platz für 20 Verbunde vom Typ `Person` geschaffen. Die Variable `angestellte` hat einen anonymen (namenlosen) Typ. Den Typ "Feld von 20 Verbunden vom Typ `Person`". Mit

```
typedef Person A[20];
A angestellte;
```

hätte man ihm einen Namen (`A`) geben und diesen dann in der Variablendefinition verwenden können. Der Ausdruck

```
angestellte[10]
```

bezeichnet den 11–ten Verbund (Felder beginnen bei 0) im Feld `angestellte`, also eine Variable vom Typ `Person`.

```
angestellte[10].gehalt
```

schließlich ist eine `int`–Variable.

Beispiel

Wir betrachten ein weiteres Beispiel zu Feldern und Verbunden. Hier soll ein Dorf mit 10 Familien modelliert werden. Wenn eine Familie durch den Typ `Familie` dargestellt wird, dann ist das Dorf mit 10 Familien ein Feld, dargestellt durch eine Variable `dorf` mit der Definition:

```
Familie dorf[10];
```

Eine Familie besteht aus Mutter, Vater und den Kindern. Alle sind Personen, die durch den Typ `Person` modelliert

werden. Eine Familie insgesamt kann dann definiert werden als:

```
struct Familie {
   Person   mutter;
   Person   vater;
   int      anzKinder;  // wieviel Kinder gibt es in der Familie
   Person   kind[10];   // Platz fuer maximal 10 Personen
};
```

Eine Person hat einen Namen, einen Vornamen ein Geschlecht und ein Geburtsdatum:

```
typedef int Tag;
enum Monat {Jan, Feb, Mar, Apr, Mai, Jun,
            Jul, Aug, Sep, Okt, Nov, Dez};
typedef int Jahr;
struct Datum {
   Tag      tag;
   Monat    monat;
   Jahr     jahr;
};
enum Geschlecht {weiblich, maennlich};

struct Person {
   string     name;
   string     vorname;
   Datum      geburtsDatum;
   Geschlecht geschlecht;
};
```

Insgesamt wird das ganze Dorf durch folgende Definitionen im Programm modelliert:

```
#include <string>

typedef int Tag;
enum Monat {Jan, Feb, Mar, Arp, Mai, Jun,
            Jul, Aug, Sept, Okt, Nov, Dez};
typedef int Jahr;
struct Datum {
   Tag      tag;
   Monat    monat;
   Jahr     jahr;
};
enum Geschlecht {weiblich, maennlich};
struct Person {             // Definition von Person
   string     name;
   string     vorname;
   Datum      geburtsDatum;
   Geschlecht geschlecht;
};
struct Familie {            // Definition von Familie
   Person   mutter;
   Person   vater;
   int      anzKinder;
   Person   kind[10];
};
Familie dorf[10];           // Definition von dorf
```

Solche Definitionen schreibt man am besten “von unten nach oben”, also von der benötigten Definition (`Familie`), zu den Hilfsdefinitionen (`Person`, `Geschlecht` ...). Da in C++ jeder verwendete Name vorher im Text deklariert sein muss, ergibt sich die textuelle Reihenfolge bei der die Hilfsdefinitionen vor den benötigten Definitionen auftauchen. Die textuelle Reihenfolge muss aber nicht die Aufschreibreihenfolge sein.

Wollen wir in unserem Dorf feststellen, ob die Mutter der Familie Nr. 0 am gleichen Tag geboren ist, wie Kind Nr. 1 von Familie Nr. 3, dann kann das mit folgender Anweisung geschehen:

```
if (dorf[3].anzKinder>=2
```

```
     && dorf[0].mutter.geburtsDatum.tag
        == dorf[3].kind[1].geburtsDatum.tag
     && dorf[0].mutter.geburtsDatum.monat
        == dorf[3].kind[1].geburtsDatum.monat
     && dorf[0].mutter.geburtsDatum.jahr
        == dorf[3].kind[1].geburtsDatum.jahr
    ) {
   cout << "Ja !" << endl;
   cout << dorf[0].mutter.name << " ist am gleichen Tag geboren wie "
        << dorf[3].kind[1].name << endl;
}
```

l–Wert, r–Wert und Verbunde

Im Gegensatz zu Feldern haben Verbunde "richtige" l– und r–Werte. Dass ein Verbund einen r–Wert hat bedeutet, dass sie "als Ganzes" in Ausdrücken vorkommen können. Leider ist der einzige Operator, der auf Verbunden (vor–) definiert ist die Zuweisung:[21]

```
Hund s1 = {"Susi", Dalmatiner};
     s2;

s2 = s1;       // OK:     = ist auf Verbunden definiert
if (s1 == s2)  // FEHLER: == ist auf Verbunden nicht (vor-) definiert!
  ...
}
```

In der Zuweisung `s1 = s2;` wird der l–Wert von `s1` und der r–Wert von `s2` berechnet. Beides funktioniert und liefert die erwarteten Ergebnisse.

Kombinierte Typ– und Variablen–Deklarationen

Die Definition eines Verbund–Typs und einer Variablen von diesem Typ kann in einem Ausdruck kombiniert werden:

```
// NICHT empfohlene Konstrukte:

struct S {
  int a;
  float b;
} s1;   // Typ S und Variable s1 vom Typ S werden gleichzeitig definiert (PFUI)!

struct {  // Anonymer (namenloser) Verbundtyp
  int a;
  float b;
} s3;   // Variable s3 und ein anonymer Typ werden gleichzeitig definiert (PFUI)!
```

Dieser Stil dient der Kompatibilität von C++ zu C und *wird nicht empfohlen!* Typ– und Variablendefinitionen sollten klar getrennt werden.

[21] Die Vergleichsoperation auf Verbunden kann vom Programmierer selbst definiert werden. Bei Felder ist dies nicht möglich, da mit ihnen keine "richtigen" Werte verbunden sind.

6.9 Übungen

Aufgabe 1

1. (Zum Knobeln) Oft macht es keinen Unterschied, ob eine Zuweisung einmal oder zweimal ausgeführt wird. Beispielsweise haben

   `a=5;` und `a=5;a=5;`

   stets die gleiche Wirkung. Gelegentlich gilt das jedoch nicht und $L = R;$ und $L = R; L = R;$ haben unterschiedliche Wirkungen. In der Regel sind

   $L = R;$ und $L = R; L = R;$

   dann äquivalent, wenn R ein Ausdruck ist, dessen Wert durch die Zuweisung $L = R;$ nicht verändert wird. Dies gilt beispielsweise oft dann, wenn – wie im Beispiel oben – R eine Konstante ist, es gibt aber auch andere Beispiele, bei denen R nicht konstant ist.

   Geben Sie ein Beispiel für L und R an, bei dem $L = R;$ und $L = R; L = R;$ eine unterschiedliche Wirkung haben. Erläutern Sie weiterhin allgemein die Bedingungen unter denen $L = R;$ und $L = R; L = R;$ nicht äquivalent sind!

   Gelegentlich sind $L = R;$ und $L = R; L = R;$ selbst dann nicht äquivalent, wenn R eine Konstante ist. Das sind die Fälle, in denen durch die Zuweisung $L = R;$ die linke Seite L verändert wird. Konstruieren Sie ein Beispiel!

2. Es sei: `a[10] = {1,2,3,4,5,6,7,8,9,0};` und `int i = 2;`. Welchen Wert hat `a` nach folgenden Zuweisungen:

   ```
   i = a[i];
   a[i] = i;
   a[(2*a[i]+a[i+1])%10] = a[a[i-1]+a[i+1]]-1;
   ```

3. Welche der folgenden Ausdrücke haben einen l–Wert, welche einen r–Wert, welche haben beides (`int i, j; int a[10]`):

   a) `1`

   b) `i`

   c) `i+j`

   d) `a[i]`

   e) `a[i]+j`

   f) `a[i+j]`

   g) `a[i+1]+j`

4. Wodurch unterscheidet sich der Typ `string` von einem char–Feld?

5. Können ganze Felder zugewiesen oder verglichen werden?

6. Ist folgendes Programmstück korrekt?

   ```
   typedef float A[12];
   int     B[10];
   int     C[13];
   A       a;
   B       b;
   a[10] = 12;
   b[10] = 22;
   C[1]  = 1;
   ```

   Wenn nein, welche Fehler enthält es und wie können sie korrigiert werden?

7. Welche Ausgabe erzeugt folgendes Programm:

```
#include <iostream>
using namespace std;
int main () {
  int a[2] = {1, 2};
  int b[2] = {1, 2};

  if ( a == b)  cout << "Gleich !" << endl;
  else          cout << "Ungleich !" << endl;
}
```

Experimentieren Sie!

8. Ersetzen Sie in folgendem Programm die While-Schleife durch eine äquivalente For–Schleife:

```
int main () {
  int f[10], i;

  f[0] = 1; i = 1;
  while (i<9) {
    f[i] = f[i-1]*i;
    i = i+1;
  }
}
```

Aufgabe 2

Schreiben Sie ein Programm zur Tabellierung der Fibonacci–Zahlen. In Ihrem Programm soll ein Feld `int fib[20];` so mit Werten belegt werden, dass `fib[`i`]` = $fib(i)$. Anschließend sollen die Werte in `fib` ausgegeben werden.

Die Fibonacci–Funktion ist definiert als:

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n-2) + fib(n-1) & n > 1 \end{cases}$$

Aufgabe 3

1. Schreiben Sie ein Programm, das zehn ganze Zahlen einliest und in einem Feld abspeichert. Ihr Programm soll danach alle Duplikate entfernen und dann das von den Duplikaten befreite Feld ausgeben.
2. Schreiben Sie ein Programm, das zehn ganze Zahlen einliest und in einem Feld abspeichert. Danach soll Ihr Programm eine Zahl k eingelesen und den k–größten Wert des Feldes ausgeben. Sollte der k–größte Wert nicht exisitieren, dann soll eine entsprechende Meldung ausgegeben werden.

   Der k–größte Wert eines Feldes ist der größte aller Werte des Feldes, für die es $k-1$ grössere im Feld gibt.

Aufgabe 4

Das bekannte Pascalsche Dreieck ist eine Tabellierung der Funktion

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \begin{cases} 0 & n < k \\ 1 & k = 0 \; oder \; n = k \\ \binom{n-1}{k-1} + \binom{n-1}{k} & 0 < k < n \end{cases}$$

Schreiben Sie ein Programm, das $\binom{n}{k}$ mit Hilfe des Pascalschen Dreiecks berechnet.

Ihr Programm soll zunächst ein 2-dimensionales Feld (N Zeilen, K Spalten) mit den Werten des Pascalschen–Dreiecks füllen.

Dann werden zwei Zahlen $0 \leq n \leq N-1$ und $0 \leq k \leq K-1$ eingelesen, der gesuchte Wert dem Feld entnommen und ausgegeben.

Aufgabe 5

Betrachten Sie folgendes Programmstück:

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

int main () {
  struct S1 {
    int  s_i;
    char s_c[5];
  };
  struct S2 {
    int  s_i;
    char s_c[5];
  };
  S1 a [10];
  S2 b [10];
  S1 s1a = {5, {'a','b','c','d','e'}};
  S1 s1b;
  S2 s2;
  string sss = "s_i";

  for (int i=0; i<10; i++){
    a[i] = s1a;
  }
  for (int i=0; i<10; i++){
    b[i].s_i = a[i].s_i;
    for (int j=0; j<5; j++)
      b[i].s_c[j] = a[i].s_c[j];
  }
}
```

1. Welche Werte haben die Variablen am Ende des Programms? Stellen Sie eine Hypothese auf und überprüfen Sie diese.
2. Wären die beiden Anweisungen

   ```
   s2 = s1a;
   s2.sss = 5;
   ```

   am Ende des Programms erlaubt, oder sollten sie zu einer Fehlermeldung des Compilers führen?

Aufgabe 6

1. Ist ein `enum`–Wert als Feld–Index erlaubt?
2. Kann man mit einem `enum`–Wert ein Verbundelement selektieren:
   `enum E {.. x, ..}; struct S {.. T x; ..}; S s; ... s.x ...`
3. Wird mit:

   a) `int i;`

   b) `typedef int i;`

   c) `struct S {int i; int j;};`

   d) `struct S {int i; int j;}; S s;`

   e) `struct S {int i; int j;}; typedef S s;`

   f) `struct S {int i; int j;}; typedef S s; s s1;`

   g) `enum A {r, g, b};`

h) `int A[10];`

i) `typedef int A[10];`

j) `int a[10];`

k) `typedef int A; A a[10];`

l) `typedef int A; typedef A a[10];`

m) `typedef int A[10]; A a[10];`

n) `enum E {r, g, b}; E A[10];`

o) `enum E {r, g, b} A[10];`

p) `typedef enum E {r, g, b} A[10];`

jeweils eine oder mehrere Variablen, Typen, von beidem etwas oder nichts (weil fehlerhaft) definiert?

Aufgabe 7

Ein eifriger, aber noch nicht weit fortgeschrittener Lehrling des Programmierhandwerks schreibt folgendes Programmstück:

```
...
struct Vektor { float x, y, z; };
Vektor v1, v2;
for ( char i = x; i <= z; ++i ) {
  v1.i = 0.0;
  v2.i = 1.0;
}
...
```

Was sagen sie ihm?

6.10 Lösungshinweise

Aufgabe 1

1. (Zum Knobeln) Gesucht war 1. ein Beispiel für eine Zuweisung $L = R;$, bei der es einen Unterschied macht, ob sie einmal oder zweimal hintereinander ausgeführt wird. Das gilt beispielsweise für die Zuweisung `x = x+1;`, denn:
   `x = x+1;` und
   `x = x+1; x = x+1;`
   sind natürlich nicht äquivalent. (Nicht äquivalent: Es gibt Variablenbelegungen, bei denen die beiden eine unterschiedliche Wirkung haben.)

   Zum 2–ten wurde nach den Bedingungen gefragt unter denen $L = R;$ und $L = R; L = R;$ ganz allgemein eine unterschiedliche Wirkung haben. Das ist dann der Fall, wenn der Wert von R von L abhängt. Beispielsweise hängt der Wert von `x+1` – dem R im Beispiel oben – von `x` – dem L von oben – ab.

   Zum 3–ten sollte eine Zuweisung $L = R;$ mit $L = R; \neq L = R; L = R;$ gefunden werden, bei der R eine Konstante ist. Da der Wert einer Konstanten immer konstant ist, kann in diesem Fall auch der Wert von R – der Konstanten – nicht von L abhängen.

   Die Wirkung einer Zuweisung besteht darin, dass an eine Variable ein Wert zugewiesen wird. Wenn der Wert der zugewiesen wird eine Konstante ist, dann kann der Unterschied zwischen der ersten und der zweiten Zuweisung in $L = R; L = R;$ nur darin liegen, dass der gleiche Wert an jeweils *eine andere Variable* zugewiesen wird. D.h. nicht die rechte Seite R hat einen unterschiedlichen Wert, sondern die linke Seite L. In dem gesuchten Beispiel muss also die Zuweisung $L = R;$ das L verändern. Einen solchen Effekt kann es mit indizierten Variablen geben.

2. Versuchen Sie zunächst theoretisch die Wirkung der Zuweisungen zu bestimmen. Benutzen Sie dann den Debugger, um Ihre Überlegungen zu verifizieren. Konsultieren Sie die Betreuer in den Übungen zur Bedienung des Debuggers.

3. a) `1` r–Wert
   b) `i` r–Wert und l–Wert
   c) `i+j` r–Wert
   d) `a[i]` r–Wert und l–Wert
   e) `a[i]+j` r–Wert
   f) `a[i+j]` r–Wert und l–Wert
   g) `a[i+1]+j` r–Wert

4. Auf die Komponenten eines char–Feldes kann direkt zugegriffen werden. Bei einer Zeichenkette gibt es spezielle Zugriffsoperationen. (`s.at(i);`). Zeichenketten können als ganzes zugewiesen werden, bei Feldern ist das nicht möglich. Insgesamt sind Zeichenketten wesentlich komfortabler aber auch komplexer als Felder.

5. Felder können nicht als ganzes zugewiesen oder verglichen werden.

6. Ist folgendes Programmstück korrekt:

```
typedef float A[12];
int     B[10];
int     C[13];
A       a;
B       b;    // Fehler: B ist KEIN Typ !
a[10] = 12;   // OK
b[10] = 22;   // Fehler: b nicht korrekt definiert,
              //         b[10] waere ausserhalb der zulaessigen
              //         Index-Werten (0..9)
C[1]  = 1;    // OK
```

   Nein! `B` ist kein Typ, `B b;` darum nicht korrekt und die Variable `b` nicht definiert.

7. Die Gleichheitsoperation auf Feldern ist zwar definiert, sie vergleicht aber nicht deren Inhalt, sondern prüft ob sie die selben (*identisch*) sind.

8.
```
f[0] = 1;
for (i = 1; i<9; i++)
  f[i] = f[i-1]*i;
```

Aufgabe 2

```
#include <iostream>
using namespace std;
int main () {
  int fib [20];
  fib [0] = 0;
  fib [1] = 1;

  for (int i = 2; i < 20; i++) {
    fib[i] = fib[i-2] + fib[i-1];
  }

  for (int i = 0; i < 20; i++){
    cout << fib[i] << "\n";
  }
}
```

Aufgabe 3

- Duplikate
  Analyse: Die Aufgabenstellung sagt nichts darüber aus, wodurch die entfernten Duplikate ersetzt werden sollen. Die Aufgabenstellung muss also in diesem Punkt präzisiert werden. Man kann Duplikate durch eine spezielle Zahl – z.B. 0 – ersetzen. Dann kann aber eine echte eingegebene “0” nicht von dem Sonderzeichen “0” unterschieden werden. Die bessere Lösung ist das Feld zu verkleinern.

  Entwurf:

  ```
  #include <iostream>
  using namespace std;
  int main () {
    int a [10];
    int z = 10; // Zahl der Elemente

    //?? a einlesen

    for (int i = 0; i < z; i++) {
      //INV: a [0..i-1] ist frei von Duplikaten

      //sorge dafuer dass a[i] kein Duplikat im Rest
      //des Feldes hat. (Dann kann i erh"oht werden
      //und die Invariante ist immer noch OK):

      for (int k = i+1; k<z; k++){
        //INV: a[i+1 .. k-1] enthaelt kein Duplikat von a[i]

        if (a[i] == a[k]) { // a[k] ist Duplikat von a[i]
          ?? Entferne a[k] durch links-Verschieben ??
          ?? von a[k+1 .. z-1]                      ??

          a[z-1] = -99; // ungueltiger Wert
          z--;          // ein Element weniger!
        }
      }
    }
    //?? a bis z-1 ausgeben ??
  }
  ```

- k–Größtes: Das k–Größte: das größte für das es k-1 größere gibt.
  - Das 1–größte: das größte für das es 0 größere gibt.
  - Das 2–größte: das größte für das es 1 größere gibt.
  - Das 3–größte: das größte für das es 2 größere gibt.
  - ...

Daraus kann man eine Schleife konstruieren!

Nehmen wir an wir hätten das i–größte gefunden und in der Variablen `g` abgespeichert. Wie können wir mit ihm das (i+1)–größte finden?

Ganz einfach: Das (i+1)–größte muss das größte von denen sein, die kleiner als `g` sind:

```
#include <iostream>
using namespace std;
int main () {
  int a [10];
  int g = INT_MAX;   // das i-groesste
  int k;
  bool gef = false;  // k-groesstes gefunden ?

  ?? a und k einlesen ??

  for (int i=1; i<=k; i++) { // suche das i-groesste
    // INV: g enthaelt das (i-1)-groesste

    int n; // bisher gefundenes i-groesstes
    if (a[0] < g) {
      n = a[0];
      gef = true;
    } else {
      gef = false;
    }
    for (int j=1; j<10; j++){
      ?? ist a[j] ein besseres als das bisherige
      ?? i--groesste in n (falls vorhanden),
      ?? dann speichere es in n
      ?? wurde noch kein i--groesstes gefunden,
      ?? dann speichere in jedem Fall a[j] in n
    }
    // n ist unser neues g:
    if (gef)
      g = n;
    else
      break;
  }
  if (gef)
    cout << "Das " << k << "-groesste ist " << g << endl;
  else
    cout << "Konnte " << k << "-groesstes nicht finden" << endl;
}
```

Aufgabe 4

Kein Lösungshinweis.

Aufgabe 5

1. Prüfen Sie ihre Hypothese zum Verhalten des Programms mit Hilfe des Debuggers.
2. - `s2 = s1a;` Nicht erlaubt: unterschiedliche Typen.
   - `s2.sss = 5;` Nicht erlaubt. Als Selektor sind nur `s_i` und `s_c` erlaubt.
     Dass `sss` eine Variable mit dem String–Wert "`s_i`" ist, ist ohne Belang! Namen (Bezeichner) und Werte bewegen sich auf verschiedenen Ebenen, zwischen denen (in C++ und ähnlichen Sprachen) keinerlei Verbindung besteht.

Aufgabe 6

1. Ist ein `enum`–Wert als Feld–Index erlaubt? : Ja nach (expliziter oder impliziter) Konversion nach `int`.

2. Kann man mit einem `enum`–Wert ein Verbundelement selektieren: Nein natürlich nicht. Verbundelemente werden mit Namen selektiert, nicht mit Werten.

3. a) Variable: `int i;`
   b) Typ: `typedef int i;`
   c) Typ: `struct S {int i; int j;};`
   d) Typ und Variable: `struct S {int i; int j;}; S s;`
   e) 2 Typen: `struct S {int i; int j;}; typedef S s;`
   f) 2 Typen, eine Variable: `struct S {int i; int j;}; typedef S s; s s1;`
   g) Typ: `enum A {r, g, b};`
   h) Variable: `int A[10];`
   i) Typ: `typedef int A[10];`
   j) Variable: `int a[10];`
   k) Typ, Variable: `typedef int A; A a[10];`
   l) 2 Typen (`a` ist Typ "Feld von 10 `A`-Variablen"): `typedef int A; typedef A a[10];`
   m) Typ (Feld), Variable (2-dim. Feld): `typedef int A[10]; A a[10];`
   n) Typ (E), Variable (A): `enum E {r, g, b}; E A[10];`
   o) Typ (E), Variable (A): `enum E {r, g, b} A[10];`
   p) Zwei Typen (E, A): `typedef enum E {r, g, b} A[10];`

Aufgabe 7

In

```
...
struct Vektor { float x, y, z; };
Vektor v1, v2;
for ( char i = x; i <= z; ++i ) {  // UNSINN
  v1.i = 0.0;                      // UNSINN
  v2.i = 1.0;                      // UNSINN
}
```

werden Variablen, Namen von Komponenten und Werte verwechselt!

7 Funktionen und Methoden

7.1 Konzept der Funktionen

Funktionen: Funktionsargumenten einen Funktionswert zuordnen

Funktionen sind aus der Mathematik als Zuordnungen von Werten zu Argumenten bekannt. Funktionen werden üblicherweise in Form einer Rechenvorschrift definiert. Beispielsweise wird mit

$f(x, y) = 2 * x + y$

eine Funktion f definiert, die den *Argumenten (Parametern)* 3 und 2 den *Wert* 8 zuordnet.

Mathematiker sehen Funktionen meist als Spezialfall der Relationen. Die Funktion f von oben ist beispielsweise als Relation die unendliche Menge von Tripeln aus Argumenten und entsprechendem Wert:

Mathematische Interpretation der Funktion f als **Relation**:

$$
\begin{array}{l}
f = \{ \\
\quad < 0,0,0 >, < 0,1,1 >, < 0,2,2 >, \cdots \\
\quad < 1,0,2 >, < 1,1,3 >, < 1,2,4 >, \cdots \\
\quad < 2,0,4 >, < 2,1,5 >, < 2,2,6 >, \cdots \\
\quad \cdots \\
\quad \}
\end{array}
$$

Diese Sicht der Funktion ist statisch. Eine andere Betrachtungsweise ist dynamisch: die Funktion ist demnach keine unendliche Menge, sondern ein Verfahren (Algorithmus) mit dessen Hilfe den Argumenten ein Wert zugeordnet wird (Siehe Abbildung 19):

Interpretation der Funktion f als **Algorithmus**:

$f(x, y)$:
Nimm den Wert x und verdoppele ihn.
Addiere dazu den Wert y.
Das Ergebnis ist der Funktionswert zu x und y

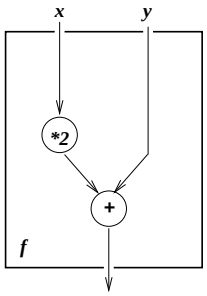


Abbildung 19: Funktion f als Algorithmus

Die Mathematikausbildung legt großen Wert darauf, dass die Sicht einer Funktion als Relation die “richtige” und “wissenschaftliche” ist. Im wahren Leben wird eine Funktion trotzdem fast immer als Verfahren verwendet, nach dessen Vorschrift Werte berechnet werden: Die Funktion wird als Algorithmus verwendet.[22] Es ist klar, dass dies auch unsere Sicht hier ist.

Eine Funktion in einem Programm: ein Algorithmus in exakter Notation

Eine Funktion ist also ein Algorithmus, ein Verfahren etwas zu berechnen. In einem Programm kann dieser Algorithmus zu dem gesamten Algorithmus des Programms beitragen. Er muss dazu in der korrekten Notation exakt nach den Vorschriften der Programmiersprache ausgedrückt werden.

Definition der Funktion f als Algorithmus:

[22] In wie weit sind die beiden Sichten einer Funktion – Algorithmus oder Relation – äquivalent?

in C++–Notation	in mathematischer Notation
`int f (int x, int y) { return 2*x + y; }`	$f(x,y) = 2 * x + y$

Die C++–Notation ist sicher intuitiv genauso verständlich wie die mathematische.

Funktionsaufruf: Funktion verwenden

Ein C++–Programm, das die Definition einer Funktion enthält, kann die Funktion auch verwenden. Man spricht dabei von einem *Aufruf der Funktion*. Im folgenden Beispiel–Programm wird f definiert, dann werden zwei Zahlen eingelesen, die Funktion f auf sie angewendet und das Funktionsergebnis ausgegeben.

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

int f (int x, int y) {     // <--- Beginn der Funktionsdefinition
   return 2*x + y;         // <--- hier geht es nach dem Aufruf hin
}

int main () {              // <--- hier beginnt der Programmlauf
   int a, b, c;
   cin >> a;
   cin >> b;

   c = f (a, b);           // <--- Aufruf der Funktion in einer Zuweisung

   cout << c << endl;      // <--- hier endet der Programmlauf
}
```

Mit der (Funktions-) Definition wird gesagt, wie die Funktion arbeitet. Beim (Funktions-) Aufruf werden die Argumente festgestellt, an die Funktion übergeben und der Funktionswert entsprechend der Definition berechnet. Im Beispiel oben:

- liest das Programm zwei Zahlen ein und speichert sie in `a` und `b`,
- dann wird die Funktion `f` aufgerufen und die Werte von `a` und `b` an `f` übergeben.
- `f` speichert die übergebenen Werte in `x` und `y`,
- `f` berechnet den Funktionswert `2*x+y`
- und gibt diesen dann zurück an die `main`–Funktion.
- `main` speichert den von `f` berechneten Wert in `c`
- und gibt ihn dann aus.

Funktionen sind als Teil–Algorithmen des Programms. Sie agieren innerhalb des Programms. Ihre Argumente (Parameter) und Werte dürfen nicht mit der Ein– und Ausgabe des Gesamtprogramms verwechselt werden (siehe Abbildung 20). Mit den Ein– / Ausgabe–Anweisungen (`cin>>...`, `cout<<...`) kommuniziert das Programm mit dem Benutzer. Mit der Parameterübergabe und der Rückgabe des Ergebnisses kommunizieren `main` und `f`.

Ablauf von Programmen mit Funktionen

Bei Programmen ohne Funktionen kann man mit einiger Berechtigung sagen, dass ihr Ablauf "oben" beginnt und dann mehr oder weniger gerade bis "unten" weitergeht. Wenn ein Programm Funktionen enthält, dann ist das nicht mehr so: der Ablauf "springt" jetzt zwischen den Funktionen hin und her. Es beginnt *immer* mit der `main`–Funktion, was auch immer im Text davor stehen mag! Von `main` aus kann es mit einem Funktionsaufruf in einer anderen Funktion weitergehen, dann zurück wieder in `main`, dann wieder in die gleiche oder eine andere Funktion, und so weiter; so lange, bis `main` endet. Beispiel:

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

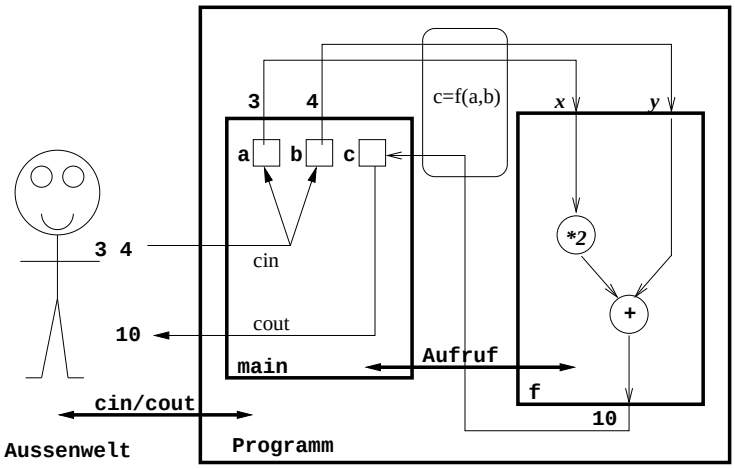


Abbildung 20: Funktionen arbeiten innerhalb des Programms

```
int g (int x) {       //  5. Aufuf mit Parameter 13
   return x+1;        //  6. von hier geht's zurueck zu f mit Wert 14
}

int f (int x, int y) {//  3. Aufuf mit Parameter 2, 3
     return g(10+y)   //  4. von hier geht's zu g mit dem Parameter 13
            + 2 * x;  //  7. von hier gehts zurueck zu main mit Wert 18 (=14+2*2)
}

int main () {         //  1. hier beginnt der Programmlauf
   int a, b, c;
   a = 2;
   b = 3;

   c = f (a, b);      //  2. von hier geht's zu f mit den Parametern 2 und 3
                      //  8. f liefert den Wert 18

   cout << c << endl; //  9. 18 wird ausgegeben
}                     // 10. hier endet der Programmlauf
```

7.2 Funktionen: freie Funktionen und Methoden

Zwei Varianten von Funktionen: freie Funktion und Methode

C++ ist *objektorientiert*. Das bedeutet, dass es neben "normalen" Funktionen auch noch *Methoden* gibt. Methoden sind eine besondere Art von Funktionen. In C++ gibt es also Funktionen in zwei Varianten:

- *freie Funktion*, kurz Funktion (engl. *free (standing) function*)
- *Methode*, auch Mitglieds–Funktion (engl. *method, member function*)

Ein Beispiel für eine vordefinierte *freie Funktion* ist `pow`, die Potenzierungsfunktion aus der Mathematik–Bibliothek:

```
#include <cmath>
...
float x, y;
...
x = pow (y, 2);
...
```

`pow (y, 2)` berechnet y^2 (wobei y der aktuelle Wert von `y` ist).

Ein Beispiel für eine vordefinierte *Methode* ist `at` auf Zeichenketten:

```
#include <string>
...
int i;
string s;  char c;
...
c = s.at(i);
...
```

`s.at(i)` hat als Wert das i–te Zeichen in der Zeichenkette s. (i ist der aktuelle Wert von `i` und s der aktuelle Wert von `s`.)

Eigenschaften von freien Funktionen und Methoden

Funktionen in beiden Varianten berechnen Werte und/oder verändern Variablenwerte. *Methoden* operieren dabei im Bereich eines bestimmten *Objekts* (einer Variablen) – daher kommt der Begriff "objektorientiert". Im Beispiel oben bestimmt `at` den Wert an Position `i` in der Variablen `s`. Die Wirkung von `at` hängt also ganz wesentlich vom aktuellen Wert von `s` ab. So bedeutet

```
s.at(i)
```

dass `at` *von* `s` aktiviert wird. Die Methode `at` von `s` berechnet einen Wert in `s`. Nicht umsonst wird bei Methoden der Punkt–Operator benutzt. Methoden sind aktive Komponenten eines Verbunds.

Freie Funktionen operieren global auf der Ebene des Programms. Ihre Freiheit besteht darin, dass sie mit ihrer Wirkung nicht an ein bestimmtes Objekt (Variable) gebunden sind. So bedeutet

```
pow(y, 2)
```

dass `pow` (im Programm) aktiviert wird.

Definition einer freien Funktion

Freie Funktionen und Methoden können in beliebiger Anzahl in einem Programm neu definiert werden. Ein Beispiel für eine selbst definierte freie Funktion haben wir in der Funktion f weiter oben gesehen. Ein etwas komplexeres Beispiel ist:

```
int my_mult (int x, int y) {
  int res = 0;
  for (int i=0; i<x; ++i)   // x Mal y
     res = res + y;
  return res;                  // Return-Anweisung: Beendet Funktion,
}                              // und liefert Wert an Aufrufstelle
...
cout << my_mult (3, 4) << endl; // Aufrufstelle: Aufruf der Funktion
...
```

Hier wird eine Funktion mit dem Namen `my_mult` definiert. Sie hat zwei Argumente und ein Ergebnis vom Typ `int`:

```
int my_mult (int x, int y)
```

In der Ausgabeanweisung

```
cout << my_mult (3, 4) << endl;
```

wird die Funktion aufgerufen.

Die Funktion multipliziert ihre Argumente durch Additionen miteinander und liefert dann das Ergebnis zurück an die Aufrufstelle. Dort wird es dann von `cout` ausgegeben.

Bitte verwechseln Sie nicht den Wert einer Funktion mit einer Ausgabe! Werte werden an die Aufrufstelle (im Programm!) geliefert, Ausgaben gehen auf den Bildschirm!

Definition einer Methode

Ein Beispiel für eine selbst definierte Methode ist hier `laenge`. Sie berechnet die Länge des Vektors *zu dem sie gehört*. Ein Vektor kann mit ihr seine Länge bekannt geben:

```
struct Vektor {            // Definition von Vektor
   float x;
   float y;
   float laenge ();        // Deklaration von Vektor::laenge
};

float Vektor::laenge () {  // Definition von Vektor::laenge
   return sqrt (x*x + y*y);
}
...
Vektor p, q = {1.0, 2.1};
float l1, l2;
...
p.x = q.x + 3;      // Zugriff auf die Datenkomponenten von p und q
p.y = q.x + q.x;
...
l1  = p.laenge();   // Zugriff auf die Methoden von p und q
l2  = q.laenge() + p.laenge();
...
```

Methoden Können genau wie die Datenkomponenten zu einem Verbund gehören. Die Definition des Typs `Vektor` oben, bedeutet dass jeder Vektor – also jede Variable mit dem Typ `Vektor` – seine eigenen drei Komponenten hat:

- `x`: eine `float`–Variable,
- `y`: eine `float`–Variable,
- `laenge`: eine Methode, die einen `float`–Wert liefert.

Die Methode `laenge` taucht als *Deklaration* innerhalb der Definition von `Vektor` auf. Der Definition des Verbundes inklusive der Deklaration seiner Methode folgt die *Definition* der Methode:

```
float Vektor::laenge () { ... }
```

In ihr wird gesagt, wie die Methode arbeitet. Eine Methoden–Definition sieht im Prinzip genauso aus, wie eine Funktionsdefinition, nur dass dem Namen `laenge` ein `Vektor::` vorangestellt wird. Damit wird zum Ausdruck gebracht, dass `laenge` zu Objekten vom Typ `Vektor` gehört.

Der wichtigste Einsatz von Methoden ist der Zugriff auf "Innereien" eines Objekts (einer Variablen). Entweder wie hier lesend oder auch verändernd. Dabei werden Berechnungen und Aktionen abhängig von den Werten der Datenkomponenten ausgeführt. Die Kombination der Werte des Objekts nennt man auch den *Zustand eines Objekts*.

Definition und Deklaration

Eine Definition beschreibt vollständig ein neu eingeführtes Konstrukt; eine Deklaration informiert den Compiler über wesentliche Eigenschaften eines Konstruktes ohne es unbedingt damit vollständig zu beschreiben. Innerhalb der Definition von `Vektor` taucht die `Deklaration` von `laenge` auf:

```
struct Vektor {           // Definition von Vektor
   float x;
   float y;
   float laenge ();       // Deklaration von Vektor::laenge
};
...
```

Hier wird `laenge` nur unvollständig beschrieben. Der Compiler wird lediglich darüber informiert, dass es eine Methode von `Vektor` ist, keinen Parameter hat und ein Ergebnis vom Typ `float` liefert. Es ist eine *Deklaration*. Die vollständige Beschreibung, die *Definition* folgt später:

```
...
float Vektor::laenge () {  // Definition von Vektor::laenge
   return sqrt (x*x + y*y);
}
```

7.3 Freie Funktionen

Funktionen: Wertberechnung

Eine (freie) Funktion ist meist ein Programmstück zur Berechnung eines (Ergebnis–) Wertes aus einem oder mehreren (Argument–) Werten. Sie ist zunächst einfach als Mechanismus zur Abkürzung eines Programms zu verstehen. Die Abkürzung besteht darin, dass Wiederholungen vermieden werden. Nehmen wir an, in einem Programm soll an zwei Stellen das Minimum von zwei Zahlen berechnet werden.

```
#include <iostream>

using namespace std;

int main () {
  int a, b, c, d;
  cout << " a: "; cin >> a;
  cout << " b: "; cin >> b;
  cout << " c: "; cin >> c;
  cout << " d: "; cin >> d;

  int m1;
  if (a < b) m1 = a;
  else       m1 = b;

  int m2;
  if (c < d) m2 = c;
  else       m2 = d;

  cout << "Min (a,b) = " << m1 << endl;
  cout << "Min (c,d) = " << m2 << endl;
}
```

Hier wird zweimal mit praktisch dem gleichen Code das Minimum berechnet. Bei solchen kleinen und trivialen Programmstückchen ist die Wiederholung nicht sehr tragisch, aber wenn es sich um ein paar hundert Zeilen handelt, kann das schon sehr lästig sein. Bereits sehr früh hat man sich darum mit den Funktionen einen Abkürzungsmechanismus überlegt. In C++ sieht das äquivalente Programm mit einer Funktion so aus:

```
#include <iostream>

using namespace std;

int min (int, int);      // Deklaration:
                         // min: int x int -> int
int main () {
  int a, b, c, d;
  cout << " a: "; cin >> a;
  cout << " b: "; cin >> b;
  cout << " c: "; cin >> c;
  cout << " d: "; cin >> d;

  int m1 = min (a,b);    // 1. Verwendung (Aufruf) von min
  int m2 = min (c,d);    // 2. Verwendung (Aufruf) von min

  cout << "Min (a,b) = " << m1 << endl;
  cout << "Min (c,d) = " << m2 << endl;
}

int min (int x, int y) { // Definition
  if (x < y) return x;   // der Funktion min
  else       return y;

}
```

Der Code zur Bestimmung des Minimums wird hier in einer Funktion zusammengefasst, die dann an beliebig vielen Stellen aufgerufen werden kann.

Deklaration einer Funktion

Das Programm beginnt mit einer *Deklaration* der Funktion `min`:

```
int min (int, int);
```

Mit dieser *Funktionsdeklaration* wird

- der *Name* `min` und
- der *Typ* der Funktion

festgelegt. Der *Typ einer Funktion* besteht aus den Informationen:

- Typ des *Ergebnises* (im Beispiel `int`), sowie
- Zahl und Typ der *Argumente* (im Beispiel zwei `int`–Argumente)

Dies wird dem Compiler mit der Deklaration bekannt gegeben. Er benutzt und benötigt diese Informationen um die Aufrufstellen korrekt übersetzen zu können.

Funktionsdeklarationen werden – aus historischen Gründen – gelegentlich auch “Prototypen” genannt.[23]

Funktionsaufruf

Die Verwendung einer Funktion wird allgemein auch Funktions–*Aufruf* genannt. Im Beispiel oben wird die Funktion `min` zweimal in einer Initialisierung aufgerufen:

```
int m1 = min (a,b);
int m2 = min (c,d);
```

Funktionsaufrufe können wie beliebige Ausdrücke mit dem Typ des Ergebnisses verwendet werden, z.B:

```
c = 2 * min(a,b) + min (c,2); oder
cout << "Min (c,d) = " << min (c,d) << endl;
```

Beim Funktionsaufruf werden die Argumente ausgewertet und als *aktuelle Parameter* an die Funktion übergeben. Bei dieser Parameterübergabe wird der Wert eines aktuellen zu dem des entsprechenden formalen Parameters. Die Funktion wird mit diesen Werten abgearbeitet und liefert dann ihr Ergebnis zurück. (Siehe Abbildung 21)

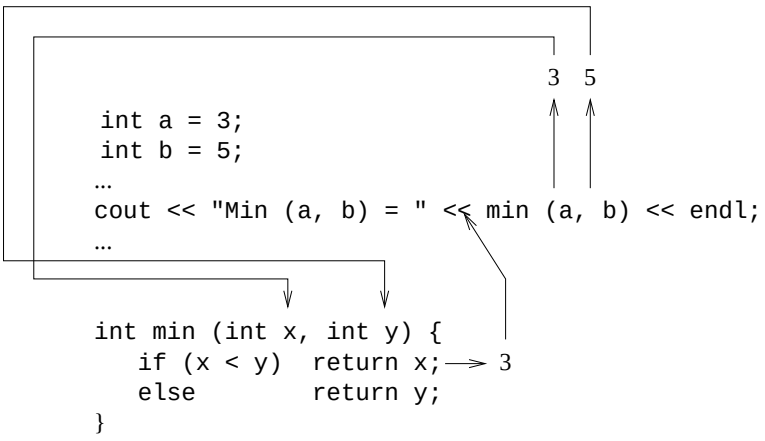


Abbildung 21: Funktionsaufruf

Funktionsdefinition

In der Funktionsdefinition wird festgelegt was und wie die Funktion arbeitet. In unserem Beispiel:

```
int min (int x, int y) {  // Kopf mit formalen Parametern
    if (x < y) return x;
    else       return y;
}
```

[23] Dies ist die Bezeichnung in (ANSI–) C.

wird festgelegt, dass die beiden aktuellen Parameter (die Argumente der Funktion) zuerst den *formalen Parametern* `x` und `y` zugewiesen werden. Nach dem Aufruf

```
min (3, 4)
```

beginnt darum `min` mit `x = 3` und `y = 4`. Nach der Belegung der formalen Parameter mit den Argumenten (den aktuellen Parametern) wird der Funktionskörper abgearbeitet. Anweisung für Anweisung wird ausgeführt, solange bis entweder eine `return`–Anweisung angetroffen wird oder die Funktion zu Ende ist. Eine `return`–Anweisung gibt ihren Wert als Wert der Funktion an die Aufrufstelle zurück.

Funktionskopf und –körper

Die Funktionsdefinition besteht aus einem *Funktionskopf* (engl. *Header*):

```
int min (int x, int y) // Funktionskopf
```

und dem *Funktionskörper* (engl. *Body*), dem Rest der Definition. Im Kopf werden die formalen Parameter definiert. Der Kopf der Definition entspricht der Funktionsdeklaration, außer, dass hier bei der Definition die Namen der formalen Parameter angegeben werden müssen. In der Deklaration kann man sie dagegen weglassen.

```
int min (int, int);            // OK, Deklaration
int min (int x, int y);        // OK, Deklaration
int min (int x, int y) {...}   // OK, Definition
int min (int, int) {..}        // FEHLERHAFTE Definition
```

Die `return`–Anweisung

Der Körper einer Funktion ist eine zusammengesetzte Anweisung, die üblicherweise eine oder mehrere `return`–*Anweisung(en)* enthält. Eine `return`–Anweisung veranlasst "die Funktion mit dem angegebenen Wert zurückzukehren". Die Funktion wird abgebrochen und der Wert der `return`–Anweisung wird zum Wert des Aufrufs. Beispielsweise hat der Aufruf

```
min (3, 4)
```

den Wert 3, da `min` mit `return x` – mit einem `x`–Wert von 3 – abgebrochen wird.

Achtung: Die `return`–Anweisung darf nicht mit der Ausgabe–Anweisung (`cout<<...`) verwechselt werden! Mit `cout` wird ein Wert aus dem gesamten Programm heraus transportiert. `return` bewegt einen Wert innerhalb des Programms.

Deklaration und Definition

In Bezug auf die Reihenfolge von Definition, Deklaration und Verwendung des definierten Namens (d.h. Aufruf) gilt:

- Eine Funktion muss *vor ihrer Verwendung* (vor dem Aufruf) deklariert werden.
- Die *Deklaration* einer Funktion *kann mehrfach* in einem Programm auftauchen.
- Eine Funktion muss in einem Programm *genau einmal definiert* werden.
- Auf die Deklaration kann verzichtet werden, wenn die Definition vor jeder Verwendung erscheint. (Die Definition ist dann gleichzeitig auch eine Deklaration.)

Definitionen und Deklarationen haben unterschiedliche Funktionen in einem Programm:

- **Deklaration:**
  Eine *Deklaration* ist eine Information des Programms an den Compiler über den *Typ* der Funktion. Diese Information benutzt der Compiler um Maschinencode für die *Aufrufstellen* der Funktion zu erzeugen. Dabei wird auch die Korrektheit der Verwendung geprüft (richtige Typen und Anzahl der Argumente, korrekter Typ des Ergebnisses, etc.). Allgemein:
  Eine Deklaration führt einen Namen mit einem bestimmten Typ in das Programm ein.
- **Definition:**
  Eine *Definition* ist eine Information des Programms an den Compiler über die *gesamte* Funktion (Typ +

Anweisungen). Diese Information benutzt der Compiler um den Maschinen–Code für *die Funktion selbst* zu erzeugen. Allgemein:

Eine Definition führt ein Konstrukt ins Programm ein und ordnet ihm einen Namen zu.

Eine Deklaration ist also eine unvollständige Definition, bei der nur der Typ festgelegt wird. Eine Definition legt neben dem Typ auch alle anderen Eigenschaften fest. *Eine Definition ist immer auch eine Deklaration*, z.B. ist jede Variablendefinition auch eine Deklaration:

```
int a[10];
```

Dies ist eine Deklaration von `a`: ab hier kann `a` verwendet werden, so wie es sich für etwas vom Typ `int`–Feld gehört. Der Compiler wird ab hier jede Verwendung wie etwa

```
a[i] = a[j];
```

prüfen und gegebenenfalls übersetzen. Es ist aber auch eine Definition, denn genau hier wird `a` als ein `int[10]` Feld definiert und damit der Compiler aufgefordert Code zu erzeugen der Platz für 10 `int`s schafft.

Dagegen ist eine Funktionsdeklaration wie etwa

```
int f (float);
```

nur eine Deklaration und keine Definition. `f` wird mit dem Typ "Funktion von `float` nach `int`" verknüpft, aber es wird nicht gesagt, welche Funktion denn nun `f` wirklich zugeordnet ist. Alle Aufrufe von `f` können ab hier übersetzt werden, aber Code für die Funktion selbst wird nicht erzeugt.

Wiederholungen der Deklaration sind erlaubt. Wiederholungen der Definition sind *nicht* erlaubt. Mit der Definition wird etwas unter einem bestimmten Namen ins Programm eingeführt. Mit einer Wiederholung der Definition würde ein Name mit zwei (wenn auch eventuell gleichartigen) Bedeutungen belegt und der Compiler müsste zweimal Code für das gleiche erzeugen.

Namen von Funktionen: Überladung

Zwei Funktionen in einem Programm dürfen den gleichen Namen tragen, wenn sie sich in ihren Argumenttypen unterscheiden. Beispiel:

```
struct Kreis {
  Punkt mitte;
  float radius;
};

struct Rechteck {
  float seite_1;
  float seite_2;
};

float flaeche (Kreis k) {
  return (3.1415 * k.radius * k.radius);
}

float flaeche (Rechteck q) {
  return (q.seite_1 * q.seite_2);
}
```

Die Vergabe des gleichen Namens an unterschiedliche Funktionen bezeichnet man als *Überladung* (engl. *overloading*).

7.4 Methoden

Methoden sind Funktionen in Objekten

Eine Methode ist eine Funktion die *innerhalb einer Verbund–Variablen* agiert. Sie berechnet einen Ergebnis–Wert aus Argument–Werten und kann dabei auf die Bestandteile der Verbund–Variablen zugreifen. Verbund–Variablen werden üblicherweise und auch hier "*Objekt*" genannt.

Methoden und freie Funktionen können alternativ verwendet werden. Im folgenden Beispiel vergleichen wir die Lösung eines Problems durch eine freie Funktion mit der durch eine Methode. Angenommen es seien Punkte

und Strecken zu modellieren. Punkte haben eine x– und eine y–Komponente. Strecken werden durch ihre zwei Endpunkte dargestellt. Eine entsprechende Typdefinition ist:

```
struct Punkt {
   float   x;
   float   y;
};
struct Strecke {
   Punkt   a;
   Punkt   e;
};
```

Die Länge einer Strecke kann nach dem Satz des Pythagoras mit einer *freien Funktion* berechnet werden:

```
#include<cmath>  // wegen der Wurzelfunktion sqrt

float laenge (Strecke s) {
   return sqrt ((s.e.x - s.a.x)*(s.e.x - s.a.x)
                + (s.e.y - s.a.y)*(s.e.y - s.a.y));
}
```

Diese Definitionen können dann wie folgt verwendet werden:

```
  Punkt p1 = {2.0, 4.0},
        p2 = {3.0, 2.0};
  Strecke str = {p1, p2};

  float d = laenge(str);
```

Hier wird “Länge” als Funktion verstanden, die Strecken auf Zahlen abbildet. Die Länge kann aber auch als *Eigenschaft* eines Objekts vom Typ Strecke betrachtet werden. Demnach ist Länge eine Komponente von Objekten vom Typ Strecke. Es ist natürlich keine Datenkomponente sondern eine *Methode*:

```
struct Punkt { .. wie oben .. };
struct Strecke {
   Punkt   a;
   Punkt   e;
   float   laenge ();   // Deklaration der Methode laenge
};
```

Die Methode muss jetzt noch definiert werden:

```
float Strecke::laenge () {
   return sqrt ((a.x - e.x)*(a.x - e.x)
                + (a.y - e.y)*(a.y - e.y));
}
```

Mit `Strecke::laenge` wird gesagt dass `laenge` zum Typ `Strecke` gehört. Diese Methode hat natürlich keinen Parameter. Innerhalb der Methode wird direkt auf die Datenkomponenten `a` und `e` zugegriffen. So ausgestattet kann jede Strecke jetzt selbst Auskunft darüber geben, wie lang sie ist.

Diese Definitionen können dann wie folgt verwendet werden:

```
  Punkt p1 = {2.0, 4.0},
        p2 = {3.0, 2.0};
  Strecke str = {p1, p2};

  float d = str.laenge();  // Strecke str, sag wie lang du bist!
```

Beim Aufruf dieser Methode findet keine Parameterübergabe statt. Die zur Längenberechnung notwendigen Werte findet die Methode direkt in “ihrem” Objekt. (Siehe Abbildung 22)

Methoden–Deklaration

Die Deklaration einer Methode ist Bestandteil der Deklaration eines Verbund–Typs:

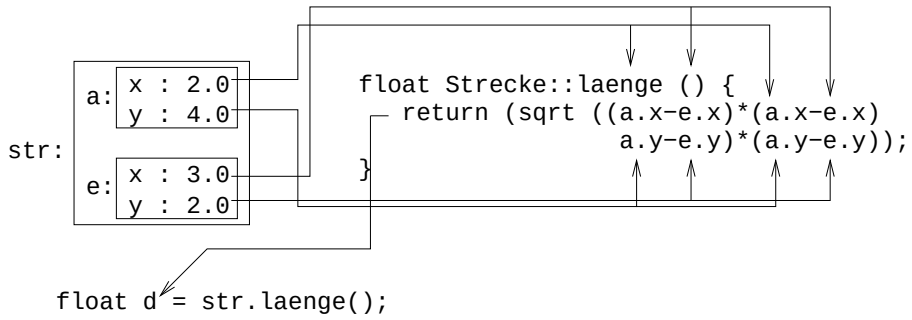


Abbildung 22: Methodenaufruf

```
struct Strecke {
    ...
    float  laenge ();
    ...
};
```

In der Methodendeklaration wird festgelegt:

- Zum (Verbund–) Typ `Strecke` gehört eine Methode mit Namen `laenge`,
- diese Methode hat keinerlei Argumente und liefert einen Wert vom Typ `float`.

Methodenaufruf

Der *Aufruf* einer Methode ist ihre Anwendung auf eine Variable des Typs für den sie deklariert wurde. Eine Methode eines Typs kann auf jede Variable dieses Typs angewendet werden.

Statt von "Variable" spricht man meist von "Objekt": Ein *Objekt* ist offiziell in C++ eine Variable mit selbst definiertem Typ (also eine `enum`–, `struct`,– oder `class`–Variable). Im allgemeinen Sprachgebrauch werden jedoch nur solche Variablen "Objekt" genannt, zu deren Typ mindestens eine Methode gehört.

Methoden können nur auf Variablen vom passenden Typ angewendet werden. Zum Zugriff benutzt man den Punkt–Operator: *Methoden sind "aktive" Verbund–Komponenten.*

Methoden liefern Ergebnisse, die von "ihrer" Variablen abhängen:

```
... Punkt und Strecke wie oben ...

Punkt   p1 = {2, 2},
        p2 = {5, 6},
        p3 = {8, 2};

Strecke v1 = {p1, p2},
        v2 = {p1, p3};

cout << "Laenge von v1: " << v1.laenge() << endl;
cout << "Laenge von v2: " << v2.laenge() << endl;
```

Dieses Programm wird die folgende Ausgabe erzeugen:

```
Laenge von v1: 5
Laenge von v2: 6
```

Methoden–Definition

Da Methoden zu den Variablen (Objekten) eines bestimmten Typs gehören, muss bei ihrer Definition dieser Typ angegeben werden:

```
float Strecke::laenge () { ...}
```

Innerhalb einer Methodendefinition können alle Komponenten des zugehörigen Verbundtyps verwendet werden. Methoden (und ebenso auch freie Funktionen) können beliebig viele Parameter haben. Beispiel:

```
struct S {
    int  x;
    int  y;
    int  f (int);
};

int S::f (int p) {
    return x+y+p;
}
```

Der Wert der Komponenten (`x, y` im Beispiel) ist der aktuelle Wert der Komponenten der Variablen, auf welche die Methode angewendet wird:

```
 S s, t;

 s.x = 1; s.y = 3;
 t.x = 2; t.y = 4;

 cout << s.f(0); // Ausgabe: 4 = 1+3+0
 cout << t.f(1); // Ausgabe: 7 = 2+4+1

 s.x = 10;

 cout << s.f(5); // Ausgabe: 18
 cout << t.f(0); // Ausgabe: 6
```

7.5 Funktionen und Methoden in einem Programm

Funktionen und Verbunde mit Methoden

Freie Funktionen und Verbunde mit Methoden können in einem Programm zusammen auftreten. Beispielsweise können Funktionen Objekte als Parameter und als Ergebnis haben. Beispiel:

```
struct Quadrat {
  float seite_1;
  float seite_2;
  float flaeche ();
};

float Quadrat::flaeche () {
  return (seite_1 * seite_2);
}

//Eine Funktion mit Objekten als Parameter
bool gleich_gross (Quadrat q1, Quadrat q2) {
  return (q1.flaeche() == q2.flaeche());
}

int main () {
  Quadrat s1 = {2.0, 4.0},
          s2 = {3.0, 3.0};

  if (gleich_gross (s1, s2))
    cout << "s1 und s2 sind gleich gross\n";
  else
    cout << "s1 und s2 sind nicht gleich gross\n";
}
```

Hier wird eine freie Funktion `gleich_gross` definiert. Sie hat zwei Parameter vom Typ `Quadrat` und benutzt deren Methode `flaeche` um ihr Ergebnis zu berechnen.

Namen von Methoden und Funktionen

Methoden unterschiedlicher Verbundtypen können den gleichen Namen haben, sie werden trotzdem unterschieden. Dies wird oft ausgenutzt um Methoden den gleichen Namen zu geben, wenn sie das Gleiche bei unterschiedlichen

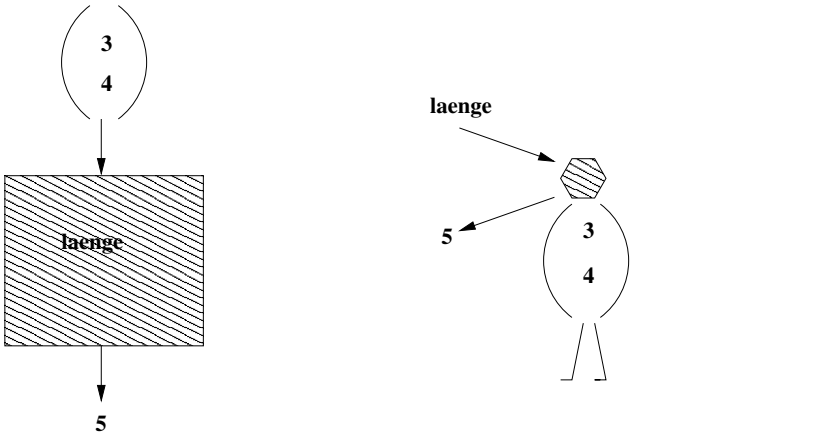
Objekten tun. Ein Beispiel ist die Berechnung der Fläche von Kreisen und Quadraten mit zwei Methoden die beide den Namen `flaeche` tragen:

```
struct Kreis {
  Punkt mitte;
  float radius;
  float flaeche ();
};

struct Quadrat {
  float seite_1;
  float seite_2;
  float flaeche ();
};

float Kreis::flaeche () {    //Kreisflaeche
  return (3.1415 * radius*radius);
}

float Quadrat::flaeche () {  //Flaeche eines Quadrats
  return (seite_1 * seite_2);
}
...
int main () {
  Kreis   k;
  Quadrat q;
  ...
  if (k.flaeche() > q.flaeche())
    cout << "Der Kreis hat die groessere Flaeche!\n";
  ...
}
```



Länge als freie Funktion

Abbildung Vektor −> Float−Zahl

Länge als Methode von Vektor

Frage an einen Vektor mit Float−Zahl als Antwort

Abbildung 23: Länge eines Vektors als freie Funktion oder Methode

Warum Methoden

Mit Methoden können Daten und Funktionen die zusammengehören auch unter einem gemeinsamen Dach zusammengefasst werden. Das “Dach” ist der Typ. Statt die Flächenberechnungen von Kreis und Quadrat als Methode zu definieren, hätte man auch zwei Funktionen verwenden können (man beachte Gemeinsamkeiten und Unterschiede im Vergleich zu oben):

```
struct Kreis {
  Punkt mitte;
  float radius;
```

```
};

struct Quadrat {
  float seite_1;
  float seite_2;
};

float flaeche (Kreis k) {
  return (3.1415 * k.radius * k.radius);
}

float flaeche (Quadrat q) {
  return (q.seite_1 * q.seite_2);
}
...
int main () {
  Kreis   k;
  Quadrat q;
  ...
  if (flaeche(k) > flaeche(q))
    cout << "Der Kreis hat die groessere Flaeche!\n";
  ...
}
```

Mit

```
Kreis::flaeche () {...}
```

sieht man sofort, dass die Funktion `flaeche` zum Typ `Kreis` gehört. Den Aufruf

```
k.flaeche()
```

erkennt man sofort als Aktion auf dem Objekt `k`.

Der Unterschied liegt nicht nur in der übersichtlicheren Organisation des Quelltexts. In der ersten (Methoden–) Form wird dem Compiler mehr von den Absichten der Programmiererin bekannt gegeben. Hier beispielsweise, dass die Flächenberechnung zu dem Verbundtyp (`Kreis` bzw. `Quadrat`) gehört. Dadurch kann dem Programmierer Arbeit abgenommen werden: So entfällt etwa die Selektion der Verbundkomponenten. Man schreibt

```
... radius ...
```

in der Methode statt

```
... k.radius ..
```

in der Funktion. Zur Laufzeit wird es natürlich kaum einen Unterschied zwischen den beiden Varianten geben. Zu beiden Programmvarianten werden letztlich die mehr oder weniger gleichen Maschineninstruktionen erzeugt werden.

Die Frage “Methode” oder “freie Funktion” ist letztlich eine philosophische Frage, bei der es darum geht wie eng man eine Aktion an einen Typ und dessen Objekte binden will. Ist die Länge beispielsweise etwas das einen Vektor verarbeitet und einen Zahlwert liefert, oder etwas, das man einen Vektor fragen kann, der dann mit einem Zahlwert antwortet. (Siehe Abbildung 23)

7.6 Übungen

Aufgabe 1

1. Schreiben Sie eine Funktion `f`, die zwei int-Werte addiert und das Ergebnis zurück gibt.
2. a) Schreiben Sie eine Funktion `f1`, die einen übergenen int–Wert *zurück* gibt.
   b) Schreiben Sie eine Funktion `f2`, die einen übergenen int–Wert *ausgibt*.
   c) Schreiben Sie eine Funktion `f3`, die int–Wert einliest und *zurück* gibt.
   d) Schreiben Sie ein Programm, in dem die drei Funktionen aufgerufen werden.
3. Definieren Sie einen Datentyp `Vektor`, der einen Vektor in der Ebene darstellt.
   a) Schreiben Sie eine (freie) Funktion `f1`, die einen Vektor einliest und zurück gibt.
   b) Schreiben Sie eine (freie) Funktion `f2`, die die Länge eines übergenen Vektors berechnet und zurück gibt.
   c) Schreiben Sie eine (freie) Funktion `f3`, die zwei Vektoren addiert und die Summe zurück gibt.
   d) Versehen Sie Ihren Vektortyp mit einer Methode `m1` die Vektoren einliest.
   e) Versehen Sie Ihren Vektortyp mit einer Methode `m2` die die Länge eines Vektors berechnet und zurück gibt.
   f) Schreiben Sie ein Programm, in dem diese Funktionen und Methoden aufgerufen werden.
4. Erläutern Sie den Unterschied zwischen dem Rückgabewert und der Ausgabe einer Funktion.

Aufgabe 2

1. Kann eine Komponente eines Verbund–Typs S1 wieder ein Verbund–Typ S2 sein? Können S1 und S2 sogar der gleiche Typ sein, also ist `struct S { ... S s; ... };` erlaubt?
2. Können in zwei Verbund–Typen Komponenten den gleichem Namen haben? Gilt Ihre Antwort auch für Methoden?
3. Schreiben Sie eine Funktion, die das Maximum ihrer drei `int`–Parameter bestimmt.
4. Schreiben Sie eine Funktion `ggt`, die den grössten gemeinsamen Teiler von zwei positiven ganzen Zahlen berechnet.
5. Schreiben Sie eine Funktion `kgv`, die das kleinste gemeinsame Vielfache von zwei positiven ganzen Zahlen berechnet.
6. Die Angestellten einer Firma seien im Feld `firma` zu finden:

```
typedef int PersonalNr; //Nr 0 ist ungueltig!
struct Angestellt {
   string     name;
   PersonalNr nr;
   PersonalNr chef;
   PersonalNr untergeben[20];
};
typedef Angestellt Firma[100];
Firma firma;
```

   Die Komponente `nr` enthält die Personalnummer der/des Angestellten, `chef` und `untergeben` enthalten Verweise (in Form einer Personalnummer) auf die Vorgesetzte bzw. die Untergebenen. Ein Wert "0" bedeutet, dass es keine Vorgesetzte bzw. keinen entsprechenden Untergebenen gibt. Die Variable `firma` enthält die Beschreibung aller Angestellten.

   Schreiben Sie eine Funktion

   `bool ist_untergeben (PersonalNr, PersonalNr, Firma)`

   die feststellt, ob der Angestellte mit der ersten Personalnr. der Angestellten mit der zweiten Personalnr. in der Firma (dritter Parameter) unterstellt ist.

7. Der Typ `Firma` sei jetzt wie folgt definiert:

```
typedef int PersonalNr; //Nr 0 ist ungueltig!
struct Angestellt {
   string     name;
   PersonalNr nr;
   PersonalNr chef;
   PersonalNr untergeben[20];
};

struct Firma {
   Angestellt angestellt [100];
   string     name (PersonalNr);
   bool       ist_untergeben (PersonalNr, PersonalNr);
};
```

   Die Methode `Firma::name` soll den Namen des Angestellten mit der übergebenen Personalnummer liefern und `ist_untergeben` soll (wie oben) testen, ob der Angestellte mit der ersten Personalnummer ein Untergebener der Angestellten mit der zweiten Personalnummer ist.

Aufgabe 3

Schreiben Sie eine Funktion die einen Wert nach folgender Formel berechnet und zurück gibt:

$$s = \sum_{i=0}^{n} 2 \cdot (i+1)$$

Aufgabe 4

Definieren Sie einen Datentyp `Feld` mit den Methoden `lese`, `schreibe` und `sortiere`, die das Feld lesen, sortieren und ausgeben. Ein `Feld` soll 20 `int`–Werte enthalten.

Aufgabe 5

Die Quersumme einer positiven ganzen Zahl ist die Summe ihrer Ziffern. Beispielsweise ist 9 die Quersumme von 135.

1. Schreiben Sie eine Funktion `quer`, die die Quersumme einer positiven ganzen Zahl (Typ `unsigned int`) berechnet. Benutzen Sie dabei folgende Hilfsfunktionen:

   a) `string ziffern (unsigend int)`

   Diese Funktion wandelt eine Zahl (Typ `unsigned int`) in eine Ziffernfolge als Zeichenkette (Typ `string`) um:

```
string ziffern (unsigned int x) {
  string res = "";
  while (x > 0) {
    res = char('0' + x%10) + res;
    x = x/10;
  }
  return res;
}
```

   b) `unsigned int zifferQuer (string)`

   Diese Funktion berechnet die Quersumme einer Zahl, die in Form einer Zeichenkette von Ziffer–Zeichen übergeben wird. Zum Beispiel: `zifferQuer ("157") = 13`

   Diese Funktion müssen Sie selbst schreiben.

2. Schreiben Sie ein Programm, das eine positive ganze Zahl einliest, ihre Quersumme (mit Hilfe Ihrer Funktion `quer`) berechnet und diese dann ausgibt.

7.7 Lösungshinweise

Aufgabe 1

1. ```
   int f (int x, int y) { return x+y; }
   ```

2. ```
   #include <iostream>
   using namespace std;

   int  f1 (int i) { return i; }
   void f2 (int i) { cout<<i<<endl; }
   int  f3 ()      { int i; cin << i; return i; }

   int main() {
     int x;
     x = f3();
     x = f1 (x+1);
     f2 (x);
   }
   ```

3. ```
   #include <iostream>
   #include <cmath>
   using namespace std;

   struct Vektor {
     float x;
     float y;
     void  m1();
     float m2();
   };
   Vektor f1 () {
     Vektor res;
     cin >> res.x >> res.y;
     return res;
   }
   float f2 (Vektor v) {
     return sqrt(v.x*v.x + v.y*v.y);
   }
   Vektor f3 (Vektor v1, Vektor v1) {
     Vektor res;
     res.x = v1.x+v2.x;
     res.y = v1.y+v2.y;
     return res;
   }
   void Vektor::m1() {
     cin >> x >> y;
   }
   void Vektor::m2() {
     return sqrt(x*x + y*y);
   }
   int main () {
     Vektor v1, v2;
     v1 = f1();         // einlesen mit Funktion
     v2.m1();           // einlesen mit Methode
     float x = f2(v1);  // Laenge berechnen mit Funktion
     float x = v2.m2(); // Laenge berechnen mit Methode
     v2 = f3 (v1, v2);  // addieren
   }
   ```

4. Siehe Skript.

Aufgabe 2

1. Ein Verbund–Typ darf Komponente eines anderen Verbund–Typs sein, es sei denn es tritt eine Rekursion in der Form auf, dass ein Verbund–Typ sich selbst direkt oder indirekt als Komponente enthält.
   ```

```
struct S1 { ... S2 s; ... };
```

ist also im Allgemeinen erlaubt (wenn auch nicht immer).

```
struct S { ... S s; ... };
```

ist nie erlaubt!

2. Ja! Ja! Komponenten und Methoden gehören zu einem bestimmten Verbund–Typ. Bei der Identifikation einer Komponente oder einer Methode eines Verbunds wird immer dessen Typ mit herangezogen.

3. Maximum von drei:

```
int max3 (int a, int b, int c) {
    if (a>b)
        if (a>c)
            return a;
        else // c>=a, a>b
            return c;
    else // b>=a
        if (b>c)
            return b;
        else // c>=b, b>=a
            return c;
}
```

4. GGT–Berechnung für positive Argumente:

```
int ggt (int a, int b) {
  while (a != b) {
     if (a>b) a = a-b;
     if (b>a) b = b-a;
  }
  return a;
}
```

5. KGV–Berechnung:
GGT und KGV stehen in folgender Beziehung: $kgv(a,b) = \frac{ab}{ggt(a,b)}$. Damit wird die KGV–Berechnung einfach:

```
int kgv (int a, int b) {
   return (a*b / ggt (a,b));
}
```

6. Festellen, ob eine Angestellte einer anderen untergeben ist:

```
typedef int PersonalNr; //Nr 0 ist ungueltig!

struct Angestellt {
   string     name;
   PersonalNr nr;
   PersonalNr chef;
   PersonalNr untergeben[20];
};

typedef Angestellt Firma[100];

bool ist_untergeben (PersonalNr a, PersonalNr b, Firma f){
  for (int i=0; i<100; i++){
    if (f[i].nr == b)
      for (int j=0; j<20; j++)
        if (f[i].untergeben[j] == a)
          return true;
  }
  return false;
}
```

7. Das Gleiche als Methode:

```
typedef int PersonalNr; //Nr 0 ist ungueltig!

struct Angestellt {
   string     name;
   PersonalNr nr;
   PersonalNr chef;
   PersonalNr untergeben[20];
};

struct Firma {
   Angestellt angestellt [100];
   string     name (PersonalNr);
   bool       ist_untergeben (PersonalNr, PersonalNr);
};

string Firma::name (PersonalNr nr) {
    for (int i=0; i<100; i++)
        if (angestellt[i].nr==nr)
            return angestellt[i].name;
    return "";
}

bool Firma::ist_untergeben (PersonalNr a, PersonalNr b){
  for (int i=0; i<100; i++) {
    if (angestellt[i].nr == b)
      for (int j=0; j<20; j++)
        if (angestellt[i].untergeben[j] == a)
            return true;
  }
  return false;
}
```

Aufgabe 3

```
int f (int n) {
    int s = 0;
    for (int i=0; i<=n; i++) // INV: s = Summe aller 2*(i+1) von 0..(i-1)
        s += 2*(i+1);
    return s;
}
```

Aufgabe 4

Keine Lösung.

Aufgabe 5

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

using namespace std;

string ziffern (unsigned int x) {
  string res = "";
  while (x > 0) {
    res = char('0'+ x%10) + res;
    x = x/10;
  }
  return res;
}

unsigned int zifferQuer (string s) {
  unsigned int res=0;
```

```
  for (int i=0; i<s.length(); ++i)
    res = res + s.at(i)-'0';
  return res;
}

unsigned int quer (unsigned int x) {
  return zifferQuer( ziffern ( x ) );
}

int main () {
  unsigned int x;
  cin >> x;
  cout << quer(x) << endl;
}
```

8 Programmstatik und Programmdynamik

8.1 Funktionsaufrufe: Funktionsinstanz, Parameter, Rückgabewert

Aktuelle Parameter

Am Anfang jeder Abarbeitung einer Funktion steht der *Aufruf*; Beispiel:

```
a = 2; b = 3; c = min (a, b+2);
```

Hier werden zunächst die Werte der *aktuellen Parameter* `a` und `b+2` bestimmt. Genau gesagt: die r–Werte der Ausdrücke, die als aktuelle Parameter auftauchen, werden bestimmt und *in die Funktion hinein transferiert*. Diesen Transfer nennt man *Parameterübergabe*.

Formale Parameter

Im Kopf der Funktionsdefinition sind die *formalen Parameter* zu finden:

```
int min (int x, int y) {...}
```

Die formalen Parameter sind hier `x` und `y`.

Aufruf: Instanz erzeugen und Parameterübergabe

Nach dem Aufruf wird eine *Instanz* der Funktionsdefinition erzeugt. Die Erzeugung einer Funktionsinstanz besteht im wesentlichen darin, dass neue Variablen mit den Namen der formalen Parameter angelegt und mit den Werten der aktuellen Parameter initialisiert werden.

Hier im Beispiel werden also zwei neue Variablen angelegt. Sie heißen `x` und `y` und ihr Wert ist 2 bzw. 5. `x` und `y` sind nur innerhalb des Körpers von `min` gültig.

Achtung: formale und aktuelle Parameter dürfen nicht verwechselt werden! Die formalen Parameter sind immer nur Platzhalter. An der Aufrufstelle wird festgelegt, mit welchen Werten die Funktion arbeitet.

Funktionsinstanz

Die Funktion – genauer gesagt die Funktionsinstanz – wird jetzt aktiviert. Die *Funktionsinstanz* besteht aus der Funktionsdefinition und den beiden neuen Variablen `x` und `y` die für die formalen Parameter erzeugt und dann mit den Werten der aktuellen Parameter initialisiert wurden.

Nehmen wir an die Funktionsdefinition sei:

```
int min (int x, int y) { if (x<y) return x; else return y; }
```

und diese Funktion werde mit den aktuellen Parametern 2 und 3 aufgerufen . Die Funktionsinstanz ist dann:

Definition	`int min (int x, int y) { if (x<y) return x; else return y; }`
nächste Anweisung	`if (..`
`x`	2
`y`	5

Zu einer Funktionsinstanz gehört auch ein Verweis auf die als nächste auszuführende Anweisung. Am Anfang ist es natürlich die erste Anweisung im Funktionskörper. Mit jedem Funktionsaufruf wird eine neue Instanz der Funktion mit neuen Instanzen der formalen Parameter erzeugt – auch dann wenn die gleiche Funktion mehrfach aufgerufen wird. (Siehe Abbildung 24)

Rückgabewert

Die Funktion (genauer die Funktionsinstanz) wird jetzt solange abgearbeitet, bis eine `return`–Anweisung angetroffen wird. So wie am Anfang die Werte der aktuellen Parameter bestimmt und in die Funktion hinein transferiert wurden, so wird jetzt der Wert des Ausdrucks in der `return`–Anweisung bestimmt und aus der Funktion *heraus an die Aufrufstelle zurück transferiert*. Der zurück transportierte Wert nimmt an der Aufrufstelle dann die Stelle des Funktionsaufrufs ein. Der Funktionsaufruf ist ein Ausdruck, er hat einen Wert und dieser Wert ist der per return zurückgegebene Wert.

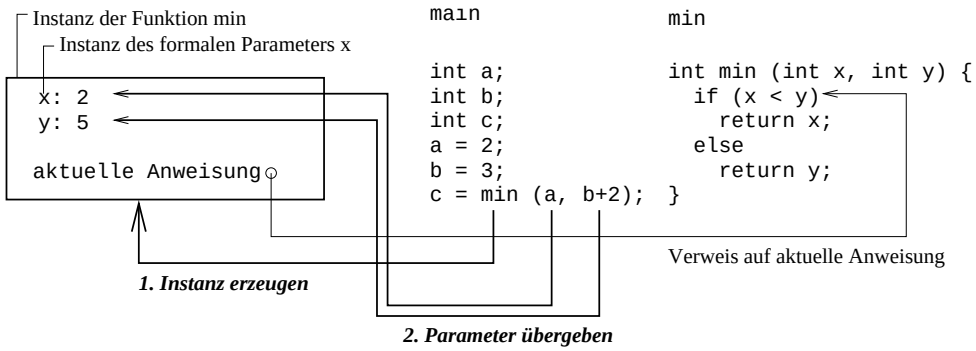


Abbildung 24: Aktivierung einer Funktion

Rückgabewert und Ausgabewert nicht verwechseln!

Die Rückgabe einer Funktion darf nicht mit einer Ausgabe verwechselt werden! Beispiel:

```
//Wert-Rueckgabe
int min1 (int x, int y) {
  if (x < y)
    return x;
  else
    return y;
}
```

```
//Wert-Ausgabe
void min2 (int x, int y) {
  if (x < y)
    cout << x;
  else
    cout << y
}
```

`return` bewegt einen Wert aus der Funktion an ihre Aufrufstelle. `cout` gibt einen Wert an das Ausgabemedium (i.d.R. den Bildschirm) und verändert dabei dessen Zustand. `cout` sorgt beispielsweise dafür, dass sich die Zahl und Position der leuchtenden Punkte auf dem Bildschirm verändern. Die Wirkung von `return` dagegen ist auf die "Innereien" des Programms beschränkt und lässt sich von außen nicht direkt beobachten.

Nach einem `cout` wird die Funktion fortgesetzt. `return` beendet den Funktionsaufruf sofort.

Formale Parameter und aktuelle Parameter

Parameter und Variablen als aktuelle Parameter werden immer unterschieden. Wenn in einer Funktion der formale Parameter verändert wird, dann hat das keinen Einfluss auf die Variablen, die als aktuelle Parameter verwendet wurden. Beispiel:

```
#include <iostream>

using namespace std;

int f (int, int);

int main () {
  int a[3] = {1,2,3};
  int i    = 0;
  cout << f(i, a[i]) //Aufruf
       << endl;
}

int f (int x, int y) {
  ++x;               //x wird veraendert, aber nicht i
  return y;          //y hat immer noch den originalen Wert von a[0]
}
```

Dieses Programm erzeugt die Ausgabe "1".

Datentyp formaler Parameter und des Ergebnisses

Als Typen der formalen Parameter und Funktionsergebnisse können beliebige Typen außer Feldtypen verwendet werden. Bei Feldern – die keine "vernünftigen" l– und r–Werte haben – gelten Sonderregeln, auf die wir später zu

sprechen kommen werden.

Beispiel:

```
#include <iostream>

using namespace std;

struct Komplex {
  float r;
  float i;
};

Komplex add (Komplex a1, Komplex a2) {
  Komplex res;         // res ist eine lokale Variable vom Typ Komplex
  res.r = a1.r + a2.r;
  res.i = a1.i + a2.i;
  return res;          // der Wert von res wird zurueck gegeben
}

int main () {
  Komplex k1 = {2, 3},
          k2 = {7, 9},
          k3;

  k3 = add (k1, k2);
  cout << "k3: (" << k3.r << ","
                  << k3.i << ")\n";
}
```

Hier werden Werte vom Typ Komplex übergeben und auch als Ergebnis zurück geliefert.

8.2 Sichtbarkeit und Lebensdauer von Parametern und Variablen

Parameter und Variable

Die formalen Parameter sind nur innerhalb der Funktion (frei oder Methode) gültig zu der sie gehören. Im folgenden Beispiel hat der Bezeichner `x` zwei völlig verschiedene Bedeutungen. Innerhalb von `f` bezeichnet `x` einen formalen Parameter, innerhalb von `main` dagegen eine Variable.

```
#include <iostream>

using namespace std;

int f (int);

int main () {                          //- die Variable x --
  int x = 3;                           //
  cout << "Funktionsergebnis: " f(x)   //
       << " Wert von x: " << x << endl;//
}                                      //-------------------

int f (int x) {                        //- der Parameter x -
  ++x;                                 //
  return x;                            //
}                                      //-------------------
```

Dieses Programm erzeugt die Ausgabe

```
Funktionsergebnis: 4 Wert von x: 3.
```

Sichtbarkeit

In diesem Programm gibt es also zwei Dinge mit gleichem Namen:

1. die mit `int x;` definierte Variable in `main`;
2. den formalen Parameter `(int x)` von `f`.

Wird innerhalb von `f` der Name `x` gebraucht, dann ist der formale Parameter gemeint; wird dagegen der Name `x` innerhalb von `main` verwendet, dann ist die Variable gemeint.

Unter dem Namen `x` ist in einem Bereich des Programms (Funktion `f`) der formale Parameter *sichtbar* und in einem anderen (Funktion `main`) ist eine lokale Variable *sichtbar*.

Sichtbarkeitsregeln

Wenn in einem Programm mehrere Dinge existieren, die den gleichen Namen tragen, dann muss es Regeln geben, welches der Dinge gemeint ist, wenn der Name gebraucht wird. Diese Regeln werden *Sichtbarkeitsregeln* genannt.

In allen relevanten Programmiersprachen kann ein Name sich innerhalb eines Programms auf eine Vielzahl von verschiedenen Dingen beziehen. An jeder Stelle des Programms müssen dann die Sichtbarkeitsregeln der Sprache exakt festlegen, welches dieser Dinge *hier* gemeint ist, bzw. *hier sichtbar* ist.

Gültigkeitsbereich

Ein formaler Parameter einer Funktion ist nur innerhalb der Funktion gültig. Die Funktion ist sein *Gültigkeitsbereich*. Dementsprechend ist der Gültigkeitsbereich einer Variablen die Funktion in der sie definiert wurde.

Lebensdauer

Jedes vom Programm erzeugte Objekt hat auch eine *Lebensdauer*, das ist die Zeitspanne zwischen seiner Erzeugung und seiner Vernichtung. Formale Parameter entstehen mit der Erzeugung der Instanz der Funktion zu der sie gehören und werden auch mit ihr wieder vernichtet. Variablen werden bei der Abarbeitung ihrer Definition erzeugt und verschwinden wenn die Funktionsinstanz, in der sie definiert wurden, beendet wird. (Siehe Abbildung 25)

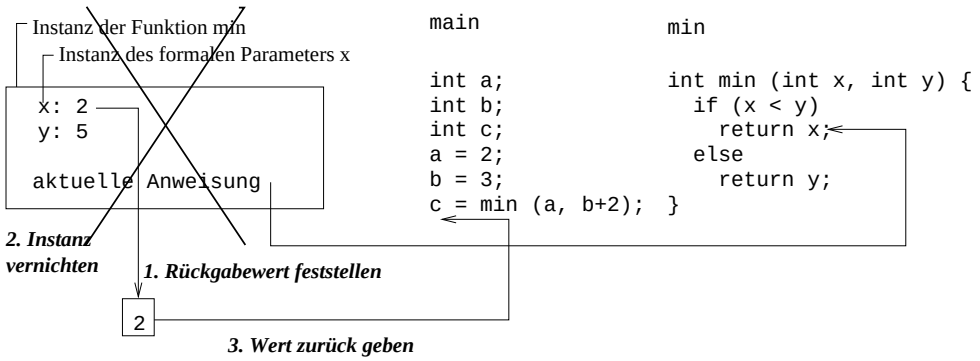


Abbildung 25: Beendigung einer Funktion

8.3 Lokale Variable, Überdeckungsregel

Lokale Variable

In jeder Funktion können Variablen definiert werden – nicht nur in `main`. Sie werden *lokale Variablen* genannt. Lokale Variablen sind stets – auch bei gleichem Namen – von den Variablen einer anderen Funktion zu unterscheiden. In den bisherigen Beispielen waren die Variablen des Programms stets die *lokalen Variablen* der Funktion `main`!

Im folgenden Beispiel gibt es zwei Funktionen `main` und `sum` mit lokalen Variablen.

```
#include <iostream>

using namespace std;
```

```
int sum (int, int);

int main () {
  int a[3] = {1,20,300}; //lokale Variablen von main
  int i    = 1;
  cout << sum(i, a[i]) << ", " << i << endl;
}

int sum (int x, int y) {
  int i;                 //lokale Variable von sum
  i = x + y;
  return i;
}
```

Das Programm gibt "21, 1" aus. Es enthält zwei Funktionen `sum` und `main` in denen jeweils eine lokale Variable mit Namen `i` deklariert ist. Beide haben (außer dem Namen und dem Typ) *nichts gemeinsam*, es sind zwei verschiedene Variablen. Sie könnten auch völlig unterschiedliche Typen haben.

Verwendung und Deklaration eines Namens

Jede Verwendung eines Namens ordnet der Compiler "der nächsten" Deklaration[24] des Namens zu. Die "nächste" Deklaration ist die erste vor dem Vorkommen des Namens *nach außen hin*:

1. eine Deklaration in der gleichen zusammengesetzten Anweisung;
2. eine Deklaration der Laufvariablen einer `for`–Schleife;
3. eine Variablendeklaration in der gleichen Funktion;
4. ein formaler Parameter der gleichen Funktion;
5. die Funktionsdeklaration.

Ein Beispiel ist:

```
int f1(int);
int f2(float);
int f3();
int f4(int);

int f1(int p) {
  float x;
  ...
  ... x ...  // float x von f1
  ... p ...  // formaler Parameter p von f1
}

int f2(float p) {
  int x;
  ...
  ... x ...  // int x von f2
  ... p ...  // formaler Parameter p von f2
}

int f3() {
  int f1;
  ...
  ... x ...  FEHLER !
  ... f1 ... // int f1 von f3
  ... f2 ... // Funktion f2
}

int f4(int p) {
```

[24] oder der nächsten Definition, wenn die Definition gleichzeitig die Deklaration ist

```
  float p;
  ...
  ... p ...  // float p von f4
  for (int p=0; p<10; ++p){
    ... p .. // int p der Schleife
  }
  ... p ...  // float p von f4
}
```

Überdeckungsregel

Eine "nähere" Deklaration überdeckt eine fernere (macht sie unsichtbar). Im Beispiel oben überdeckt in `f4` die lokale Deklaration `float p` den formalen Parameter `p` und in `f3` überdeckt `int f1` die Funktion `int f1(int)`.

Die Überdeckungsregel sorgt dafür, dass eine Deklaration in ihrem Gültigkeitsbereich "Löcher der Sichtbarkeit" haben kann. Beispielsweise ist die lokale Variable `p` im gesamten Bereich von `f4` sichtbar, aber innerhalb der Schleife ist sie durch die Laufvariable – die ebenfalls `p` heißt – abgedeckt und damit dort "unsichtbar".

Die Überdeckungsregel ist ein wesentlicher Teil der Sichtbarkeitsregeln von C++. Sehr viele wichtige Programmiersprachen haben äquivalente Sichtbarkeitsregeln.

Sichtbarkeitsregeln und Überladung

Weiter oben haben wir erwähnt, dass es möglich ist, zwei Funktionen (frei, oder Methode) den gleichen Namen zu geben, vorausgesetzt sie unterscheiden sich in ihren Argumenten (Anzahl, Typ). Diese "Überladung" genannte Fähigkeit von C++ hat nichts mit den Sichtbarkeitsregeln zu tun. Die Information über die Argumente wird quasi wie ein Bestandteil des Namens behandelt. Unterschiedliche Funktionen mit dem gleichen (überladenen) Namen tangieren sich gegenseitig nicht. Sie können aber insgesamt durch eine Variable oder einen Parameter mit dem gleichen Namen abgedeckt werden. Beispiel:

```
#include <iostream>

using namespace std;

float  f (int);    // f-int
float  f (float);  // f-float

int main () {
  int   i = 3;
  float d = 3.0;

  cout << "f-int:   " << f (i) << endl; // OK: f-int
  cout << "f-float: " << f (d) << endl; // OK: f-float

  for (int f=0; f<10; ++f)  // Lokales Schleifen-f
    cout << f (3) << endl;  // FEHLER: beide Funktionen sind ueberdeckt !
}

float f (int x)   { return x/2; }
float f (float x) { return x/2; }
```

8.4 Globale Variablen und Prozeduren

Globale Variable

Eine *globale Variable* ist eine Variable, die auf Programmebene definiert wurde, d.h. ihre Definition ist außerhalb von allen Funktionen. Beispiel:

```
int   x;        // x, a: globale Variablen
float a[10];

int f (...);
```

```
...
int main () { ... x = a[1]; ... } // Verwendung in main

int f (...) { ... a[x] = 5; ... } // Verwendung in f
```

Lokale Variablen sind nur in ihrer Funktion sichtbar. Auf eine globale Variable kann dagegen von *jeder Funktion aus* zugegriffen werden. Ein etwas ausführlicheres Beispiel ist:

```
int a = 1;     // Globale Variable mit Initialisierung

int f (int);

int main () {
  int u;
  ++a;
  u = f (a);
  cout << " u: " << u << endl;
  cout << " a: " << a << endl;
}

int f (int x) {
  a++;
  return a+x;
}
```

Hier wird sowohl aus `main` als auch aus `f` auf die globale Variable `a` zugegriffen. Zuerst wird in `main` der Wert von `a` auf 2 erhöht. Dann wird 2 an `f` übergeben und dient dort als Initialwert von `x`. Weiter in `f` wird `a` auf 3 erhöht und die Summe des Wertes von `x` (2) und von `a` (3) als Funktionswert zurückgegeben. Dann erfolgt die Ausgabe:

```
u: 5
a: 3
```

Initialisierung globaler Variablen

Globale Variablen können natürlich wie alle anderen auch initialisiert werden. Entsprechende Beispiele haben wir schon gesehen. Das Besondere bei globalen Variablen ist, dass nicht explizit initialisierte globale Variablen *implizit mit 0 initialisiert* werden. Das gilt nur für globale Variablen! Der Wert nicht initialisierter lokaler Variablen ist undefiniert, es kann jeder beliebige zufällige Wert sein.

```
int a;      // implizite Initialisierung => int a = 0;

int main () {
  int b;    // keine implizite Initialisierung,
            // Wert von b undefiniert (zufaellig)
  ...
}
```

Trotz der impliziten Initialisierung wird empfohlen auch globale Variablen immer *explizit* zu initialisieren, wenn eine Initialisierung notwendig ist.

Der Datentyp `void`

Funktionen können mit dem Ergebnistyp `void` definiert werden. `void` ist ein Pseudotyp und steht dafür, dass nichts zurückgegeben wird. `void f (...);` gibt also keinen Wert vom Typ `void` zurück sondern gar nichts. Es ist eine Funktion ohne Ergebnis. Ergebnislose Funktionen werden auch *Prozeduren* genannt. Prozeduren werden wie Anweisungen (statt wie Ausdrücke sonst) verwendet. Beispiel:

```
#include <iostream>

using namespace std;

float glob;            //globale Variable, implizit: = 0.0
```

```
float b[10];           // implizit: = {0.0, 0.0, 0.0, 0.0, 0.0,
                       //              0.0, 0.0, 0.0, 0.0, 0.0};


void fahrenheit ();

int main () {
  float a[10] = {0.0, 1.0, 2.0, 3.0, 4.0,
                 5.0, 6.0, 7.0, 8.0, 9.0};

  for (int i=0; i<10; ++i) {
    glob = a[i];
    fahrenheit (); //Aufruf: eine Anweisung, veraendert den Wert von glob
    b[i] = glob;
  }
}
void fahrenheit () {
  glob = (glob * 9.0);  //Zugriff auf die globale Variable glob
  glob = glob/5.0 + 32;
}
```

Die Funktion `fahrenheit` hat keinen Parameter und kein Ergebnis. Sie kommuniziert statt dessen mit ihrer Umwelt über die globale Variable `glob`, deren Wert gelesen und verändert wird.

Seiteneffekt: Wirkung einer Funktion auf globale Variablen

Eine Funktion sollte im Allgemeinen über Parameter und Ergebnis mit ihrer Umwelt kommunizieren. Die Veränderung globaler Variablen wird darum als *Seiteneffekt* bezeichnet. Eine Prozedur funktioniert nur über ihre Seiteneffekte. Eine Funktion kann aber auch gleichzeitig Parameter, Werte und Seiteneffekte haben:

```
int g;
int f (int x) {  // Parameter
  g = g + x;     // Seiteneffekt
  return 2*g;    // Wert
}
```

So etwas wird allgemein nicht empfohlen. Funktionen sollten möglichst klar und einfach sein. Einer Funktion mit Parametern und Ergebnis sieht man sofort an, welche Werte hinein gehen und welche herauskommen. Die Seiteneffekte einer Funktion erkennt man aber nur nach eingehender Analyse des Codes der Funktion.

Seiteneffekte sollen also nach Möglichkeit vermieden werden. Wenn aus Werten neue Werte zu berechnen sind, dann sollte in jedem Fall eine seiteneffektfreie Funktion verwendet werden. Temperaturen von Celsius in Fahrenheit umrechnen ist ein Beispiel für diese Situation. Statt wie oben sollte man also besser definieren:

```
...
int main () {
  ...
  for (int i=0; i<10; ++i)
    b[i] = fahrenheit (a[i]);
}
float fahrenheit (float x) {
  float t;
  t = (x * 9.0);  //Zugriff auf den Parameter
  return t/5.0 + 32;
}
```

Diese Version von `fahrenheit` hat keine "versteckte" Wirkung nach außen. Ihr Verhalten kann man vollständig am Funktionskopf erkennen: sie ist seiteneffektfrei.

8.5 Funktionen: Sichtbarkeit und Lebensdauer

Programmstatik

Mit "statisch" bezeichnet man alle Phänomene und Konzepte, die sich aus dem Text des Programms ergeben und nicht erst zu dessen Laufzeit beobachtet werden können. Die wesentlichen Begriffe zur Programmstatik wurden

bereits eingeführt:

- Sichtbarkeitsregel:
  Auf welche Deklaration bezieht sich ein Name an einer Verwendungsstelle. (Welche Definition ist an der Verwendungsstelle "sichtbar")
- Gültigkeitsbereich:
  Der Bereich des Programmtextes in dem eine Deklaration gültig ist. Der Sichtbarkeitsbereich einer Deklaration ist gleich deren Gültigkeitsbereich, es sei denn eine andere Deklaration hat mit einer Überdeckung eine "Lücke" im Sichtbarkeitsbereich verursacht.
- Überdeckungsregel:
  Lokale Deklarationen überdecken globale Deklarationen.

Diese Regeln definieren, wie *freien Bezeichnern* eine Deklaration zugeordnet wird, also Bezeichnern die nicht lokal in dem Gültigkeitsbereich deklariert wurden, in dem sie verwendet werden. Die entsprechende Regel war "von innen nach außen".

Variablen können global (im globalen Gültigkeitsbereich) oder lokal (im Gültigkeitsbereich einer Funktion) deklariert werden. Funktionen dagegen dürfen *nur global* (im globalen Gültigkeitsbereich) deklariert werden. Eine Funktionsdefinition kann darum zwar beliebige Variablendefinitionen aber niemals eine Funktionsdefinition enthalten.

Eine weitere Konsequenz der Beschränkung von Funktionsdefinitionen auf den globalen Gültigkeitsbereich ist die Tatsache, dass freie Bezeichner einer Funktion immer an eine Deklaration im globalen Gültigkeitsbereich gebunden sein müssen. In der Definition der Funktion `f`

```
int f ( int x) { int t=2; return t*y + g (x); }
```

sind `x` und `t` "gebunden": es gibt lokale Deklarationen für sie. `y` und `g` dagegen sind frei. In welchem Programm auch immer die Definition von `f` auftaucht, `y` und `g` müssen stets im globalen Gültigkeitsbereich des Programms zu finden sein.

Programmdynamik

Sichtbarkeit und Gültigkeitsbereich sind statische Konzepte, mit denen im Programmtext zu der Verwendungsstelle eines Bezeichners die zugehörige Deklaration gefunden werden kann.

Zur *Laufzeit* des Programms ("dynamisch" !) werden die "wirklichen Dinge", die Variablen, Funktionsinstanzen, etc. erzeugt, auf die sich die im Programmtext vorkommenden Namen (Bezeichner) beziehen. Nur bei äußerst einfachen Programmen wird dieses "Arbeitsmaterial" des Programms bei dessen Start erzeugt und existiert bis zu dessen Ende. Bei normalen Programmen herrscht ein ständiges Kommen und Gehen: Dinge werden erzeugt und wieder vernichtet, einiges lebt sehr lange anderes nur für eine kurze Zeit.

Lebensdauer

Die Lebensdauer von Variablen, Parametern und Funktionsinstanzen ist unterschiedlich und wie folgt definiert:

- *Globale Variable*
  Für jede globale Variable wird zu Beginn des Programms ein *globales Objekt* erzeugt, das bis zum Ende des Programms weiter existiert.
- *Funktionen*
  Bei jedem Funktionsaufruf wird eine Instanz der aufgerufenen Funktion erzeugt. Sie existiert dann weiter bis zur Ausführung der ersten `return`–Anweisung bzw. bis zur Ausführung der letzten Anweisung der Funktion.
- *Lokale Variablen*
  Für jede lokale Variable einer Funktion wird mit jeder Funktionsinstanz ein *automatisches Objekt* erzeugt, das mit der Funktionsinstanz wieder vernichtet wird.
- *Formale Parameter*
  Für die formalen Parameter werden, genau wie bei den lokalen Variablen, mit der Instanz Objekte erzeugt. Sie existieren für die Dauer der Funktionsaktivierung (Instanz) zu der sie gehören.

Die Objekte, die für lokale Variablen erzeugt werden, nennt man auch *automatische Objekte*, da sie "automatisch" mit der Instanz zu der sie gehören erzeugt und wieder vernichtet werden. Erzeugung und Vernichtung ist dabei als Reservierung eines Speicherbereichs und dessen Freigabe für andere Zwecke zu verstehen.[25]

Statische lokale Variablen

Für eine lokale Variable einer Funktion wird also bei jedem Aufruf aufs neue Speicherplatz reserviert und am Ende der Funktion wieder freigegeben. Es ist darum unmöglich in einer Funktion einen Wert von Aufruf zu Aufruf aufzuheben. Eine Funktion, die die Zahl ihrer Aufrufe mitzählt muss darum eine globale Variable verwenden. Die Funktion

```
int wie_oft () { int z = 0; ++z; return z;}
```

wird immer 1 als Ergebnis liefern. Lässt man die Initialisierung weg:

```
int wie_oft () { int z; ++z; return z;}
```

dann wird immer noch nicht gezählt. Man muss statt dessen mit beliebigen zufälligen Ergebnissen rechnen. Nur

```
int z = 0;

int wie_oft () {
  ++z;
  return z;
}
```

arbeitet wie gewünscht. Das Problem hier ist aber, dass `z` als globale Variable von jeder Funktion nach Belieben gelesen und verändert werden kann. Mit der Definition einer *statischen lokalen Variablen* können beide Ziele erreicht werden:

```
int wie_oft () {
  static int z = 0;  // statische lokale Variable
  ++z;
  return z;
}
```

`z` hat die Lebensdauer einer globalen Variablen und bleibt darum von Aufruf zu Aufruf mit seinem aktuellen Wert erhalten. Gleichzeitig ist die Sichtbarkeit von `z` auf die Funktion `wie_oft` beschränkt. Ein Zugriff von außen ist unmöglich.

Für statische lokale Variablen wird bei Programmbeginn ein *statisches lokales Objekt* erzeugt. Das Objekt bleibt bis zum Programmende bestehen. Statische lokale Objekte sind also im Gegensatz zu den automatischen lokalen Objekten nicht Bestandteil der Funktionsinstanzen. Die Initialisierung wird nur beim ersten Aufruf ausgeführt, ansonsten bleibt der Wert von Aufruf zu Aufruf erhalten.

8.6 Methoden: Sichtbarkeit und Lebensdauer

Gültigkeitsbereich eines Verbunds

Verbunde haben ihre eigenen Sichtbarkeitsregeln. Betrachten wir dazu kurz ein Beispiel:

```
int  x;      // globale Variable x
char z;      // globale Variable z
struct S {   // Beginn Gueltigkeitsbereich von S
  char x;    // <- Verbundkomponente x
  int  y;    //
};           // Ende Gueltigkeitsbereich von S
S s1;
int main () {
  int z;     // ueberdeckt globales z
  s1.x = z;  // OK: s1.x ist S::x, z ist lokales z
  s1.x = x;  // OK: s1.x ist S::x, x ist globales x
  s1.z = x;  // FEHLER: S::z gibt es nicht
}
```

[25] Der Begriff automatisch ist ein Erbe von C an C++. Lokale Variablen haben dort die Speicherklasse "automatisch".

Es ist klar, dass x in

```
s1.x = x;
```

zwei unterschiedliche Bindungen (Bedeutungen) hat. In s1.x ist natürlich die x–Komponente von s1 gemeint. Das andere x bezieht sich dagegen auf die globale Variable.

Verbundtypen haben ihren eigenen Gültigkeitsbereich. Mit dem Punktoperator in s1.x wird der Gültigkeitsbereich von S "aufgemacht" und der nachfolgende Bezeichner x wird in diesem Bereich gesucht und nicht im Bereich von main oder dem globalen Bereich. Es ist auch klar, dass die Suche nach Bezeichnern hinter dem Punktoperator auf den Bereich von S beschränkt bleibt.

```
s1.z = x;
```

ist nicht korrekt, da z nicht im Bereich von S zu finden ist. Nach ihm wird weder in main noch im globalen Bereich gesucht.

Der Bereichsauflösungsoperator

In den Kommentaren des letzten Beispiels haben wir den (Gültigkeits–) Bereichsauflösungsoperator "::" verwendet. Mit S::x ist das x aus dem Gültigkeitsbereich von S gemeint. Der ::–Operator wird bei der Definition von Methoden benötigt:

```
struct S {
  char x;
  int  y;                          // Deklaration von S::f,
  int  f (int);                    // S:: nicht notwendig, wir sind in S
};
                                   // Definition von S::f,
int S::f(int x) { return x + y; } // S:: notwendig, wir sind nicht in S
```

Freie Bezeichner in Methoden

Freie Bezeichner einer Funktion beziehen sich auf *globale* Deklarationen. Bei Methoden wird bei der Suche nach der gültigen Deklaration eines Bezeichners ebenfalls von innen nach außen vorgegangen. Allerdings wird vor dem globalen Gültigkeitsbereich noch der des Verbundes abgesucht. Die Überdeckungsregel gilt dabei ebenfalls:

- lokale Variablen überdecken Parameter
- Parameter überdecken Verbundkomponenten
- Verbundkomponenten überdecken globale Namen

Beispiel:

```
int   x;
short y;
char  z;

struct S {
  char x;
  int  y;
  int  f (int);
};

int S::f (int x) { // x deckt S::x ab
  return x    // <- Parameter x,
       + y    // <- S::y
       + z;   // <- globales z (::z)
}

int f (int x) {    // x deckt globales x ab
  return x    // <- Parameter x,
       + y    // <- globales y (::y)
       + z;   // <- globales z (::z)
}
```

```
int main () {
  S s;
  cout << f (1) << endl;   // globales f, da es kein lokales f gibt
  cout << s.f (1) << endl; // S::f, der Punkt--Operator "."
}                          //       macht den Bereich von S auf
```

Hier ist `f` als Methode von `S` und als freie Funktion definiert. Die beiden Definitionen befinden sich in unterschiedlichen Bereichen und kollidieren darum nicht miteinander. Bei `f` geht die Suche nach einer Deklaration aus der Funktion direkt in den globalen Bereich, `y` wird als das globale `y` identifiziert. Bei `S::f` geht die Suche über `S` und `y` wird als `S::y` erkannt.

Verbundkomponenten in Methoden

Wird in einer freien Funktion auf eine globale Variable Bezug genommen, dann ist diese eindeutig bestimmt und für jeden Aufruf die gleiche. Eine Verbundkomponente in einer Methode kann sich dagegen auf die Komponente mit dem entsprechenden Namen einer beliebigen, wechselnden – und an der Definitionsstelle unbekannten – Instanz des Verbundes beziehen.

```
int ag;
Struct S {
  int as;
  int f (int);
}

int f (int x)    { ... ag ... } // ag ist für jeden Aufruf von f das gleiche (::ag)

int S::f (int x) { ... as ... } // as ist unbestimmt
```

Zu der Definition `int ag;` gibt es zur Laufzeit genau ein Objekt. Zur Definition `int as;` dagegen gibt es zur Laufzeit beliebig viele Objekte. `as` in `S::f` meint immer das `as`, welches zur Instanz des Verbundes gehört, für die `S::f` aktiviert wurde. Ein etwas komplexeres Beispiel ist:

```
#include <iostream>

using namespace std;

int i = 1000;     // globales i (::i)

struct S {
  int i;          // S::i
  int f (int);
};
int main () {
  S s1, s2;        // zwei Verbunde: zwei Verbundelemente i
  int   i = -100;  // lokales i in main
  s1.i = 1;
  s2.i = 10;
  cout << s1.f(4) << ", "   // gibt  5 aus (4 + S::i(von s1))
       << s2.f(4) << endl;  // gibt 14 aus (4 + S::i(von s2))
}
int S::f (int n) {
  return n     // n ist lokal definiert
         + i;  // i -> S::i
}
```

Genau wie jeder Verbund vom Typ `S` seine Komponente `i` hat, so hat er auch seine private Methode `f`. Es gibt zwar nur eine einzige Definition von `S::f`, aber so viele Varianten von `S::f`, wie es Variablen (Objekte) vom Typ `S` gibt. Diese Varianten unterscheiden sich allerdings nur in der genauen Bedeutung des freien Bezeichners `i` in der Definition von `S::f`.

8.7 Wert– und Referenzparameter

Wertparameter

Die Parameter der Funktionen in den bisherigen Beispielen waren alle *Wertparameter*. Wertparameter haben ihren Namen daher, dass sie ein Mechanismus sind um *Werte* in eine Funktion (frei oder Methode) zu transferieren. Vor der Aktivierung der Funktionsinstanz werden alle aktuellen Parameter berechnet und die Werte sind die Initialwerte der formalen Parameter. Ein Rücktransport der Werte der formalen Parameter an die Aufrufstelle findet nicht statt. Ein Beispiel für eine Funktion mit Wertparametern ist:

```
void no_swap (int x, int y) {
  int t = x;
  x = y;
  y = t;
}
int main () {
  int a = 3, b = 4;
  no_swap (a, b);     // == no_swap (3, 4)
  ...                 // Aufruf ohne Wirkung
}
```

Der Aufruf der Funktion `no_swap` hat keinerlei Wirkung nach außen. Insbesondere werden die Werte von `x` und `y` nicht ausgetauscht.

Referenzparameter

Eine Funktion mit Referenzparametern kann dagegen benutzt werden, um die Werte von Variablen zu vertauschen. Man beachte die Kennzeichnung der formalen Parameter mit "&":

```
void swap (int &x, int &y) {
  int t = x;
  x = y;
  y = t;
}
int main () {
  int a = 3, b = 4;
  swap (a, b);        // != swap (3, 4)
  ...                 // jetzt a == 4, b == 3
}
```

Referenzparameter werden mit einem anderen Mechanismus übergeben. Bei Wertparametern werden *die Werte der aktuellen Parameter* in die Funktion bewegt. Bei Referenzparametern werden dagegen die *aktuellen Parameter selbst* in die Funktion transferiert. Alle Aktionen auf den formalen Parametern sind darum Aktionen auf den aktuellen Parametern. Wenn also in

```
no_swap (int x, int y) {...} // Funktion mit Wertparametern
```

die Anweisung

```
x = y; // x,y formale Parameter
```

ausgeführt wird, dann wird ein lokales Objekt von `no_swap` verändert. Die gleiche Anweisung in

```
swap (int & x, int & y) {...} // Funktion mit Referenzparametern
```

ist eine Zuweisung an die Variable `a`, die als aktueller Parameter übergeben wurde. Um es knapp und exakt auszudrücken:

- Bei Wertparametern wird der *r–Wert* und
- bei Referenzparametern wird der *l–Wert*

des aktuellen Parameters berechnet und übergeben. Der r–Wert dient als Initialwert eines lokalen Objektes, auf das sich alle Operationen dann beziehen. Bei Referenzparametern dagegen beziehen sich alle Operationen direkt auf den l–Wert, d.h. auf die übergebene Variable.

Eine Funktion kann gleichzeitig Wert– und Referenzparameter haben:

```
void add (int x, int y, int &z) {
  z = x + y;
}
int main () {
  int a = 2,
      b = 3;
  add (a, b, a;); // a = a + b;
}
```

In `add` wird `z` als *Ergebnisparameter* verwendet. Ein Ergebnisparameter kann als Ersatz oder Alternative für einen Ergebnis– (`return`–) Wert benutzt werden. Will man etwa zwei Werte zurückgeben, dann kann der eine als Funktionsergebnis, der andere über einen Referenzparameter zurückgegeben werden. Beispiel:

```
#include <iostream>

using namespace std;

bool ggt (int a, int b, int &e) {
  if (a < 1 || b < 1) return false; //ggt-Berechnung nicht moeglich
  while (a != b)
    if (a > b) a = a-b;
    else       b = b-a;
  e = a;
  return true;
}
int main () {
  int a, b, c;
  cin >> a;
  cin >> b;
  if (ggt(a,b,c))
    cout << "GGT (" << a << "," << b << ") = " << c << endl;
  else
    cout << "Bitte positive Eingaben \n";
}
```

Man beachte, dass die Variablen `a` und `b` aus `main` durch `ggt` nicht verändert werden. Sie werden ja per Wertübergabe an die Funktion übergeben. Innerhalb von `ggt` kann man formale Wertparameter immer bedenkenlos verändern, die Änderung dringt ja nicht nach außen. Der Aufruf

```
ggt(a,b,a)
```

wäre natürlich auch möglich, er würde aber den Wert in `a` durch den berechneten ggt ersetzen.

r–Wert als Referenzparameter

Da bei der Referenzübergabe der l–Wert des aktuellen Parameters übergeben wird, kann man erwarten, dass dieser einen l–Wert haben muss; dass also ein Aufruf wie etwa

```
ggt (2, 4, 6); // FEHLER
swap (7, 8); // FEHLER
```

nicht erlaubt ist (`swap` und `ggt` mit Referenzparametern wie oben definiert).

Wird allerdings ein r–Wert als aktueller Parameter an einen *konstanten (!)* Referenzparameter übergeben, dann wird eine interne anonyme Variable (ein l–Wert) erzeugt und mit dem r–Wert des aktuellen Parameters initialisiert.

```
void f (int & x) { ... }        // <<-- nicht-konstanter Referenzparameter
void g (const int & x) { ... }  // <<-- konstanter Referenzparameter

int main () {
  f (2); // FEHLER
  g (2); // OK
}
```

Parameter von Methoden

Wert– und Referenzparameter werden bei Methoden völlig äquivalent zu Wert– und Referenzparametern bei freien Funktionen behandelt. Beispiel:

```
#include <iostream>

using namespace std;

struct Swapper {
  int t;
  void swap (int &, int &);
};

int main () {
  Swapper s;
  int i = 1, j= 2;
  s.swap (i,j);
  cout << i << ", " << j << endl; // Ausgabe: 2, 1
}

void Swapper::swap (int &i, int &j) {
  t = i;
  i = j;
  j = t;
}
```

Variablen mit einem Verbundtyp können natürlich auch als Referenzparameter an eine Funktion oder eine Methode übergeben werden. Beispiel:

```
struct Komplex {
  float r;
  float i;
};

void add (Komplex a1, Komplex a2, Komplex &res) {
  res.r = a1.r + a2.r;
  res.i = a1.i + a2.i;
}

int main () {
  Komplex k1 = {2, 3},
          k2 = {7, 9},
          k3;

  add (k1, k2, k3);
  cout << "k3: (" << k3.r << ","
                  << k3.i << ")\n";
}
```

Zusammenfassung: Transport von Werten in und aus Funktionen

Werte können auf verschiedene Arten in eine Funktion/Methode transportiert werden:

- als Wert eines Wertparameters,
- als Wert eines Referenzparameters,
- als Wert einer globalen Variablen,
- als Wert einer Komponente des zugehörigen Verbundes (nur bei Methoden).

Beispiel:

```
struct S {
  int x;
  void f (int, int &);
};

int g = 0;

void S::f(int wp, int & rp) {
  cout << wp << endl;          // Wertparameter
  cout << rp << endl;          // Referenzparameter
  cout <<  g << endl;          // globale Variable
  cout <<  x << endl;          // Verbundkomponente
}
```

In umgekehrter Richtung kann ein Wert aus einer Funktion/Methode gelangen als:

- als Rückgabewert,
- als Wert eines Referenzparameters,
- als Wert einer globalen Variablen,
- als Wert einer Komponente des zugehörigen Verbundes (nur bei Methoden).

Beispiel:

```
struct S {
  int x;
  int f (int &);
};

int g;

int S::f(int & rp) {
  rp = 0;                      // Referenzparameter
  g  = 1;                      // globale Variable g
  x  = 2;                      // Verbundkomponente S::x
  return 3;                    // Rueckgabewert
}
```

8.8 Felder als Parameter

Felder als Parameter

Felder können nicht per Wertübergabe in eine Funktion oder Methode transportiert werden. Das liegt daran, dass ihr r–Wert ein Zeiger auf das erste Element und nicht das Feld mit all seinen Elementen ist.[26] Felder werden darum nicht als Wertparameter übergeben.

Felder als Referenzparameter

Es ist ohne weiteres möglich, Felder per Referenz zu übergeben. Beispiel:

```
#include <iostream>

using namespace std;

typedef int A[10];

void f (A &x) {  // Feld mit benanntem Feldtyp als Referenzparameter
  for (int i=0; i<10; ++i)
```

[26] Da der r–Wert eines Feldes ein Zeiger auf das erste Element und nicht das Feld als ganzes ist, wird bei einer Funktion mit Wertparameter – z.B. `void f(int a[10]);` – nur dieser Zeiger – im Beispiel der Zeiger auf a[0] – übergeben. Wir werden uns damit später, nach der Betrachtung der Zeiger, näher beschäftigen.

```
    cout << x[i] << endl;
}

A a;
A b = {1,2,3,4,5,6,7,8,9,0};

int main () {
 f(a);
 f(b);
}
```

Bei der Referenzübergabe eines Feldes wird wie sonst auch eine Referenz auf die Feldvariable übergeben.

Auch wenn der Typ des Feldes nicht, wie im Beispiel oben, per `typedef` mit einem Namen versehen wird, kann das Feld als Referenzparameter übergeben werden:

```
...
void f (int (&x)[10]) {  // <- Feld mit anonymem Feldtyp als Referenzparameter
  for (int i=0; i<10; ++i)
    cout << x[i] << endl;
}
...
int a[10];
...
f(a);
```

Die runden Klammern um `&x` bei der Angabe des Referenzparameters in

```
void f (int (&x)[10])
```

sind notwendig, da die eckige Klammer stärker bindet als `&`. Ohne sie würde der Ausdruck als

```
void f (int & (x[10]))
```

interpretiert: als ein (nicht erlaubtes) Feld von Referenzen als Parameter.[27]

8.9 Namensräume

Namensraum

Ein Namensraum (engl. *namespace*) ist ein recht einfaches Konzept, mit dem Deklarationen und Definitionen in einem Programm zusammengefasst werden können. Beispiel:

```
namespace Suche {
  int gr (int (&a) [10]) {
    int gi = 0;
    for (int i=1; i<10; ++i)
      if (a[i] > a[gi]) gi = i;
    return a[gi];
  }
  int kl (int (&a) [10]) {
    int gi = 0;
    for (int i=1; i<10; ++i)
      if (a[i] < a[gi]) gi = i;
    return a[gi];
  }
}      // kein Semikolon am Ende des Namensraums

...
int a[10] = {1,8,6,9,3,5,7,2,4,0};
...
int g = Suche::gr (a);  // OK Aufruf von "Suche::gr"
int k = Suche::kl (a);  // OK Aufruf von "Suche::kl"
...
int g = gr (a);         // FEHLER: "gr" undefiniert
int k = kl (a);         // FEHLER: "kl" undefiniert
```

[27] Felder von Referenzen sind generell verboten, nicht nur als Parametertyp. Referenzen werden später behandelt.

`Suche` ist ein Namensraum mit zwei Elementen: den Funktionen `gr` und `kl`. Die Elemente des Namensraums werden mit Hilfe des Bereichsauflösungsoperators "`::`" (kürzer "Bereichsoperator", engl. *"scope operator"*) explizit qualifiziert. Die Qualifizierung ist nur ausserhalb des Namensraums nötig. Beispiel:

```
namespace NS {
  int doppel   (int x) { return 2*x; }
  int vierfach (int x) {
     int xx = doppel (x);  // innerhalb von NS: doppel wird
     return doppel (xx);   // ohne Qualifizierung verwendet
  }
}
...
int x = NS::doppel (2);    // ausserhalb von NS: doppel
...                        // muss qualifiziert werden
```

Verstreute Namensraumdefinition

Ein Namensraum muss nicht zusammenhängend definiert werden. Seine Definitionen können über das Programm verstreut werden:

```
namespace NS {  // NS beginnt
  int doppel   (int);
  int vierfach (int);
}               // NS endet

int f4 () {
  return NS::doppel (2);
}

int main () {
  cout << NS::vierfach (2) << endl;
}

namespace NS {  // NS beginnt erneut
  int doppel   (int x) { return 2*x; }
  int vierfach (int x) {
    int xx = doppel (x);
    return doppel (xx);
  }
}               // NS endet wieder
```

Üblicherweise platziert man alle Deklarationen in einem Abschnitt eines Namensraums und die entsprechenden Definitionen in einem zweiten.

Der globale Namensraum

Alle Definitionen, die sich *nicht* in einem bestimmten Namensraum befinden, werden dem *globalen Namensraum* zugeordnet. Der globale Namensraum hat einen leeren Namen. Beispiel:

```
int a = 5;
int f (int p) {
  return p + ::a; // Bezug auf das a im globalen Namensraum
}
```

In diesem Beispiel ist die Qualizierung natürlich überflüssig `a` und `::a` sind hier exakt das gleiche. Die Qualifizierung kann aber sinnvoll benutzt werden, um auf Namen zuzugreifen die andernfalls verdeckt sind:

```
int a = 5;        // globales a
int f (int p) {
  int a = 2*p;    // lokales a
  return p + a    // lokales a
           + ::a; // globales a
}
```

Die `using`–Deklaration

Wenn ein Name aus einem Namensraum `NS` oft benutzt wird, dann ist es gelegentlich lästig ihn ständig zu qualifizieren. Mit einer `using` *Deklaration* kann die Kurzform des Namens – der unqualifizierte Namen – eingeführt werden:

```
namespace NS {
  int doppel   (int);
  int vierfach (int);
}

int f (int x) {
  using NS::doppel;      // using-Deklaration: ab hier ist
  ...                    // innerhalb von f doppel gleich NS::doppel
  return doppel (2);     // OK
}
... doppel (..) ...      // FEHLER: doppel nicht bekannt
... NS::doppel (..) ...  // OK
```

Die `using`–Deklarationen werden wie alle anderen Deklarationen behandelt: für sie gelten die üblichen Sichtbarkeitsregeln und die mit ihnen eingeführten Namen dürfen nicht mit anderen Namen kollidieren.

```
int a = 5;

int f (int p) {
  int a = 2*p;
  using ::a;      // FEHLER: Doppeldeklaration von a
  a = 12;         // Welches a ???
  ...
}
```

Die `using`–Direktive

Mit einer `using`–Deklaration wird ein Name aus einem Namensraum verfügbar gemacht. mit der `using`–*Direktive* werden *alle* Namen eines Namensraums eingeführt. Beispiel:

```
namespace NS {
  int doppel   (int);
  int vierfach (int);
}
...
using namespace NS;       // Using-Direktive:
...                       // alle Namen von NS werden eingefuehrt
... doppel  (..) ...      // OK
... viefach (..) ...      // OK
```

Die `using`–*Direktive* entspricht im Wesentlichen einer Serie von `using`–Deklarationen. Ein feiner Unterschied von beiden liegt in Behandlung von Namenskollisionen. Using–*Deklarationen* werden wie "normale" Deklarationen an der Stelle der `using`–Deklaration behandelt. Using–*Direktiven* werden dagegen wie Definitionen an der Stelle der `using`–Direktive behandelt. Kollisionen werden aber unterdrückt, wenn der entsprechende Name nicht verwendet wird. Das folgende Beispiel zeigt den Unterschied:

```
namespace NS {
  int a;
  int b;
  int c;
}

int c;

void f1 () {     // f1 mit using-Deklarationen
  int b;
  using NS::a;          // OK
  using NS::b;          // FEHLER: Kollision von lokalem b und NS::b
```

```
  a++;                  // NS::a
  b++;                  // undefiniert
  c++;                  // globales c
}

void f2 () {     // f2 mit using-Direktive
  int b;
  using namespace NS;   // OK, NS::b ist durch lokales b abgedeckt
                        // NS::c und ::c kollidieren hier noch nicht
  a++;                  // OK: NS::a
  b++;                  // OK: lokales b
  c++;                  // FEHLER: NS::c oder das globale ::c ???
}
```

Die Using–Deklaration setzt man ein, wenn man genau weis, dass man einen bestimmten Namen aus dem Namensraum verwenden will. Eine Überdeckung ist dann nicht erwünscht und wird wie erwartet vom Compiler aufgedeckt.

Die Using–Direktive setzt man ein, wenn man generell alle Namen eines Namensraums verfügbar haben will, sich aber die Option einer lokalen Überdeckung offen halten will. Der Compiler zeigt nur dann einen Fehler an, wenn der verwendete Namen nicht eindeutig einer Definition zugeordnet werden kann. Mit der Using–Direktive wird die Schachtel Namensraum entfernt und alles, was dann ein Fehler wäre, wird gemeldet.

Aliasnamen für Namensräume

Namensräume können lokal umbenannt werden. Beispielsweise wenn der originale Name zu lang ist:

```
namespace ein_langer_namensraum {
  int a;
  ...
}
...
namespace eln = ein_langer_namensraum;
...
eln::a++; // entspricht: ein_langer_namensraum::a++;
```

Der Namensraum `std`

Die Komponenten der C++ Standardbibliothek, wie etwa alle Definitionen in `iostream` oder `string`, sind im Namensraum `std` definiert. Aus diesem Grund beginnen Programme typischerweise mit

```
using namespace std;
```

Sebstverständlich kann man statt dessen auch explizite Qualifizierungen und `using`–Deklarationen verwenden:

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

int main () {
  using std::string;        // string = std::string
  string gruss = "Hallo";
  std::cout << gruss << std::endl;
}
```

Namensräume sind ein recht neues C++–Konstrukt. Sie werden darum von etlichen Compilern noch nicht unterstützt. Oft lassen sich die Bestandteile des Namensraums `std` auch ohne Using–Direktive verwenden.

8.10 Übungen

Aufgabe 1

1. In einem Programm taucht in der Funktion `main` mehrfach folgende Anweisungsfolge auf, mit der – abhängig vom Wert von `n` – in `s` eine Summe berechnet und dann verwendet wird:

```
...
n = ...;
i = 0;
s = 0;
while (i < n) {
  i++;
  s = s + i;
}
... = ... s ...;
...
```

   Verbessern Sie die Struktur des Programms durch die Definition einer entsprechenden Funktion: Wie sieht die Funktionsdefinition und wie die Aufrufe aus?

2. Geben Sie ein Beispiel für ein Programm, in dem drei Variablen mit dem gleichen Namen auftauchen und erläutern Sie, wie die Variablen von (funktionierenden) Programmierern und Compilern unterschieden werden können. Erläutern Sie an Hand Ihres Beispiels die Begriffe “Sichtbarkeit” und “Gültigkeitsbereich”.

3. Betrachten Sie folgende Funktionsdefinitionen zur Berechnung des Maximums:

```
int max1 (int i, int j) {
   if (i > j) return i;
   return j;
}
int max2 (int i, int j) {
   if (i > j) return i;
   else return j;
}
int max3 (int i, int j) {
   if (i > j) cout << i;
   cout << j;
}
int max4 (int i, int j) {
   if (i > j) cout << i;
   else cout << j;
}
int max5 (int i, int j) {
   if (i > j) {
     cout << i; return i;
   } else {
     cout << j; return j;
   }
}
```

   Welche der Definitionen sind korrekt, worin besteht der Unterschied?

4. Betrachten Sie folgendes Programm:

```
int f (int i, int j) { i++; return j; }
int main () {
   int i = 0;
   int a[2] = {0, 1};
   cout << f(i, a[i]);
   cout << i;
}
```

   Was wird von diesem Programm ausgegeben?

5. Was wird von obigem Programm ausgegeben, wenn die Definition von `f` ersetzt wird durch:

   ```
   int f (int & i, int & j) { i++; return j; }
   ```

6. Welche Bedeutung hat der Typ `void` als Ergebnistyp einer Funktion? Wie werden Funktionen mit einem `void`–Ergebnis noch genannt?

7. Geben Sie in folgendem Programm zu jeder Verwendung eines Bezeichners die entsprechende Deklarationstelle an:

   ```
   #include <iostream>
   using namespace std;
   struct S {
     int t;
     int f (S &);
     int g ();
   };
   int i = 0;
   int t = 5;

   int S::f(S &x) {
     return t+x.g();
   }
   int f () {
     t++;
     return t;
   }
   int S::g() {
     i++;
     return i;
   }
   int main () {
     S   s[2];
     int i = 0;
     for (int i=0; i<2; i++) {
       s[i].t = i;
     }
     cout << s[i%2].f(s[(i+1)%2]) << endl;
   }
   ```

   Welche Ausgabe erzeugt das Programm (falls es korrekt ist, wenn nicht eliminieren Sie die Fehler!), verfolgen Sie den Programmlauf im Debugger!

8. Schreiben Sie zwei Funktionen: `void mehr ();` und `int wieOft ();`. Jeder Aufruf von `wieOft` soll als Ergebnis die Zahl der Aufrufe von `mehr` seit dem letzten Aufruf von `wieOft` liefern (oder seit dem Programmstart, wenn `wieOft` noch nicht aufgerufen wurde).

9. Schreiben Sie eine Funktion `swapSort` mit drei Parametern, die den Inhalt ihrer drei Parameter aufsteigend sortiert. Zum Beispiel:

   ```
   // a == 2, b == 0; c == 1
   swapSort (a, b, c);
   // a == 0, b == 1, c == 2
   ```

10. Betrachten Sie folgendes Programm:

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

    void swap (int &x, int &y) {
      int t;
      t = x;
      x = y;
      y = t;
    }
    int a[4] = {4,3,2,1};
    int i = 0;
    ```

```
int f () {
  i++;
  return i;
}
int main () {
  swap (i, a[f()]);
  cout << a[0]<<a[1]<<a[2]<<a[3]<< endl;
}
```

Was passiert hier? Wird hier `i` $= 0$ mit `a[0]` $= 4$ getauscht oder `i` $= 1$ mit `a[0]` $= 4$ oder ... ? Experimentieren Sie! Erklären Sie!

11. Beseitigen Sie alle fehlerhaften Anweisungen in folgendem Programm.

```
#include <iostream>

namespace Bim {
  int b = 1;
}
namespace Pam {
   int p = 2;
}

int b = 3;
int p = 4;

void f() {
  int p;
  using Bim::b;
  using namespace Pam;
  p = 10*p + ::p + Pam::p;
  b = 10*b + ::b + Bim::b;
}

int main () {
  int b = 10;
  f();
  std::cout << "b=      " << b      << std::endl
            << "::b=    " << ::b    << std::endl
            << "::p=    " << ::p    << std::endl
            << "Bim::b= " << Bim::b << std::endl
            << "Pam::p= " << Pam::p << std::endl;
}
```

Welche Ausgabe erzeugt das Programm, wenn alle fehlerhaften Anweisungen entfernt wurden?

Aufgabe 2

Rationale Zahlen können als Brüche dargestellt werden:

```
enum Vorzeichen {plus, minus};
struct Bruch {
  Vorzeichen vz;
  unsigned int        zaehler;
  unsigned int        nenner;
};
```

Schreiben Sie folgende Funktionen zum Erweitern, Kürzen und Gleichnamigmachen (beide werden auf den gleichen Nenner gebracht) von Brüchen:

- ohne Referenzparameter:

  1. `Bruch erweitere (Bruch, unsigned int);`

2. `Bruch kuerze (Bruch);`

- mit Referenzparameter:

1. `void kuerze      (Bruch &);`
2. `void erweitere   (Bruch &, unsigned int);`
3. `void gleichnamig (Bruch &, Bruch &);`

Aufgabe 3

Brüche seien wie oben definiert. Wir nehmen an, dass Brüche immer in *normierter* Form vorliegen: Zähler und Nenner sind positive Zahlen und der Bruch ist so weit wie möglich gekürzt (Zähler und Nenner haben keinen gemeinsamen Teiler ausser 1).

Schreiben Sie vier Funktionen:

1. `Bruch add (Bruch, Bruch);`
2. `Bruch sub (Bruch, Bruch);`
3. `Bruch mul (Bruch, Bruch);`
4. `Bruch div (Bruch, Bruch);`

zu Addition, Subtraktion, Multiplikation und Division von Brüchen. In jeder Funktion können Sie von einer normierten Darstellung der übergebenen Werte ausgehen. Das Ergebnis Ihrer Funktionen soll ebenfalls normiert sein.

Aufgabe 4

Betrachten Sie folgende Definition:

```
namespace BruchNS {
  enum Vorzeichen {plus, minus};

  struct Bruch {
    Vorzeichen vz;
    int   zaehler;
    int   nenner;

    void  lese();
    void  schreibe();
    void  kuerze();        // soweit wie moeglich kuerzen
    void  erweitere(int);
  };

  Bruch add  (Bruch, Bruch);
  Bruch sub  (Bruch, Bruch);
  Bruch mul (Bruch, Bruch);
  Bruch div  (Bruch, Bruch);
}
int main () {
  ..????...
  zwei Brueche einlesen,
  beide in maximal gekuerzter Form sowie
  Summe, Produkt, Differenz, Quotient ausgeben
  ..????...
}
namespace BruchNS {
  ..????...
  weitere Definitionen nach Bedarf
  ..????...
}
```

Ergänzen Sie das Programm um die fehlenden Bestandteile: Schreiben Sie die Funktion `main`, die zwei Brüche – jeweils positiv oder negativ – gemäß dieser Definition einliest und ihre Summe, ihr Produkt, ihre Differenz, ihren Quotient und beide in gekürzter Form ausgibt. Definieren Sie notwendigen Methoden und freien Funktionen.

8.11 Lösungshinweise

Aufgabe 1

1. Die Eingabe in die Anweisungsfolge ist der aktuelle Wert von `n`, das Ergebnis der Summation befindet sich am Ende in `s`. `n` wird darum zum Parameter und `s` zum Ergebnis:

```
int sum (int n) {
  int i = 0;
  int s = 0;
  while (i < n) {
    i++;
    s = s + i;
  }
  return s;
}
```

   Die Anweisungsfolge kann dann wie folgt durch den Aufruf ersetzt werden:

```
  ...
  n = xyz;
  i = 0;
  s = 0;
  while (i < n) {         ...
    i++;               => uvw = sum (xyz);
    s = s + i;           ...
  }
  uvw = s;
  ...
```

2. Im Skript finden sich viele Programme in denen mehrere Variablen den gleichen Namen tragen. Ein weiteres Beispiel ist:

```
  int f1 () {
    int x;  // Deklaration 1
    ...     // Gueligkeitsbereich von Dekl. 1
  }

  int main () {
    int x; // Deklaration 2
    ...    // x von Dekl. 2 hier sichtbar
    for (int x = ... ...) {  // Deklaration 3
       ...  // Gueligkeitsbereich von Dekl. 3
       ...  // (x von Dekl. 2 hier unsichtbar)
    }
    ...     // x von Dekl. 2 hier wieder sichtbar
  }
```

   Die Bedeutung des jeweiligen Auftretens des Bezeichners x wird nach den Sichtbarkeitsregeln festgelegt.

3. `max1` und `max2` sind äquivalent und OK. `max3` und `max4` sind nicht korrekt, da sie keine Werte zurück geben. `max5` gibt die richtigen Werte zurück; die Ausgabeanweisungen sind aber sicher überflüssig.

4. In der Version mit

   ```
   int f (int i, int j) { i++; return j; }
   ```

   ist die Erhöhung von `i` innerhalb von `f` ohne jede Wirkung. `f` gibt den Wert des zweiten Parameters – also 0 – zurück und dieser Wert wird dann auch ausgegeben, danach wird der – unveränderte – Wert 0 von `i` ausgegeben.

5. Mit

   ```
   int f (int & i, int & j) { i++; return j; }
   ```

   wird das `i` von `main` innerhalb von `f` – unter dem "f–Namen" `i` – erhöht. Die zweite Ausgabe ist darum 1. Die Erhöhung von `i` hat aber keinen Einfluss auf den Wert des zweiten Parameters, die erste Ausgabe ist darum weiterhin 0.

6. `void` als Ergebnistyp einer Funktion: Prozedur, kein Ergebniswert! Aufruf ist Anweisung.

7.
```
#include <iostream>
using namespace std;
struct S {     Dekl. 1
  int t;       Dekl. 1.1
  int f (S &); Dekl. 1.2
  int g ();    Dekl. 1.3
};
int i = 0;     Dekl. 2
int t = 5;     Dekl. 3

int S::f(S
           &x) {   Dekl. 5
  return t         -> Dekl. 1.1
         +x.       -> Dekl. 5
            g();   -> Dekl. 1.3
}
int f () {     Dekl. 6
  t++;         -> Dekl. 3
  return t;    -> Dekl. 3
}
int S::g() {
  i++;         -> Dekl. 2
  return i;    -> Dekl. 2
}
int main () {  Dekl. 8
  S
       s[2];   Dekl. 9
  int i = 0;   Dekl. 10
  for (int i=0;   Dekl. 11
           i<2;   -> Dekl. 11
           i++) { -> Dekl. 11
    s[            -> Dekl.  9
      i].         -> Dekl. 11
         t        -> Dekl.  1.1
           = i;   -> Dekl. 11
  }
  cout << s[      -> Dekl.  9
          i%2].   -> Dekl. 10
          f(      -> Dekl.  1.2
          s[(     -> Dekl.  9
          i+1)%2])-> Dekl. 10
          << endl;
}
```

   Ihre Vermutungen über die Ausgabe können Sie durch eigene Experimente verifizieren.

8. `wieOft` und `mehr` müssen auf eine globale Variable zugreifen:

```
int zaehl = 0;
void mehr () { zaehl++; }
int wieOft () { int t = zaehl; zaehl = 0; return t; }
```

9. `swapSort` muss drei Referenzparameter haben. Der Rest ist dann ein Sortieren, z.B:

```
void swap (int &a, int &b) {
  int t = a;
  a = b; b = t;
}

void swapSort (int &a, int &b, int &c) {
  if (a > b) swap (a, b);
  if (a > c) swap (a, c);
  if (b > c) swap (b, c);
}
```

10. Das Interessante an diesem Beispiel ist der Aufruf einer Funktion als Teil einer Parameterberechnung. Die Funktion greift dazu noch auf globale Variable zu, die selbst Referenzparameter ist. Eine Parameterberechnung verändert also den anderen Parameter ... Experimentieren Sie!

11. Das Programm ist korrekt. Es erzeugt die Ausgabe:

```
b=      10
::b=    3
::p=    4
Bim::b= 14
Pam::p= 2
```

Aufgabe 2

```
unsigned int ggt (unsigned int x, unsigned int y) {
  while (x != y) {
    if (x > y) x = x-y;
    if (y > x) y = y-x;
  }
  return x;
}

unsigned int kgv (unsigned int x, unsigned int y) {
  return (x*y/ggt(x,y));
}

Bruch kuerze (Bruch b) {
  if (b.zaehler == 0) return b;
  unsigned int f = ggt (b.zaehler, b.nenner);
  b.zaehler =  b.zaehler / f;
  b.nenner  =  b.nenner  / f;
  return b;
}

Bruch erweitere (Bruch b, unsigned int f) {
  Bruch res;
  res.vz = b.vz;
  res.zaehler = b.zaehler*f;
  res.nenner  = b.nenner*f;
  return res;
}

// Das Gleichnamig-machen von Bruechen kann nur als Funktion nur mit
// Hilfe von Refernzparametern formuliert werden.

//Varianten mit Referenzparameter:

void kuerze (Bruch &b) {
  if (b.zaehler == 0) return;
  unsigned int f = ggt (b.zaehler, b.nenner);
  b.zaehler =  b.zaehler / f;
  b.nenner  =  b.nenner  / f;
}

void erweitere (Bruch &b, unsigned int f) {
  b.zaehler = b.zaehler*f;
  b.nenner  = b.nenner*f;
}

void gleichnamig (Bruch &b1,
                  Bruch &b2) {
  unsigned int n = kgv (b1.nenner, b2.nenner);
  erweitere (b1, n/b1.nenner);
  erweitere (b2, n/b2.nenner);
}
```

Aufgabe 3

Diese Aufgabe sollte kein besonderes Problem bereiten, wenn man die Bruchrechnung beherrscht. Z.B. die Addition von Brüchen:

```
Bruch add (Bruch x, Bruch y) { // addiere normierte (gekuerzte) Brueche
  Bruch res;
  gleichnamig (x,y);
  res.nenner = x.nenner;
  if (x.vz == y.vz) { // x und y haben gleiche Vorzeichen
    res.vz = x.vz;
    res.zaehler = x.zaehler + y.zaehler;
  } else {
    // x und y haben unterschiedliche Vorzeichen
    if (x.zaehler > y.zaehler) {
      res.vz = x.vz;
      res.zaehler = x.zaehler - y.zaehler;
    } else {
      res.vz = y.vz;
      res.zaehler = y.zaehler - x.zaehler;
    }
  }
  kuerze (res);
  return res;
}
```

Aufgabe 4

Die Verarbeitung von Brüchen war bereits in den letzten Aufgaben Thema. Hier geht es im Wesentlichen darum die Definitionen in einen Namensraum einzufügen.

```
  namespace BruchNS {
    ... wie in der Aufgabenstellung ...
  }
  int main () {
    Bruch b1, b2, b3;
    b1.lese();   b2.lese();
    b1.kuerze(); b2.kuerze();
    b3 = add (b1, b2);
    b3.schreibe();
    ...
  }
  namespace BruchNS {
    ...
  }
```

9 Techniken und Anwendungen

9.1 Funktionen und schrittweise Verfeinerung

Primfaktorzerlegung

Am Beispiel des Problems der Primfaktorzerlegung wollen wir hier Funktionen als Mittel zur schrittweisen Verfeinerung eines Programms betrachten. Bei der Primfaktorzerlegung wird eine positive ganze Zahl als Produkt von Primzahlen dargestellt. Die Primfaktorzerlegung ist immer eindeutig. Beispiel:

$60 = 2 * 2 * 3 * 5$

Ein erster einfacher Ansatz ist:

```
bool teilt (int n, int t);  //wird n von t geteilt
bool prim (int n);          //ist n eine Primzahl
int  potenz (int n, int p); //bestimmt Potenz des Primfaktors p von n

int main (){
  int n;          // zu zerlegende Zahl
  int count = 0;  // Zahl der Primfaktoren

  // Zahl einlesen
  cin >> n;

  cout << "Primfaktoren von " << n << "\n";

  // Jeder Teiler der prim ist, ist ein Primfaktor:
  for (int i=2; i<n; ++i){
    if (teilt (n, i) && prim (i)) {
      cout << i << " hoch " << potenz (n, i) << "\n";
      ++count;
    }
  }
  if (count == 0)
    cout << n << " ist eine Primzahl\n";
  else
    cout << endl;
}
```

Hier wird einfach jede Zahl, die kleiner als die Eingabe n ist, daraufhin untersucht, ob sie ein Teiler von n und dazu noch prim ist. So erhält man alle Primfaktoren. Die Funktion `potenz` stellt noch fest, mit welcher Potenz ein gefundener Primfaktor in n auftritt.

Funktionen

Mit diesem ersten Ansatz wurde das Problem der Primfaktorzerlegung auf drei einfachere Teilprobleme reduziert deren Lösung an – noch ungeschriebene – Funktionen delegiert wurde:

- `teilt`: Feststellen ob eine Zahl ein Teiler einer anderen ist.
- `prim`: Feststellen, ob eine Zahl eine Primzahl ist.
- `potenz`: Unter der Voraussetzung, dass p ein Teiler von n ist, soll die größte Potenz von p gefunden werden die n teilt; d.h. die Funktion liefert das größte x mit: p^x teilt n.

Die Funktion `teilt`

Wir beginnen mit `teilt`, der einfachsten Funktion:

```
bool teilt (int n, int t) {
  /* Wird n von t geteilt */
  if (n % t == 0)
    return true;
```

```
    else return false;
}
```

oder einfacher:

```
bool teilt (int n, int t) {
  /* Wird n von t geteilt */
  return (n % t == 0);
}
```

t ist ein Teiler von n, wenn gilt $n \, modulo \, t = 0$.

Die Funktion `prim`

Eine Zahl ist eine Primzahl, wenn sie keinen Teiler hat. Diese Definition wird einfach in eine Schleife umgesetzt:

```
bool prim (int n) {
  // ist n prim?
  if (n == 2) return true;      // 2 ist eine Primzahl
  for (int i = 2; i<n; ++i) {  // hat n einen Teiler?
    if (teilt (n, i)) return false;
  }
  return true;
}
```

Die Funktion `potenz`

Es bleibt nur noch die Funktion `potenz`, welche die Potenz eines Primfaktors bestimmt. Auch hier gehen wir so einfach wie möglich vor. Gesucht ist das größte x mit $p^x \; teilt \; n$. Wir suchen systematisch danach:

```
int potenz (int n, int p) {
  // liefert groesstes x mit p hoch x teilt n
  // falls p ein Teiler von n ist.
  int i = 0,
      pp = 1; // pp == p^i

  while (teilt (n, pp)) {
    ++i;
    pp *= p;
  }
  return i-1;
}
```

Aus diesen Bestandteilen kann jetzt das Gesamtprogramm zusammengesetzt werden.

9.2 Methoden und Objektbasierter Entwurf

Programmentwurf – nicht nur Geschmacksache

Nicht nur Anfängern, auch fortgeschrittenen Software–Entwicklerinnen stellt sich regelmäßig die Frage: Benutze ich Funktionen oder Methoden, wenn ja welche, geht es eventuell auch ohne beides? Eine generelle Antwort gibt es nicht, sie muss für jede Problemstellung neu gefunden werden. Eindeutige Kriterien gibt es dabei nicht. Gelegentlich ist der Einsatz einer Methode genauso plausibel, wie der einer freien Funktion, oder der Verzicht auf beides. Es bleibt stets dem Geschmack des Software–Entwicklers überlassen, mit welchen Mechanismen der Programmiersprache er arbeiten will.

Damit ist die Sache aber nicht völlig beliebig. Zunächst einmal gibt es guten und schlechten Geschmack. In der auf Kooperation angewiesen Softwarebranche kann ein Programmierer mit einem schlechten Geschmack beim Programmentwurf seinen Ruf vollständig ruinieren. Wann aber verrät man einen guten Geschmack bei der Gestaltung eines Programms? Ein Programm zeigt den Programmier–Geschmack seines Autors mit seiner Struktur, also mit seiner Aufteilung in Typdefinitionen, Methoden und Funktionen. Diese Aufteilung muss dem Problem angemessen sein:

> Ein Stück Software hat eine gute (schöne) Struktur, wenn seine Aufteilung in Typen, Methoden und Funktionen der Struktur des Problems entspricht, das von dieser Software bearbeitet wird.

Schrittweise Verfeinerung: Funktionale Programmstruktur

Die schrittweise Verfeinerung ist eine Entwicklungsmethode für Programme. Man geht dabei davon aus, dass die zu lösende Problemstellung in eine Folge von Unteraufgaben aufgegliedert werden kann. Jede der Unteraufgaben wird dann in Unter–Unteraufgaben zergliedert. Mit diesem Prozess der Zergliederung – oder "Verfeinerung" – fährt man so lange fort bis die gesamte Aufgabenstellung in einfache und elementare Basisaufgaben zerlegt ist. Funktionen erhöhen die Übersichtlichkeit einer Verfeinerung, wenn sie zur Lösung der Unter– und Unter–Unteraufgaben eingesetzt werden.

```
// Verfeinerungsstufe 1 ------------------------------------------
T1    f_1 (...);
T2    f_2 (...);
int   main () { ...f_1 (...)... ...f_2 (...)... }

// Verfeinerungsstufe 2 ------------------------------------------
T11   f_11 (...);
T12   f_12 (...);
T1    f_1 (...) { ...f_11 (...)... ...f_12 (...)... }
T2    f_2 (...) { ... }
int   main () { ...f_1 (...)... ...f_2 (...)... }

// Verfeinerungsstufe 3 ------------------------------------------
T111  f_111 (...);
T112  f_112 (...);
T11   f_11 (...) { ...f_111 (...)... ...f_112 (...)... }
T12   f_12 (...) {  ...  }
T1    f_1 (...)  { ...f_11 (...)... ...f_12 (...)... }
T2    f_2 (...)  { ... }
int   main ()    { ...f_1 (...)... ...f_2 (...)... }

// Verfeinerungsstufe 4 ------------------------------------------
...
```

Ein Programmentwurf entsprechend der schrittweisen Verfeinerung führt zu einer funktionalen Programmstruktur: Das Gesamtprogramm besteht aus einer Serie von Funktionen. Die Main–Funktion repräsentiert den ersten Verfeinerungsschritt. Die von `main` direkt aufgerufenen Funktionen stellen den zweiten Schritt der Verfeinerung dar. Sie rufen selbst wieder Unterfunktionen auf, die die nächste Stufe der Verfeinerung darstellen, und so weiter.

Beispiel, Problemstellung: Lineares Gleichungssystem und seine Lösung

Mit Verbunden und Methoden kann unser Arsenal an Programmentwicklungsmethoden über die schrittweise Verfeinerung hinaus erweitert werden. Bevor wir dazu kommen, wollen wir an einem einfachen Beispiel die schrittweise Verfeinerung zeigen. Anschließend wird dann das gleiche Beispiel von einem etwas anderen Standpunkt aus betrachtet.

Als Beispiel für die zwei Methoden der Programmentwicklung nehmen wir uns das Lösen eines einfachen linearen Gleichungssystems vor. Die Problemstellung und der mathematisch formulierte Lösungsweg ist schnell beschrieben.

Ein Gleichnungssystem mit drei Unbekannten x_1, x_2, x_3 sei geben durch eine Matrix $\mathbf{A}$ und einen Vektor $\vec{b}$:

$$\mathbf{A}\vec{x} = \vec{b} \qquad wobei: \quad \mathbf{A} = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix} \quad \vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}, \quad \vec{b} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

Falls die Determinate von $\mathbf{A}$ ungleich Null ist, ist die Lösung des Systems:

$$x_1 = \frac{|\mathbf{A^{[1]}}|}{|\mathbf{A}|} \quad x_2 = \frac{|\mathbf{A^{[2]}}|}{|\mathbf{A}|} \quad x_3 = \frac{|\mathbf{A^{[3]}}|}{|\mathbf{A}|}$$

$|\mathbf{A}|$ ist dabei die Determinante von $\mathbf{A}$ und $\mathbf{A}^{[\mathbf{i}]}$ ist die Matrix $\mathbf{A}$, bei der die i–te Spalte durch $\vec{b}$ ersetzt wurde, z.B.:

$$\mathbf{A}^{[\mathbf{2}]} = \begin{pmatrix} a_{1,1} & b_1 & a_{1,3} \\ a_{2,1} & b_2 & a_{2,3} \\ a_{3,1} & b_3 & a_{3,3} \end{pmatrix}$$

Programmentwurf nach schrittweiser Verfeinerung

Der mathematisch formulierte Lösungsweg soll jetzt in ein Programm umgesetzt werden. Nach der Methode der schrittweisen Verfeinerung wird man dazu zunächst die groben Schritte identifizieren:

```
...
typedef float Matrix[3][3];
typedef float Vektor[3];


int main () {
  Matrix A;
  Vektor b, x;

  ??.. A und b einlesen ..??

  float d_A = ??.. Determinante von A ..??

  if ( d_A == 0) {
    cout << "Gleichungssystem nicht loesbar\n";
    return 1;
  }

  for (int i=0; i<3; ++i) {
     Matrix A_mod = ??... A mit Spalte i durch b ersetzt ??
     float d = ??.. Determinate von A_mod ..??
     x[i] = d/d_A;
  }

  ??.. x ausgeben ..??
}
```

Den informal spezifizierten Arbeitsschritten werden jetzt Funktionen zugeordnet. Etwa wie folgt:

```
#include <iostream>

typedef float Matrix[3][3];
typedef float Vektor[3];

void  lies     (Matrix &);
void  lies     (Vektor &);
void  schreib  (const Vektor &);
float det      (const Matrix &);
void  hoch     (const Matrix &, const Vektor &, int, Matrix &);

int main () {
  Matrix A;
  Vektor b, x;

  lies(A);
  lies(b);

  float d_A = det(A);

  if ( d_A == 0) {
    cout << "Gleichungssystem nicht loesbar\n";
```

```
    return 1;
  }

  for (int i=0; i<3; ++i) {
    Matrix A_mod;
    hoch(A, b, i, A_mod);

    float d = det(A_mod);
    x[i] = d/d_A;
  }

  schreib(x);
  cout << endl;
}
```

Im nächsten Verfeinerungsschritt sind die Definitionen der Funktionen zu formulieren. Dabei ergeben sich eventuell weitere Unterfunktionen. Wir überlassen dies dem Leser.

Objektbasierter Entwurf: Datentypen *Können* und *Verantworten* etwas

Die Beschränkung der schrittweisen Verfeinerung liegt in ihrer rein "handlungsorientierten" Sicht der Probleme und ihrer Lösung. Die Aufmerksamkeit richtet sich auf Aktionen und Unteraktionen. Eine moderne Entwurfsmethodik ergänzt die Sicht auf Handlungen um eine Betrachtung von Daten und deren Typen. Man fragt dazu nicht mehr (nur) "Was macht das Programm, welche Werte werden in ihm in welchen Schritten berechnet?" sondern fragt nach den *"Dingen"* oder *"Objekten"*, die im Problem und damit auch in seiner Lösung, dem Programm, vorkommen. In Anlehnung an den Begriff der schrittweisen Verfeinerung wird die Entwicklungsmethodik, die sich aus einer an den Dingen und ihren Typen orientierten Sicht ergibt, oft als *Datentyp–Verfeinerung* bezeichnet. Die Datentyp–Verfeinerung ist eine *Objekt–basierte* Sicht der Dinge und die erste Stufe des *objektorientierten Entwurfs*, mit dem wir uns später beschäftigen werden.

> Die zentrale Frage des *objektbasierten Entwurfs* ist:
>
> Welche Dinge/Objekte kommen im Problembereich vor? Was sollen diese Dinge/Objekte *wissen* und und was sollen sie *können*? Welche Verantwortung übernehmen sie damit?

Die Antwort auf die Frage nach den Dingen/Objekten ist bei unserem kleinen Beispiel sehr einfach: Wir haben es mit einer Matrix und zwei Vektoren zu tun. Was sind und was können Vektoren und Matrizen in diesem Beispiel? Es *sind* Kollektionen von Werten. Und was sollen sie *können*? Diese Frage ist neu. Mathematisch gesehen kann eine Matrix und ein Vektor nichts. In einer modernen Entwicklungsmethodik für Software tendiert man aber – anders als in der Mathematik und nach vielen Jahren der Erfahrung – dazu, Objekten eine *Verantwortung* zuzuschreiben. Verantwortung kann aber nur übernommen werden, wenn *Wissen* und *Können* vorhanden ist.

Vektoren und Matrizen sind strukturiert. Sie wissen damit wie sie strukturiert sind. Mit diesem Wissen können sie die Verantwortung dafür übernehmen, dass ihre Daten eingelesen und ausgegeben werden. Eine Matrix kann dazu noch die Verantwortung für die Berechnung ihrer Determinate übernehmen. Wir statten Vektoren und Matrizen also mit *Methoden* aus, die es ihnen erlauben ihrer Verantwortung für Lesen, Schreiben und Determinantenberechnung nachzukommen. Der erste Entwurf des Programms nach der objektbasierten Entwurfsmethode ist damit:

```
struct Vektor {
  float v[3];       // Ein Vektor ist eine Kollektion von 3 Floats
  void lies ();     // Ein Vektor kann sich einlesen
  void schreib ();  // Ein Vektor kann sich ausgeben
};

struct Matrix {
  float m[3][3];    // Eine Matrix ist eine Kollektion von 3*3 Floats
  void lies ();     // Eine Matrix kann sich einlesen
  void schreib();   // Eine Matrix kann sich ausgeben
  float det ();     // Eine Matrix kann ihre Determinante berechnen
};

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

Man beachte, dass in diesem Programmentwurf nur die Definitionen von Datentypen vorkommen. Er enthält noch keinerlei Aktionen!

Lesen, Schreiben und das Berechnen der Determinate sind damit erledigt – zumindest was den Entwurf betrifft. Was tun wir mit den Matrizen, die durch Austausch einer Spalte aus **A** entstehen? Wird die Verantwortung dafür in Form einer Methode der Matrix zugewiesen, oder trägt das Programm in Form einer freien Funktion global die entsprechende Verantwortung. Wir entscheiden uns für eine Methode und erweitern den Programmentwurf entsprechend um eine Methode `hoch`.

Jetzt sind alle Teilaufgaben in Form von Verantwortlichkeiten verteilt und der Programmentwurf kann um die notwendigen Aktionen erweitert werden:

```
struct Vektor {
  float v[3];
  void lies ();
  void schreib ();
};

struct Matrix {
  float m[3][3];
  float det ();
  void lies ();
  void schreib ();
  Matrix hoch (Vektor, int);  // Eine Matrix kann Kopie von sich mit einer
};                            // durch einen Vektor ersetzten Spalte erzeugen

int main () {
  Matrix a;
  Vektor b, x;

  a.lies();
  b.lies();

  if (a.det() == 0) {
    cout << "Das System hat keine eindeutige Loesung\n";
    return 1;
  }

  for (int i=0; i<3; ++i)
    x.v[i] = (a.hoch(b,i)).det()/a.det();

  cout << "Loesung:\n";
  x.schreib();
  cout << endl;
}
```

Diesem Programmentwurf fehlt zum vollständigen Programm nur noch die Definition der Methoden. Wir geben dies als Aufgabe in die Verantwortung der Leserin bzw. des Lesers und hoffen, dass in ihr oder ihm eine Methode bereit steht, um diese Aufgabe zu lösen.

9.3 Rekursion

Direkte und indirekte Rekursion

Rekursive Funktionen sind Funktionen, die sich selbst aufrufen, entweder direkt, oder indirekt über andere Funktionen. Ein Beispiel für *direkte Rekursion* ist:

```
#include <iostream>

using namespace std;

int fak (int n) {              // rekursive Funktion
  if (n==0) return 1;
  else      return fak(n-1)*n; // rekursiver Aufruf
}
```

```
int main () {
  int x;
  cin >> x;
  cout << x << "! = " << fak (x) << endl;
}
```

Die Funktion `fak` berechnet die Fakultätsfunktion. Man sieht wie die rekursive Definition der Fakultät

$$\begin{aligned} fak(0) &= 1 \\ fak(n) &= fak(n-1) * n \end{aligned}$$

sich unmittelbar in der Funktionsdefinition oben widerspiegelt. Das schließt auch den Definitionsbereich mit ein. Die mathematische Funktion fak ist nur für positive Werte und Null definiert. Genauso die Funktion `fak`, die auch nur für Null und positive Werte ein Ergebnis liefert.

Die Fakultätsfunktion ist ein Beispiel für *direkte Rekursion*: die Funktion ruft sich selbst auf. Im folgenden Beispiel für eine *indirekte Rekursion* ruft eine Funktion sich selbst nicht direkt, sondern indirekt über eine andere auf:

```
#include <iostream>

using namespace std;
                        // Gegenseitiger Aufruf, indirekte Rekursion
bool ungerade (int);    // ruft gerade auf
bool gerade (int);      // ruft ungerade auf

int main () {
  int x;
  cin >> x;
  cout << x;
  if (gerade(x))
    cout << " ist gerade" << endl;
  else
    cout << " ist ungerade" << endl;
}
bool gerade (int x) { //indirekt rekursive Funktion
  if (x==0) return true;
  else      return ungerade (x-1); //indirekt rekursiver Aufruf
}
bool ungerade (int x) {
  if (x==0) return false;
  else      return gerade (x-1);
}
```

Auch hier wurde eine mathematische Definition direkt umgesetzt:

$$\begin{aligned} gerade(0) &= true \\ gerade(n) &= ungerade(n-1) \\ ungerade(0) &= false \\ ungerade(n) &= gerade(n-1) \end{aligned}$$

Instanzen rekursiver Funktionen

Beim Aufruf einer Funktion wird eine Funktionsinstanz erzeugt und aktiviert. Bei rekursiven Funktionen ist das genauso. Programme mit rekursiven Funktionen – egal ob direkt oder indirekt rekursiv – haben damit eine interessante Eigenschaft: Zu einem Zeitpunkt während des Programmlaufs können gleichzeitig beliebig viele Instanzen der gleichen Funktion existieren.

Im ersten Beispiel der Funktion `fak`

- erzeugt ein Aufruf `fak(4)` eine erste Instanz von `fak`.
- Von dieser wird `fak(3)` aufgerufen. Damit wird eine zweite Instanz von `fak` erzeugt (ohne die erste zu beenden!).

- In `fak(3)` (genauer: in der Instanz von `fak`, die zur Berechnung von `fak(3)` erzeugt wurde) wird `fak(2)` aufgerufen und damit die dritte Instanz erzeugt.
- In `fak(2)` wird `fak(1)` aufgerufen und damit die vierte Instanz erzeugt.
- `fak(0)` – die fünfte und letzte Instanz – liefert ein Ergebnis. Alle Instanzen werden jetzt in der umgekehrten Reihenfolge ihrer Erzeugung wieder vernichtet.

Man beachte, dass jede Instanz ihre eigenen formalen Parameter und – falls vorhanden – lokalen Variablen hat. Der Aufruf `fak(4)` erzeugt beispielsweise fünf Exemplare des formalen Parameters `n`, die alle zur gleichen Zeit im Speicher vorhanden und mit unterschiedlichen Werten belegt sind. (Siehe Abbildung 26)

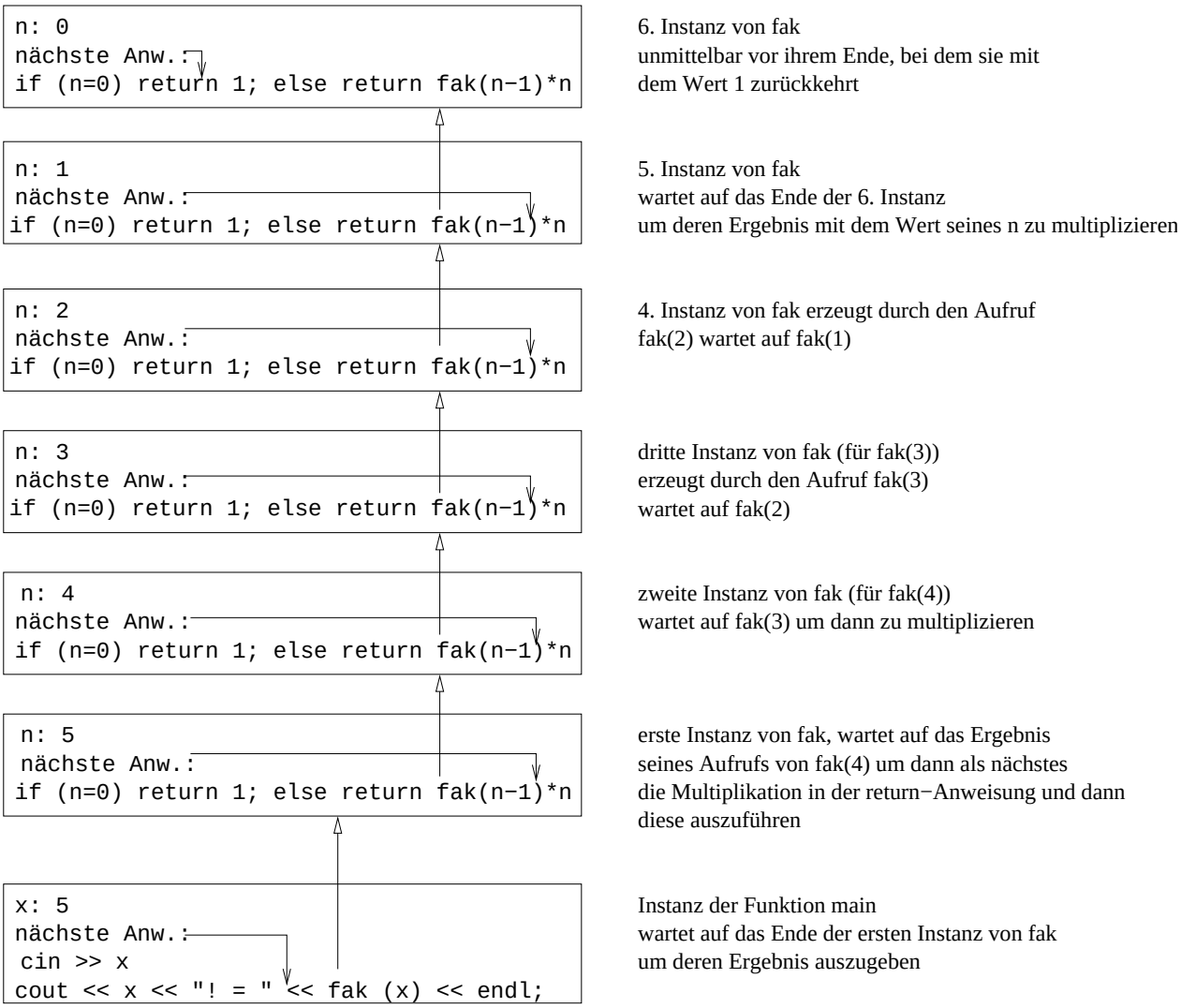


Abbildung 26: Instanzen einer rekursiven Funktion

Endlos–Rekursion

Gelegentlich reicht der Speicher des Rechners nicht aus, um all die Instanzen einer rekursiven Funktion anzulegen, deren Erzeugung das Programm verlangt. – Speziell dann wenn es die Erzeugung unendlich vieler Instanzen fordert.

Ruft man beispielsweise die Funktion `gerade` aus dem zweiten Beispiel von oben mit einem negativen Argument auf, dann verlangt das Programm die Erzeugung unendlich vieler Instanzen von `gerade` und `ungerade`. `gerade (-1)` beispielsweise führt zur Erzeugung von `ungerade (-2)`, `gerade (-3)`, etc. Die Rekursion endet nicht (freiwillig), es ist eine *Endlos–Rekursion*. Ihre Abarbeitung wird allerdings dadurch begrenzt, dass der Rechner nicht unendlich viele Funktionsinstanzen erzeugen kann. Jede von ihnen benötigt ja einen eigenen Speicherbereich für lokale Variablen und Parameter. Nach einer bestimmten Zahl von Schritten endet die

Sache damit dass kein freier Speicher mehr vorhanden ist und das Programm mit Fehlermeldung abgebrochen wird (`Stack Overflow`, `Segmentation fault` o. ä.).

9.4 Rekursion als Kontrollstruktur, Rekursion und Iteration

Rekursion zur Bearbeitung rekursiver Definitionen

Der einfachste Einsatz rekursiver Funktionen ist die Umsetzung von rekursiven Definitionen in Funktionen. Am Beispiel der Berechnung der Fakultätsfunktion haben wir die Äquivalenz bereits gezeigt.

Ein weiteres Beispiel ist die Berechnung des Binomialkoeffizienten:

$$\begin{pmatrix} n \\ k \end{pmatrix} = \begin{cases} 0 & n < k \\ 1 & k = 0 \; oder \; n = k \\ \begin{pmatrix} n-1 \\ k-1 \end{pmatrix} + \begin{pmatrix} n-1 \\ k \end{pmatrix} & 0 < k < n \end{cases}$$

Diese mathematische Funktion wird einfach in folgende C++–Funktion umgesetzt:

```
int binomRekursiv (int n, int k) {
  if (n==k || k == 0) return 1;
  if (n < k)          return 0;
  return binomRekursiv (n-1, k-1) + binomRekursiv (n-1, k);
}
```

Rekursion und Iteration

Man vergleiche diese einfache und elegante Lösung mit der iterativen (= mit Schleife) Variante:

```
int binomIterativ (int n, int k) {
  int a[20], //fuer n<20, k<20
      j = 1;
  for (int l=0; l<20; ++l) a[l] = 0;
  a[0] = 1;
  while (j<=n) {
      for (int x=j; x>0; --x) {
         //INV : a[x+1..j] ist neu a[1..x] ist alt
         a[x] = a[x-1] + a[x];
      }
      ++j;
  return a[k];
}
```

An der größeren Komplexität der iterativen im Vergleich zur rekursiven Lösung erkennt man hier, dass die Rekursion mächtiger ist als die Iteration (= Schleife). Die iterative Version der Binomialfunktion benötigt ein Feld. Die Feldgröße ist hier auf 20 gesetzt. Zur Berechnung beliebiger Eingaben müsste man ein unbeschränkt großes Feld zur Verfügung haben.

Damit haben wir schon die Essenz der Rekursion:
Rekursion = Schleifen + unbeschränkt große Felder!

Die Rekursion benötigt natürlich ebenfalls Platz. Statt der Felder mit vielen Feldelementen erzeugt sie viele Funktionsinstanzen. Der Unterschied zwischen beiden liegt daran, dass bei der Rekursion der Platz während der Berechnung *nach Bedarf* beschafft wird (für jede erzeugt Instanz etwas Speicher). Bei der iterativen Lösung muss dagegen *im voraus* Platz in Form eines Feldes geschaffen werden.

Rekursion ist eine Kontrollstruktur

Funktionen sind also nicht nur ein Mittel zur Verbesserung der Programmstruktur, rekursiv angewendet sind sie auch eine mächtige Kontrollstruktur.

Kontrollstruktur nennt man die Elemente einer Programmiersprache wie `if` oder `while`, die selbst keine Anweisung oder Ausdruck sind, sondern als "Steuerelemente" zur Steuerung des Programmablaufs ("Kontrollfluss") eingesetzt werden.

Was die Schleife für Anweisungen ist, das ist die Rekursion für Funktionen: Ein Mechanismus zur unbestimmten – situationsabhängigen – Wiederholung des gleichen.

Schleifen sind Rekursionen

In Schleifen wird eine Folge von Anweisungen immer wieder ausgeführt – so lange, bis das Ziel der Schleife erreicht ist. Mit einer rekursiven Funktion verhält es sich genauso, sie wird (von sich selbst) so lange aufgerufen, bis sie ihr Ziel erreicht hat.

Die enge Verwandtschaft von Schleifen und rekursiven Funktionen erkennt man daran, dass jede `while`–Schleife (und damit auch jede andere Schleife) in eine rekursive Funktion transformiert werden kann:

```
while (B) A;
```

ist äquivalent zu:

```
void doWhile () { if (B) { A; doWhile (); } }
```

Beispiel: Konversion zwischen Schleife und Rekursion

Als Beispiel einer Umwandlung betrachten wir noch einmal die Fakultätsfunktion f. Der Ausgangspunkt ist eine rekursive mathematische Definition:

$$\begin{aligned} f(0) &= 1 \\ f(x) &= f(x-1) * x \end{aligned}$$

Diese Definition kann mechanisch und ohne bemerkenswerten Einsatz von Gehinmasse in eine C++–Funktion umgesetzt werden (siehe oben). Eine iterative Lösung ist nicht ganz so leicht zu finden. Mit etwas Nachdenken kommt man aber sicher zu einer Lösung wie:

```
int f (int x) {
  int a = 1;
  int i = 0;
  while (i < x) {
    ++i;
    a = a*i;
  }
  return a;
}
```

Im Gegensatz zur rekursiven Version sieht man dieser Variante nicht unmittelbar an, dass sie die Fakultätsfunktion berechnet. Bei der Umsetzung in eine Schleife hat also eine gewisse Transformation stattgefunden. Diese machen wir deutlich, indem wir die Schleife mit Hilfe unserer allgemeinen Äquivalenzformel

```
while (B) A; ≡ void doWhile () { if (B) { A; doWhile (); } }
```

wieder in eine rekursive Funktion `doWhile` zurück transformieren.

Die Schleife

```
while (i < x) {
  ++i;
  a = a*i;
}
```

hängt von den Variablen `x`, `i` und `a` ab. In eine Funktion `doWhile` umgewandelt, muss diese die drei als Parameter übernehmen:

```
void doWhile (int x, int i, int & a) { ... }
```

`a` muss den Ergebniswert heraustransportieren und ist darum Referenzparameter. Den Funktionskörper finden wir durch mechanisches Umsetzen entsprechend der Äquivalenzformel:

```
void doWhile (int x, int i, int &a) {
  if (i < x) {
    ++i;
    a = a*i;
    doWhile (x, i, a);
  }
}
```

Diese "While–Funktion" ersetzt innerhalb der ursprünglichen Funktion `f` die While–Schleife:

```
void doWhile (int x, int i, int &a) {
  if (i < x) {
    ++i;
    a = a*i;
    doWhile (x, i, a);
  }
}

int f (int x) {
  int a = 1;
  int i = 0;
  doWhile (x, i, a);
  return a;
}
```

Wie wir an diesem Beispiel gezeigt haben, lässt sich jede Schleife als rekursive Funktion ausdrücken. Umgekehrt kann aber eine beliebige rekursive Funktion *nicht* ohne weiteres in eine Schleife umgesetzt werden. Schleifen sind Rekursionen, bei denen der rekursive Aufruf die letzte Aktion der Funktion ist, bei denen es also keine "nachhängenden Operationen" gibt. Die Fakultätsfunktion ist nicht von dieser Art. Nach dem Aufruf muss das Ergebnis des rekursiven Aufrufs noch mit dem Parameter multipliziert werden.

Bei der Realisation einer rekursiv definierten Funktion als Schleife transformiert die Programmiererin die Definition völlig unbewusst in eine Form ohne "nachhängende Operationen". Dies zeigt die natürliche Begabung der (meisten) Menschen fürs Programmieren. Allerdings ist eine solche Transformation gelegentlich *nicht möglich*, in diesem Fall gibt es keine Schleifenlösung für einen rekursiven Algorithmus.

Rekursion ist also die mächtigste Kontrollstruktur. Mit ihr lassen sich alle Probleme lösen, die sich mit Schleifen lösen lassen und noch einige mehr.

9.5 Rekursion als Programmiertechnik, rekursive Daten

Rekursion ist ein Mittel zur Problemreduktion

Rekursion kann als Programmiertechnik eingesetzt werden: sie erlaubt oft eine einfachere Lösung als eine Schleife. Gelegentlich gibt sie überhaupt erst die erste Idee zur Lösung eines Problems.

Generell eignen sich rekursive Funktionen zur Lösung von Problemen die folgende Struktur haben: Das Problem ist

- entweder einfach, oder
- seine Lösung lässt sich auf die Lösung *einfacherer gleichartiger* Teilprobleme reduzieren.

Sowohl die Fakultätsfunktion, als auch die Binomialkoeffizienten sind von dieser Art. Betrachten wir beispielsweise die Fakultätsberechnung:

- $0!$ zu berechnen ist einfach: $0! = 1$;
- $n!$ lässt sich auf die Berechnung von $(n-1)!$ reduzieren: $n! = (n-1)! * n$.

Das zu lösende Teilproblem muss *gleichartig* sein – sonst haben wir keine Rekursion; es mus eine *Basis* in Form eines einfachsten Falls geben – sonst hat die Rekursion kein Ende; das Teilproblem muss *einfacher* sein – sonst gibt es keine Bewegung in Richtung Basis und kein Ende der Rekursion.

Wann ist ein Problem *einfacher*? Das ist intuitiv klar: $(n-1)$ ist einfacher als n. Es ist einfacher, weil es dichter an der Basis 0 liegt. Rekursive Algorithmen auf natürlichen Zahlen haben darum als Basis stets 0, oder andere kleine Zahlen und das Teilproblem ist die Berechnung für kleinere Werte.

Rekursive Funktionen können auf Daten operieren, die eine Basis haben und sich in gleichartige einfachere Teildaten aufgliedern lassen. Solche Daten nennen wir *rekursiv*. Die natürlichen Zahlen sind rekursive Daten. Die Basis ist 0, die Zergliederung ist die Subtraktion von 1, bei der eine einfachere $n-1$ entsteht.

Rekursion kann auch eingesetzt werden, wenn wir es nicht mit einem mathematischen Problem zu tun haben. Die Daten sind dann nicht mehr unbedingt natürliche Zahlen und es ist nicht mehr klar, ob es sich um rekursive Daten handelt. Wenn doch, dann haben wir eine gute Chance den gesuchten Algorithmus als rekursive Funktion definieren zu können. Es lohnt sich darum die Daten zu betrachten und zu entscheiden, ob sie rekursiv sind.

Rekursive Daten

Rekursive Funktionen sind bestens geeignet Berechnungen auf rekursiven Daten auszuführen. *Rekursive Daten* sind Mengen (von "Dingen"), die sich systematisch aus einer Grundmenge – der Basis – konstruieren lassen:

Ein Ding ist

- entweder einfach, also ein Element aus einer Grundmenge einfacher Dinge,
- oder es kann mit Hilfe eines Konstruktionsmechanismus' aus anderen gleichartigen Dingen hergestellt werden.

Das erste Beispiel für rekursive Daten sind die natürlichen Zahlen. Eine natürliche Zahl ist:

- entweder 0,
- oder durch Addition von 1 aus einer anderen natürlichen Zahl hergestellt.

Ein anderes Beispiel sind einfache, vollständig geklammerte, arithmetische Ausdrücke. Ausdrücke sind

- entweder Ziffern (z.B.: `1, 4, 0`),
- oder geklammerte Ausdrücke: Durch Operatoren verknüpfte Ausdrücke in Klammern (z.B.: `(2+3)`, `((2-1)*3)`, `((1+2)*(5-1))` etc.).

Ein weiteres Beispiel sind Zeichenketten: Eine Zeichenkette ist

- entweder leer,
- oder eine zusammengesetzte Zeichenkette, die durch Anfügen eines Zeichens an eine einfachere (um ein Zeichen kürzere) Zeichenkette entstanden ist.

Die Struktur rekursiver Daten ist nicht von Mutter Natur oder sonst irgendwem vorgegeben. Sie wird von Software–Entwicklern bei und nach Bedarf definiert. Manchmal sind mehrere Definitionen möglich.

Rekursive Daten und rekursive Algorithmen

Rekursive Daten haben eine Struktur. Ihr innerer Aufbau entspricht ihrer Konstruktion aus einfacheren Komponenten gleicher Art.

Wenn die Daten aus Komponenten gleicher Art zusammengesetzt sind, dann kann der Algorithmus, der auf dem "Gesamt–Ding" operiert, auch benutzt werden, um dessen Komponenten zu verarbeiten.

Die Fakultätsfunktion operiert beispielsweise auf den natürlichen Zahlen als rekursiven Daten. Die Definition und Aktion der Funktion

$$fak(n) = \begin{cases} 1 & n = 0 \\ fak(n-1) * n & n > 0 \end{cases}$$

folgt dem Aufbau der natürlichen Zahlen aus 0 (Basiselement) und der Addition von 1 (Konstruktion eines neuen komplexeren Wertes aus einem gleichartigen Einfacheren).

Der rekursive Algorithmus zerlegt und verarbeitet Zahlen entsprechend ihrem Aufbau:

1. Er entscheidet, ob das zu bearbeitende "Ding" einfach oder komplex ist.
2. Im einfachen Fall wird das Ergebnis direkt aus dem Basiselement 0 erzeugt.
3. Im komplexen Fall
   a) wird das "Ding" zerlegt, d.h. die einfacheren Teile, aus denen es konstruiert wurde, werden freigelegt ($n = (n-1)+1$);
   b) der Algorithmus wird auf die Teile angewendet und liefert Teilergebnisse ($fak(n-1)$);
   c) aus den Teilergebnissen wird das Gesamtergebnis hergestellt ($fak(n-1) * n$).

Interpretation einer Zeichenkette als Zahl

Zeichenketten können wie oben dargelegt als rekursive Daten aufgefasst werden, bei denen längere (kompliziertere) Zeichenketten aus einfacheren (kürzeren) durch Anhängen eines Zeichens entstehen. Zeichenkette können aber auch in andrer Art als rekursive Daten aufgefasst werden: Die längeren können aus den einfacheren durch Voranstellen (statt Anhängen) eines Zeichens entstehen. Nach dieser Aufassung ist eine Zeichenkette

- entweder ein einzelnes Zeichen,
- oder aus einem Zeichen und einer Zeichenkette durch Voranstellen des Zeichens erzeugt worden.

Die Interpretation einer Zeichenkette s als Zahl durch den menschlichen Geist folgt (nach Selbstbeobachtung) diesem Aufbau. Handelt es sich bei der Zeichenkette um ein Folge von Ziffern, deren Gesamtwert zu bestimmen ist, dann geht das Hirn des Menschen (ebenfalls nach Selbstbeobachtung) nach folgendem rekursiven Algorithmus vor:

$$zahlWert(s) = \begin{cases} zifferWert(c) & s \;=\; Zeichen\ c \\ zifferWert(c_1) \cdot 10^{\mathit{Länge}(s)-1} + zahlWert(s_1) & s \;=\; c_1\, s_1 \end{cases}$$

Das bedeutet nichts anderes, als dass man bei der Bestimmung des Werts einer Ziffernfolge s entscheidet, ob es sich um eine einzelne Ziffer handelt. Wenn ja ist der Wert sofort klar, es ist der Wert der Ziffer. Wenn nein, dann bestimmt man den Wert der hinteren Ziffernfolge und nimmt die erste mit der richtigen Wertigkeit dazu. Beispiel:

$$\begin{aligned} zahlWert(\texttt{321}) &= zifferWert(\texttt{3}) \cdot 10^2 + zahlWert(\texttt{21}) \\ &= 300 + zifferWert(\texttt{2}) \cdot 10^1 + zahlWert(\texttt{1}) \\ &= 300 + 20 + zifferWert(\texttt{1}) \cdot 10^0 \\ &= 300 + 20 + 1 \\ &= 321 \end{aligned}$$

Diese Definition lässt sich unmittelbar in eine Funktion umsetzen:

```
int zifferWert (char z) {
  return (z - '0');
}

int zahlWert (string s) {
  if (s.length() == 1)
    return zifferWert (s.at(0));
  else
    return zifferWert (s.at(0)) * int(pow (10, s.length()-1))
           + zahlWert (s.substr (1, s.length()-1));
}
```

Eine alternative Zerlegung der Zeichenkette

Einen gegebenen rekursiven Algorithmus – wie etwa die Fakultätsfunktion – auf gegebenen rekursiven Daten – z.B. den natürlichen Zahlen – in C++ umzusetzen ist kein Problem. Etwas anspruchsvoller wird die Angelegenheit wenn weder die Struktur der Daten, noch der Algorithmus unmittelbar bekannt ist. In dem Fall hilft nur Nachdenken und Ausprobieren. Man muss sich dabei darüber im Klaren sein, dass man dabei auch eventuell einmal in einer Sackgasse landen kann.

Bei der Analyse von Zeichenketten als Zahlen ist die Interpretation der Zeichenketten "von vorn" ja beispielsweise keineswegs offensichtlich. Man kann sie ebenso gut "von hinten" rekursiv zerlegen und als elementares Datum die leere Zeichenkette festlegen können: Eine Zeichenkette nach dieser Interpretation ist

- entweder leer,
- oder eine Zeichenkette der ein Zeichen folgt.

Auch zu dieser Definition von Zeichenketten als rekursiven Daten kann ein Algorithmus zur Wertberechnung angegeben werden:

$$zahlWert(s) = \begin{cases} 0 & s\ ist\ leer \\ zahlWert(s_1) \cdot 10 + zifferWert(c_1) & s\ =\ s_1\, c_1 \end{cases}$$

Diese zweite Definition der Zeichenkette als rekursive Daten sowie des passenden Algorithmus ist sicher genauso richtig, wie die weiter oben. Sie lässt sich auch recht einfach in eine C++–Funktion umsetzen. Wir müssen allerdings etwas heftiger auf der Zeichenkette operieren, um zum Ergebnis zu kommen:

```
int zifferwert (char c) {
  return (c - '0');
}

int zahlwert (string s) {
  if (s.length() == 0) return 0;
  else
    return
      zahlwert (s.substr(0, s.length()-1)) * 10
      +
      zifferwert (s.at(s.length()-1));
}
```

An diesem Beispiel sieht man, dass eine unterschiedliche Interpretation der gleichen Daten zu unterschiedlichen Algorithmen für das gleiche Problem führt. Ein Risiko besteht darin, dass man unter Umständen die "falsche" Interpretation der Daten wählt, eine zu der es nur einen umständlichen oder eventuell sogar gar keinen Algorithmus gibt.

9.6 Rekursive Daten: logische kontra physische Struktur der Daten

Ausdrücke: Auswertung nach ihrer logischen Struktur

Die Auswertung von Ausdrücken der Form

`(2+3), ((2-1)*3), ((1+2)*(5-1))` etc.

kann nach den Prinzipien für rekursive Funktionen erfolgen. Wir stellen zunächst fest, dass Ausdrücke eine Struktur haben, bei der komplexe Ausdrücke stets aus einfacheren aufgebaut sind. Es sind also rekursive Daten. Der Wert eines Ausdrucks kann entsprechend ihrer Struktur berechnet werden.

Ein Ausdruck ist eine Ziffer oder ein aus einfacheren Ausdrücken zusammengesetzter geklammerter Ausdruck.

- Ziffern: den Wert von Ziffern zu bestimmen ist einfach.
- geklammerte Ausdrücke: Beide Teilausdrücke werden ausgewertet und die Teilwerte dann mit dem Operator verknüpft. (Reduktion der Auswertung des Gesamtausdrucks auf die Auswertung der Teilausdrücke.)

Damit erhält man eine erste Skizze für eine Funktion zur Ausdrucksauswertung:

```
int ausdruckWert (Ausdruck a) {
  if (a ist Ziffer)
    return (Zifferwert (a))
  else {
    sei a == (a1 op a2)
    int w1 = ausdruckWert (a1);
```

```
      int w2 = ausdruckWert (a2);
      return op(w1, w2);
    }
  }
```

Physische Struktur der Ausdrücke

Ausdrücke kann man einfach als Zeichenketten (Typ `string`) abspeichern. Für unsere Zwecke passt das aber ganz und gar nicht: Der rekursive Algorithmus basiert auf der Tatsache, dass Ausdrücke aus Teilausdrücken aufgebaut sind. Er ruft sich selbst zur Verarbeitung von *Teilausdrücken* auf.

Kollision der logischen und der physischen Struktur der Ausdrücke

Neben dem Gesamtausdruck müssen auch Teilausdrücke verarbeitet werden. Dieser Zerlegungsaspekt des Algorithmus macht aber Probleme: Wir haben ja leider

- nicht aus *Teilausdrücken* zusammengesetzte *Gesamtausdrücke* (logische Struktur) vorliegen,
- sondern aus *Zeichen* zusammengesetzte *Zeichenketten* (phsysische Struktur).

(Siehe Abbildung 27)

Abbildung 27: logische und physische Struktur von $((1-5)*2)$

Kurz die logische Struktur der Ausdrücke, auf der die Auswertung definiert ist, ist eine andere, als die der tatsächlich zu verarbeitenden Daten. Dieses Problem übersieht man leicht, da der Mensch von der Natur mit einem sehr wirkungsvollen System zur Mustererkennung ausgestattet wird. – Wir sehen im Gegensatz zu unseren Programmen in $((2+3)*(4-(6-2)))$ sofort neben der physischen Struktur "Folge von Zeichen", auch die logische Struktur "Ausdruck aus Unterausdrücken".

Analyse der Zeichenketten als Voraussetzung der Berechnung

Man könnte nun versuchen aus der physischen Struktur die logische zu berechnen, d.h. beispielsweise aus dem Gesamtausdruck einen Teilausdruck als Zeichenkette heraus zu holen, um ihn dann an einen rekursiven Aufruf zu übergeben.

```
ausdruckWert ("(2 + ((1 + 1) * 3))") =
  ausdruckWert ("2") + ausdruckWert ("((1 + 1) * 3)")
```

Diese Methode würde allerdings erfordern, dass die Ausdrücke (in Form von Zeichenketten) analysiert und korrekt zerlegt werden. Eine korrekte Zerlegung ist aber etwa genauso schwierig, wie die Bestimmung ihres Wertes. Genauer gesagt ist die Bestimmung des Wertes ein Leichtes, wenn eine Zerlegung vorliegt.

Ausdrücke sind Zeichenfolgen ab einer bestimmten Position

Statt Zeichenfolgen zu zerlegen, könnte man sie durch eine Anfangs- und eine Endposition charakterisieren:

```
ausdruckWert ("(2 + ((1 + 1) * 3))", 0, 12) =
  ausdruckWert ("(2 + ((1 + 1) * 3))", 1, 1)
         + ausdruckWert ("(2 + ((1 + 1) * 3))", 3, 11)
```

Der Wert der Zeichenfolge `(2+((1+1)*3))` von Position 0 bis einschließlich 12 ist der Wert des Zeichens an Position 1 plus der Wert der Zeichen von Position 3 bis einschließlich Position 11.

Bei unseren einfachen Ausdrücken brauchen wir aber die Endposition für die Wertbestimmung gar nicht zu kennen. Die Ausdrücke haben eine feste Struktur, die ihr Ende unzweifelhaft festlegt. (Eine Ziffer ist ein Zeichen lang; ein geklammerter Ausdruck ist so lang wie die beiden Teilausdrücke, plus zwei Klammern, plus ein Operator.)

Die letzte Position eines Teilausdrucks wird lediglich gebraucht, um die Startposition des nächsten Teilausdrucks zu bestimmen. Das muss aber nicht im voraus geschehen. Jede Auswertung eines Teilausdrucks kann einfach nebenbei ihre Zeichen mitverfolgen und so die erste Position ihres Nachfolgers liefern: Wir definieren eine rekursive Funktion

```
int ausdruckWert (string s, int &pos);
```

Sie liefert den Wert des Ausdrucks in `s`, der an Position `pos` beginnt und verschiebt dabei `pos` hinter diesen Ausdruck. `pos` wird also als Startpunkt des zu berechnenden Ausdrucks innerhalb von `s` in die Funktion übergeben und kommt als sein Endpunkt heraus. Dazu liefert die Funktion den Wert des durch die Anfangsposition bestimmten Ausdrucks.

Beispiel:

```
pos = 0;
ausdruckWert ("(2 + ((1 + 1) * 3))", pos):
```

liefert den Wert 8 und setzt `pos` auf 13, d.h. auf die erste Position hinter dem Ausdruck, der an Position 0 beginnt. Die Funktion `ausdruckWert` kann recht einfach rekursiv definiert werden und wie folgt agieren:

```
pos = 0;
ausdruckWert ("(2 + ((1 + 1) * 3))", pos):

  ++pos; // Klammer-auf ueberlesen

  // 1-ten Teilausdruck auswerten:
  v1 = ausdruckWert ("(2 + ((1 + 1) * 3))", pos);

  // Jetzt: v1= 2, pos = 2

  // Operator bestimmen:
  op = "(2 + ((1 + 1) * 3))".at(pos);

  // Jetzt: op= +
  ++pos; // Operator ueberlesen

  // 2-ten Teilausdruck auswerten:
  v2 = ausdruckWert ("(2 + ((1 + 1) * 3))", pos)

  // Jetzt: v2= 6 = ((1 + 1) * 3), pos = 12
  ++pos; // Klammer-zu ueberlesen

  // Operation ausfuehren, Wert berechnen:
  return (v1 op v2);
```

`pos` muss dabei ein Referenzparameter sein, denn jeder rekursive Aufruf hat ja `pos` entsprechend der Länge des von ihm zu bearbeitenden Teil–Strings weiter zu schieben.

Dieses Konzept setzen wir in Programmcode um und erhalten folgendes Gesamtprogramm zur Ausdrucksauswertung:

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

using namespace std;

int  ausdruckWert (string, int &);
bool istZiffer (char);
int  zifferWert (char);
```

```
int  anwende (char, int, int);

int main () {
  string a;    //Eingelesene Zeichenkette
  int pos = 0; //Start des zu bestimmenden Ausdrucks
  cin >> a;
  cout << ausdruckWert (a, pos);
  cout << endl;
}

int ausdruckWert (string zeichen, int &pos) {
  if (istZiffer(zeichen.at(pos))) {

    ++pos;
    return  zifferWert(zeichen.at(pos-1));

  } else if (zeichen.at(pos) == '(') {
    int v1, v2, res;
    char op;

    ++pos;
    v1 = ausdruckWert (zeichen, pos);
    op = zeichen.at(pos);
    ++pos;
    v2 = ausdruckWert (zeichen, pos);
    res = anwende (op, v1, v2);
    ++pos; //')'
    return res;
  }
  cout << "Ausdruck ist nicht OK an pos " << pos << " : ( erwartet \n";
  return -99;
}

bool istZiffer (char c) {
  return (c >= '0' && c <= '9');
}
int zifferWert (char c) {
  return (c - '0');
}

int anwende (char op, int v1, int v2) {
  switch (op) {
  case '+': return v1 + v2;
  case '-': return v1 - v2;
  case '*': return v1 * v2;
  case '/': return v1 / v2;
  default:
    cout << "Ausdruck ist nicht OK, Operator +-*/ erwartet! \n";
  }
  return -99;
}
```

9.7 Vor– und Nachbedingungen

Funktionen liefern unter bestimmten Bedingungen bestimmte Werte

Bei einer Funktion, die zwei Zahlen addiert, haben wir eine klare Beziehung zwischen den Parametern und dem Ergebnis. Das Ergebnis ist die Summe der Parameter und ansonsten wird nichts vorausgesetzt und sollte nichts passieren.

Die Funktion `ausdruckWert` von oben hat ebenfalls zwei Argumente und liefert einen Wert. Die Beziehungen zwischen Argumenten und Wert ist hier aber etwas komplexer:

- Die beiden Parameter sind nicht beziehungslos irgendeine Zeichenkette und irgendein `int`–Wert. Der `int`–Wert muss die Startposition eines korrekten Ausdrucks innerhalb der Zeichenkette sein!

- Die Funktion soll auch nicht nur einen Wert liefern, sie muss auch dafür sorgen, dass ihr zweiter Parameter nach dem Aufruf auf eine bestimmte Position zeigt, nämlich hinter den erkannten und berechneten Teilausdruck!

Vorbedingung und Nachbedingung

Eine *Vorbedingung* (engl. *precondition*) ist eine Voraussetzung unter der eine Funktion korrekt zu funktionieren verspricht.

In unserem Beispiel verspricht die Funktion `ausdruckWert` (bzw. ihr Autor) *nicht* mit *allen* `string`– und `int`–Parametern zu funktionieren, sondern *nur mit solchen*, welche die Vorbedingung erfüllen, dass der `int`–Parameter die *Startposition* eines *korrekten* Ausdrucks innerhalb der Zeichenkette ist.

Die *Nachbedingung* (engl. *postcondition*) ist das, was die Funktion zu leisten verspricht. Im Gegenzug zur Vorbedingung, für deren Erfüllung die Aufrufstelle sorgen muss, garantiert die Funktion `ausdruckWert` (bzw. ihr Autor) nicht nur einen richtigen Ausdruckswert zu liefern, sondern auch, dass der Parameter `pos` auf das erste Zeichen hinter dem erkannten Ausdruck gesetzt wird.

Vor– und Nachbedingung stellen somit einen *Vertrag* zwischen einer Funktion (bzw. ihrem Autor) und dem Rest des Programms (bzw. deren Autorin) dar: *Wenn der Programmrest für die Einhaltung der Vorbedingung sorgt, dann garantiert die Funktion ein korrektes Ergebnis und die Nachbedingung!*

Kommentare zu Funktionen

Komplexe Funktionen sollten stets mit folgenden Kommentaren versehen werden:

- *Parameter*: Beschreibung der Parameter,
- *Ergebnis*: Ergebniswert der Funktion.
- *Vorbedingung*: Bedingung an globale Variablen und/oder Parameter, die für das korrekte Arbeiten der Funktion erfüllt sein müssen.
- *Nachbedingung:* Wirkung der Funktion auf globalen Variablen und/oder Referenzparametern.

Ein Beispiel ist:

```
// ausdruckWert
//
// bestimmt den Wert einer Zeichenfolge, interpretiert als
// arithmetischer Ausdruck.
//
// Parameter:
//   - zeichen : string der den Ausdruck enthaelt
//   - pos     : Startposition des Ausdrucks innerhalb
//               von zeichen
// Ergebnis:
//   Wert des Ausdrucks, der ab Position pos im
//   Parameter zeichen zu finden ist.
// Vorbedingung:
//   ab Position pos muss sich in zeichen ein vollstaendiger
//   und korrekter Ausdruck befinden.
// Nachbedingung: pos zeigt auf das erste Zeichen
//   hinter dem erkannten Ausdruck

int ausdruckWert (string zeichen, int &pos) {
  ...
}
```

9.8 Funktionen als Parameter

Funktionsparameter

Funktionen können als aktuelle und formale Parameter einer Funktion auftreten. Einer Sortierfunktion kann beispielsweise das Sortierkriterium (Art des Vergleichs) als Parameter übergeben werden:

```
typedef bool CompFunc (int, int); // Typ der Vergleichsfunktionen

// Zwei Vergleichsfunktionen (vom Typ CompFunc):
bool kleiner  (int a, int b) {return a<b; }
bool groesser (int a, int b) {return a>b; }

void sort_a (CompFunc vergl);     // Deklaration der Sortierfunktion

int a[10] = {0,9,1,6,4,6,8,2,7,6};

int main () {
  //a aufsteigend sortieren:
  sort_a (kleiner);

  //a absteigend sortieren:
  sort_a (groesser);
}

void sort_a (CompFunc vergl) {   // Definition der Sortierfunktion
  int m, t;

  for (int i=0; i<10; ++i) {
    m = i;
    for (int j=i+1; j<10; ++j)
      if (vergl (a[j], a[m]))
        m =j;
    t    = a[i];
    a[i] = a[m];
    a[m] = t;
  }
}
```

Funktionstyp

Der Typ des Funktionsparameters wird in der Deklaration:

```
typedef bool CompFunc (int, int);
```

festgelegt. Man erkennt das gleiche Muster wie bei der Deklaration von Feldtypen: Eine Typdeklaration hat dort die Form einer Variablendeklaration mit vorgestelltem `typedef`. Hier bei Funktionstypen ist es das gleiche:

Eine Deklaration eines Funktionstyps hat die Form einer Funktionsdeklaration mit vorgestelltem `typedef`.

Statische Bindung

Die *statische Bindung* von Namen bleibt auch bei der Übergabe von Funktionsparametern erhalten. Mit dem Begriff "statische Bindung" ist gemeint, dass die Zuordnung von Namen zu den Dingen, die sie bedeuten, allein von der Programmstruktur gesteuert ist und nicht von irgendwelchen aktuellen Situationen zur Laufzeit.

Beispielsweise ist in

```
void f (int x) { y=x+y; }
```

der Name `y` frei und bezieht sich somit auf die globale Variable `y` (soweit vorhanden.) Dies ergibt sich aus dem Text des Programms ("statisch"). Das bleibt auch so, wenn `f` als Parameter in eine Funktion transportiert wird, in der es ein lokales `y` gibt und dann von dort aufgerufen wird.

Beispiel:

```
typedef void Func (int);

int y = 0;        // globales y
void f (int x) { y=x+y; }  // greift immer auf das globale y zu
void g (Func pf) {
  int y = 10;     // lokales y
  pf (2);         // Die Bedeutung der freien Namen
}                 // in pf wird statisch festgelegt
```

```
int main () {
  int y = 100;    // lokales y
  g (f);
}
```

Hier wird der Aufruf von `pf` in `g` das globale `y` verändern und nicht das lokale `y` in `g`, das doch "näher" an der Aufrufstelle liegt und auch nicht das `y` in `main`.

9.9 Typen mit Ein–/Ausgabe–Methoden

Ein– und Ausgabe von Vektoren

C++ bietet in Form der `iostream`–Bibliothek mächtige Konstrukte um Daten einzulesen und auszugeben. Unter anderem enthält diese Bibliothek die Definition der Eingabe– und Ausgabe–Ströme `cin` und `cout`. Die beiden sind vom Typ `istream` *(input–stream)* bzw. `ostream` *(output–stream)*.

Es spricht nichts dagegen `cin` und `cout` als aktuelle Parameter an eine Funktion oder Methode zu übergeben. Damit kann man dann leicht einen Typ `Vektor` definieren, der auch in der Lage ist Vektoren – also Werte vom Typ `Vektor` – einzulesen und auszugeben:

```
#include <iostream.h>

struct Vektor {
  float x;
  float y;
  void drucke (ostream &);
  void lese   (istream &);
};

//Vektor-Ausgabe im Format (x,y)
void Vektor::drucke (ostream &s) {
  s << "(" << x << ", " << y << ")";
}

//Einlesen im gleichen Format:
void Vektor::lese (istream &s) {
  char c;
  s >> c;
  if (c != '(') {cout << "FEHLER: ( erwartet !\n"; return;}
  s>>x;
  s>>c;
  if (c != ',') {cout << "FEHLER: , erwartet !\n"; return;}
  s>>y;
  s>>c;
  if (c != ')') {cout << "FEHLER: ) erwartet !\n"; return;}
}
```

Die Verwendung dieser Definitionen ist einfach:

```
...
Vektor v;
...
v.lese (cin);
...
v.drucke(cout);
cout << endl;
```

Will man immer nur von `cin` lesen und auf `cout` schreiben, dann müssen die beiden auch nicht als Parameter übergeben werden.

```
...
void Vektor::drucke () {
  cout << "(" << x << ", " << y << ")";
}
```

```
...
Vektor v;
v.drucke();
...
```

Geraden

Eine Gerade kann in Punktrichtungsform mit Hilfe von zwei Vektoren dargestellt werden:

$\vec{x} = \vec{p} + \lambda\vec{a}$

Ein selbst definierter Typ `Gerade` kann also aus zwei Vektoren bestehen. Selbstverständlich kann auch dieser Typ mit passenden Ein–/Ausgabe–Funktionen ausgestattet werden:

```
struct Gerade {
  Vektor p;
  Vektor a;
  void drucke (ostream &);
  void lese   (istream &);
};
//-------------------------------------------------------
void Gerade::drucke (ostream &s) {
  s << "[";
  p.drucke(s);
  s << " + ";
  a.drucke(s);
  s << "]";
}
//-------------------------------------------------------
void Gerade::lese (istream &s) {
  char c;
  s >> c;
  if (c != '[') {cout << "FEHLER: [ erwartet !\n"; return;}
  p.lese(s);
  s>>c;
  if (c != '+') {cout << "FEHLER: + erwartet !\n"; return;}
  a.lese(s);
  s>>c;
  if (c != ']') {cout << "FEHLER: ] erwartet !\n"; return;}
}
```

Parallelität von Geraden

Zwei Geraden

$\vec{p_1} + \lambda\vec{a_1}$ und $\vec{p_2} + \lambda\vec{a_2}$

sind parallel, wenn sie die gleiche Richtung haben. Wenn also $\vec{a_2}$ und $\vec{a_1}$ kollinear sind, es darum ein x gibt mit:

$\vec{a_2} = x\vec{a_1}$

Dies wiederum ist der Fall, wenn die Determinante $\left| \vec{a_1}\vec{a_2} \right|$ den Wert 0 hat.

```
Vektor v_minus (Vektor a, Vektor b) {
  Vektor res;
  res.x = a.x - b.x;
  res.y = a.y - b.y;
  return res;
}
//-------------------------------------------------------
bool kollinear (Vektor a, Vektor b) {
  return (a.x * b.y - a.y * b.x) == 0;
}
//-------------------------------------------------------
bool parallel (Gerade g1, Gerade g2) {
  return kollinear (g1.a, g2.a);
}
```

Zwei Geraden sind gleich, wenn sie parallel sind und dazu noch $\vec{a_1}$ und $\vec{p_2} - \vec{p_1}$ kollinear sind:

```
bool g_gleich (Gerade g1, Gerade g2) {
  return parallel (g1, g2) && kollinear (g1.a, v_minus (g2.p, g1.p));
}
```

Schnittpunkt zweier Geraden

Zwei Geraden $\vec{p_1} + \lambda\vec{a_1}$ und $\vec{p_2} + \lambda\vec{a_2}$ schneiden sich, wenn es zwei Werte s und t gibt mit:

$$\vec{p_1} + s\vec{a_1} = \vec{p_2} + t\vec{a_2}$$

Der Schnittpunkt ist dann $\vec{p_1} + s\vec{a_1}$. s und t können aus der Gleichung

$$s\vec{a_1} - t\vec{a_2} = \vec{p_2} - \vec{p_1}$$

bestimmt werden. D.h. als Lösung des Gleichungssystems:

$$\begin{aligned} s * a_1.x - t * a_2.x &= p_2.x - p_1.x \\ s * a_1.y - t * a_2.y &= p_2.y - p_1.y \end{aligned}$$

(Die Unbekannten sind hier s und t)

Wir setzen voraus, dass die Geraden nicht parallel sind und lösen das Gleichungssystem nach s auf:

$$\begin{aligned} s &= \frac{D_s}{D} \\ D_s &= \vec{p_2} - \vec{p_1},\ -\vec{a_2} \\ D &= \vec{a_1},\ -\vec{a_2} \end{aligned}$$

Wir beginnen die Schnittpunktbestimmung mit einer Funktion zur Berechnung der Determinante von zwei Vektoren:

```
float determinante (Vektor a, Vektor b) {
  return a.x * b.y - a.y*b.x;
}
```

Damit lässt sich zu zwei Geraden g_1 und g_2 der Wert s bestimmen:

```
float D_s = determinante (v_minus (g2.p, g1.p), sv_mult (-1, g2.a));
float D   = determinante (g1.a, sv_mult (-1, g2.a));
float s   = D_s / D;
```

`sv_mult` soll hier einen Skalar mit einem Vektor multiplizieren. Mit diesem s kann der Schnittpunkt ($\vec{p_1} + s * \vec{a_1}$) sofort angegeben werden:

```
Vektor schnittPunkt = v_plus (g1.p, sv_mult (s, g1.a));
```

`v_plus` ist die triviale Funktion, die zwei Vektoren addiert.

Gesamtprogramm

Ein Gesamtprogramm zum Test der Schnittfunktion ist:

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

struct Vektor {
  float x;
  float y;
  void drucke (ostream &);
  void lese   (istream &);
};

struct Gerade {
```

```
  Vektor p;
  Vektor a;
  void drucke (ostream &);
  void lese   (istream &);
};

void Vektor::drucke (ostream &s) {
  s << "(" << x << ", " << y << ")";
}
void Vektor::lese (istream &s) {
  ... wie oben ...
}
void Gerade::drucke (ostream &s) {
  ... wie oben ...
}
void Gerade::lese (istream &s) {
  ... wie oben ...
}

Vektor v_minus (Vektor a, Vektor b) {
  Vektor res;
  res.x = a.x - b.x;
  res.y = a.y - b.y;
  return res;
}
Vektor v_plus (Vektor a, Vektor b) {
  Vektor res;
  res.x = a.x + b.x;
  res.y = a.y + b.y;
  return res;
}
bool kollinear (Vektor a, Vektor b) {
  return (a.x * b.y - a.y * b.x) == 0;
}
bool parallel (Gerade g1, Gerade g2) {
  return kollinear (g1.a, g2.a);
}
bool g_gleich (Gerade g1, Gerade g2) {
  return parallel (g1, g2) && kollinear (g1.a, v_minus (g2.p, g1.p));
}
float determinante (Vektor a, Vektor b) {
  return a.x * b.y - a.y*b.x;
}
Vektor sv_mult (float s, Vektor v) {
  Vektor res;
  res.x = s*v.x;
  res.y = s*v.y;
  return res;
}
//-------------------------------------------------
Vektor schnitt (Gerade g1, Gerade g2) {
  float D_s = determinante (v_minus (g2.p, g1.p), sv_mult (-1, g2.a));
  float D   = determinante (g1.a, sv_mult (-1, g2.a));
  float s   = D_s / D;
  Vektor res;
  res       = v_plus (g1.p, sv_mult (s, g1.a));
  return res;
}
//-------------------------------------------------

using namespace std;

int main (){
  Gerade g1;
  Gerade g2;
  cout << "Bitte die erste Gerade eingeben" << endl;
```

```
  g1.lese (cin);
  cout << "Bitte die zweite Gerade eingeben" << endl;
  g2.lese (cin);
  cout << "\n g1: ";
  g1.drucke(cout);
  cout << "\n g2: ";
  g2.drucke(cout);
  cout << endl;

  if (g_gleich (g1, g2))
    cout << "g1 und g2 sind gleich\n";
  else if (parallel (g1, g2))
    cout << "g1 und g2 sind parallel\n";
  else {
    cout << "Die Geraden schneiden sich im Punkt: "
    Vektor schnittpunkt = schnitt (g1, g2);
    schnittpunkt.drucke(cout);
    cout << endl;
  }
}
```

Namensräume als Module

Eine Kollektion zusammengehöriger Definitionen bezeichnet man oft als *Modul*. Die “geometrischen” Definitionen von oben sind ein Modul in diesem Sinn. Man sollte sie darum auch in einem Namensraum zusammenfassen:

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

Namesapce GeoNS {
  struct Vektor {
    float x;
    float y;
    void drucke (ostream &);
    void lese   (istream &);
  };

  struct Gerade {
    Vektor p;
    Vektor a;
    void drucke (ostream &);
    void lese   (istream &);
  };

  void Vektor::drucke (ostream &s) {
    ...
  }
  ... etc. ...
}

int main (){
  GeoNS::Gerade g1;
  GeoNS::Gerade g2;
  std::cout << "Bitte die erste Gerade eingeben" << std::endl;
  g1.lese (std::cin);
  ...
}
```

Selbstverständlich kann man auch Using–Direktiven oder –Deklarationen einsetzen.

9.10 Überladung von Operatoren

Operatoren mit selbst definierter Wirkung

Funktionen müssen nicht unbedingt Namen haben, sie können auch als Operatoren definiert werden. Damit können sehr schöne Programme geschrieben werden, beispielsweise das Hauptprogramm des Beispiels von oben:

```
int main (){
  Gerade g1;
  Gerade g2;
  cout << "Bitte die erste Gerade eingeben" << endl;
  ... wie oben ...

  if (g1 == g2)          //Gleichheit von Geraden
    cout << "g1 und g2 sind gleich\n";
  else if (g1 || g2)     //Parallelitaet
    cout << "g1 und g2 sind parallel\n";
  else {
    Vektor schnittpunkt = g1 * g2;
    cout << "Die Geraden schneiden sich im Punkt: "
    schnittpunkt.drucke(cout);
    cout << endl;
  }
}
```

Wir müssen dazu nur einige Funktionsdefinitionen so modifizieren, dass sie nicht über einen Namen, sondern über einen Operator aktiviert werden:

```
//Vektor-Subtraktion
Vektor operator- (Vektor a, Vektor b) {
  Vektor res;
  res.x = a.x - b.x;
  res.y = a.y - b.y;
  return res;
}

//Vektor-Addition
Vektor operator+ (Vektor a, Vektor b) {
  Vektor res;
  res.x = a.x + b.x;
  res.y = a.y + b.y;
  return res;
}

//Parallelitaet von Geraden
bool operator|| (Gerade g1, Gerade g2) {
  return kollinear (g1.a, g2.a);
}

//Gleichheit von Geraden
bool operator== (Gerade g1, Gerade g2) {
  return (g1 || g2) && kollinear (g1.a, g2.p - g1.p);
}

//Multiplikation Skalar * Vektor
Vektor operator* (float s, Vektor v) {
  Vektor res;
  res.x = s*v.x;
  res.y = s*v.y;
  return res;
}

// Schnittpunkt zweier Geraden
Vektor operator* (Gerade g1, Gerade g2) {
  float D_s = determinante (g2.p - g1.p, -1 * g2.a);
  float D   = determinante (g1.a,        -1 * g2.a);
```

```
  float s = D_s / D;
  Vektor res;
  res = g1.p + s*g1.a;
  return res;
}
```

Überladung von Operatoren – Operator als freie Funktion

Die verschiedenen Funktionen können jetzt mit einem Operator aktiviert werden. Statt

```
Vektor v_plus (Vektor a, Vektor b) ...
```

und

```
v3 = v_plus (v1, v2);
```

schreibt man

```
Vektor operator+ (Vektor a, Vektor b) ...
```

und

```
v3 = v1 + v2; // das gleiche wie: v3 = operator+ (v1, v2);
```

Es stört dabei nicht, dass "+" schon als Addition von `int` und `float` Zahlen definiert ist. C++ erlaubt ja die *Überladung* von Funktionen: Solange sie am Typ ihrer Argumente eindeutig unterschieden werden können, dürfen beliebig viele Funktionen den gleichen Namen tragen. Dies gilt auch für Operatoren. Der Operator "+" kann darum mit der Vektoraddition überladen werden.

Im Beispiel wird "*" zur Multiplikation von zwei `floats`, zur Multiplikation eines `float` mit einem `Vektor` und zur Berechnung des Schnittpunkts von zwei Geraden benutzt. Diese Überladung ist nicht nur bei "normalen" Funktionen mit einem Namen, sondern generell bei allen Funktionen möglich – auch solchen mit einem Operator als Namen.

Der Compiler entscheidet mit Hilfe des Typs der Argumente darüber, welche der möglichen Funktionen aktuell gemeint ist. Diese Entscheidung erfolgt statisch, d.h. zur Übersetzungszeit.

In C++ sind alle Operatoren überladbar außer denen, die sich mit Gültigkeitsbereichen beschäftigen: "`::`" (Bereichsauflösungsoperator), "`.`" (Punktoperator) und "`.*`" (noch nicht besprochen).

Operatoren als Methoden

Die überladenen Operatoren in den Beispielen oben waren freie Funktionen. Ein Operator kann auch als Methode definiert werden. Beispiel:

```
struct Vektor {
  float x;
  float y;
  Vektor operator+ (Vektor);
};

//Vektor-Addition als Methode (ein Argument weniger !):
Vektor Vektor::operator+ (Vektor b) {
  Vektor res;
  res.x = x + b.x;  // statt res.x = a.x + b.x beim freien Operator
  res.y = y + b.y;
  return res;
}

...
Vektor v1, v2, v3;

v3 = v1 + v2;  // das gleiche wie   v3 = v1.operator+ (v2);
```

Operatoren als Methoden haben immer einen Parameter weniger als ihre freien Brüder. Das liegt daran, dass

```
v3 = v1 + v2;
```

bei einem *freien Operator* als

```
v3 = operator+ (v1, v2);
```

interpretiert wird. Bei einer *Methode* dagegen als:

```
v3 = v1.operator+ (v2);
```

Die Rolle des ersten Parameters übernimmt "das Objekt selbst". Statt `a.x` – das `x` des ersten Parameters – taucht darum in der Methode nur `x` – d.h. "mein `x`" – auf.

Vergleich von Verbunden

Weiter oben bei der Einführung der Verbundtypen sagten wir, dass die Zuweisung (d.h. der `operator=`) die einzige Operation ist, die auf Verbunden vordefiniert ist und dass es insbesondere keinen vordefinierten Vergleich von Verbunden gibt. Jetzt wissen wir, wie diese Lücke zu füllen ist. Sollen zwei Verbunde verglichen werden, dann muss für den entsprechenden Typ eine freie Funktion oder Methode `operator==` definiert werden. Z.B.:

```
namespace GeoNS {
  struct Vektor {
    float x;
    float y;
    bool   operator== (Vektor);
  };

  //Vektor-Vergleich
  bool Vektor::operator== (Vektor b) {
    if (x == b.x && y == b.y) return true;
    else                      return false;
  }
  ...
}

using namespace GeoNS;
int main () {
  Vektor v1, v2;
  ...
  if (v1 == v2) ...  // das gleiche wie  if (v1.operator==(v2)) ...
  ...
}
```

9.11 Übungen

Aufgabe 1

Schreiben Sie ein Programm zur Lösung eines linearen Gleichungsystems:

$$\mathbf{A}\vec{x} = \vec{b} \qquad wobei: \quad \mathbf{A} = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix} \quad \vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \quad \vec{b} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

Die Werte von $\mathbf{A}$ und $\vec{b}$ sind aus einer Datei einzulesen. Falls das Gleichungssystem keine eindeutige Lösung hat, ist eine entsprechende Meldung auszugeben.

Verwenden Sie zur Lösung dieser Aufgabe folgendes Verfahren:

$$x_1 = \frac{|\mathbf{A}^{[1]}|}{|\mathbf{A}|} \quad x_2 = \frac{|\mathbf{A}^{[2]}|}{|\mathbf{A}|} \quad x_3 = \frac{|\mathbf{A}^{[3]}|}{|\mathbf{A}|}$$

falls $|\mathbf{A}| \neq 0$, ansonsten hat das System keine eindeutige Lösung.

$|\mathbf{A}|$ ist die Determinante von $\mathbf{A}$ und
$\mathbf{A}^{[i]}$ ist die Matrix $\mathbf{A}$, bei der die i–te Spalte durch $\vec{b}$ ersetzt wurde, z.B.:

$$\mathbf{A}^{[2]} = \begin{pmatrix} a_{1,1} & b_1 & a_{1,3} \\ a_{2,1} & b_2 & a_{2,3} \\ a_{3,1} & b_3 & a_{3,3} \end{pmatrix}$$

Die Determinanten der Matrizen werden nach der Regel von Sarrus berechnet:

Falls

$$\mathbf{A} = \begin{pmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{pmatrix}$$

dann gilt:

$$|\mathbf{A}| = a_1b_2c_3 + b_1c_2a_3 + c_1a_2b_3 - a_3b_2c_1 - b_3c_2a_1 - c_3a_2b_1$$

Gehen Sie systematisch vor und benutzen Sie nach Möglichkeit Funktionen und Methoden.

Aufgabe 2

Eine Integer–Liste ist eine Folge von Integer–Werten. Beispiele für Integer–Listen: Leere Liste: $<>$, eine Liste mit drei Elementen: $< 1, -2, 7 >$, eine Liste kann Wiederholungen enthalten: $< 1, 2, 2, 2, 5 >$.

1. Definieren Sie eine Datentyp `Liste` für Integer–Listen mit maximal 10 Elementen.
2. Geben Sie für die Listen $<>$, $< 1, -2, 7 >$, $< 1, 2, 2, 2, 5 >$ jeweils an, wie sie von Ihrem Datentyp dargestellt werden.
3. Erweitern Sie Ihren Datentyp `Liste` um eine Methode `leere`, die die Liste, für die sie aktiviert wird, leert. Die leere Liste kann also mit

   ```
   Liste l; l.leere();
   ```

   erzeugt werden.
4. Schreiben Sie eine freie Funktion `leereListe` die eine leere Liste erzeugt. Ein Beispiel für die Verwendung dieser Funktion ist:

   ```
   Liste l; l = leereListe();
   ```
5. Schreiben Sie eine Methode `anhaenge` die einen übergebenen Integer–Wert an ihre Liste anhängt. Beispiel: Wenn `l` die Liste $< 2, 5 >$ darstellt, dann wird mit `l.anhaenge(6)` die Darstellung der Liste $< 2, 5, 6 >$ erzeugt.

Aufgabe 3

1. Die Funktion $f(x) = x^2$ kann rekursiv wie folgt definiert werden:

   $$\begin{aligned} f(1) &= 1 \\ f(n) &= f(n-1) + 2 * (n-1) + 1 \end{aligned}$$

   Schreiben Sie eine entsprechende rekursive Funktion.

2. Die Funktion n über k mit $0 \leq n \leq k$ kann rekursiv definiert werden:
   $\left(\begin{array}{c} n \\ k \end{array} \right) = \left(\begin{array}{c} n-1 \\ k \end{array} \right) + \left(\begin{array}{c} n-1 \\ k-1 \end{array} \right)$
   Des weiteren gilt:
   $\left(\begin{array}{c} n \\ 0 \end{array} \right) = \left(\begin{array}{c} n \\ n \end{array} \right) = 1$
   (Nach dieser Definition wird das Pascalsche Dreieck berechnet!) Schreiben Sie eine entsprechende rekursive Funktion.

3. Folgende Funktion berechnet die Darstellung einer positiven ganzen Zahl als Ziffernfolge

   ```
   string ziffern (unsigned int x) {
     string res = "";
     while (x > 0) {
       res = char('0'+ x%10) + res;
       x = x/10;
     }
     return res;
   }
   ```

   Geben Sie eine rekursive Version dieser Funktion an.

Aufgabe 4

1. Betrachten Sie die im Skript vorgestellte rekursive Funktion zur Berechnung von $n!$.

   ```
   int fak (int x) {
     if (x == 0) return 1;
       else return x*fak(x-1);
   }
   ```

   Geben Sie an, wie mit ihr 3! berechnet wird. Zeichnen Sie die erzeugten und wieder vernichteten Funktionsinstanzen mit all ihren lokalen Variablen und formalen Parametern, geben Sie an welche aktuellen Parameter übergeben und welche Werte zurück geliefert werden!

   Verfolgen Sie die Abarbeitung im Debugger! Fragen Sie gegebenenfalls die Übungsbetreuung um Rat! Achtung: Die Fähigkeit zur elementaren Debugger–Bedienung ist *Lernziel der Veranstaltung* und damit Klausur–relevant!

2. Was bedeutet der Begriff “Kontrollstruktur”. Geben Sie ein Beispiel an. Erläutern Sie warum Rekursion als Kontrollstruktur bezeichnet wird.

3. Welchen Wert liefert die im Skript vorgestellte rekursive Funktion zur Berechnung von $n!$ bei einem negativen Argument?

4. Welcher Aufruf von einer der beiden folgenden Funktionen wird wohl mehr Speicherplatz benötigen:

   ```
   int f1 (int x) {
     int a[10];
     for (int i=1; i<10; i++)
       a[i] = i;
     return a[x%10];
   }
   ...
   cout << f1 (10);
   ```

   ```
   int f2 (int x) {
     if (x < 0) return x+10;
     return f2 (x-1);
   }


   ...
   cout << f2 (10);
   ```

Begründen Sie Ihre Antwort!

5. Betrachten Sie die beiden Funktionen:

```
int f1 (int a, int b) {
  if (a == 0) return 0;
  return b + f1(a-1, b);
}
```

```
int f2 (int a, int b) {
    if (b == 0)  return a;
    return 1 + f2(a, b-1);
}
```

Was wird von beiden Funktionen berechnet? Geben Sie Definitionen der berechneten Funktionen an!

Formulieren Sie jeweils eine sinnvolle Vorbedingung und geben Sie eine äquivalente iterative Funktion an, also eine in der die Rekursion durch eine Schleife ersetzt wurde.

6. Vergleichen Sie `f1` und `f2` von oben mit folgendem `f` (mit der Hilfsfunktion `ff`). Welche – bekannte oder unbekannte – Funktion wird hier realisiert:

```
void ff (int &a, int &b) {
  if (b == 0) return;
  else {
    a = a+1;
    b = b-1;
    ff (a, b);
  }
}
int f (int a, int b) {
  ff (a, b);
  return a;
}
```

7. Erläutern Sie wie jede `do-while`–Schleife durch Rekursion ersetzt werden kann.

8. Ist es möglich, dass eine Funktion an sich selbst als aktuellen Parameter übergeben wird? Ist also

```
... f (... f, ...);
```

für irgendeine Funktion `f` ein zulässiger Ausdruck in C++? Wenn ja geben Sie ein Beispiel an, wenn nein, erläutern Sie bitte warum dies nicht möglich ist!

9. (Zum Knobeln) Ist es möglich, dass eine Methode einer Verbundvariablen diese Variable als aktuellen Parameter hat? Ist also so etwas wie:

```
... s.m(... s ... ) ...
```

möglich? Wenn ja geben Sie ein Beispiel an, wenn nein, erläutern Sie bitte warum dies nicht möglich ist!

10. Eine rekursive Funktion, von der Form:

```
T1 f (T2 x) {
   if (P(x))
     return H(x);
   else return f(R(x));
}
```

bei der der rekursive Aufruf die letzte Aktion der Funktion ist, kann leicht in eine äquivalente Schleife umgesetzt werden. In welche? (Die rekursive Berechnung des ggt ist von dieser Form.)

11. C++ benutzt *statische Bindung*.[28] Das bedeutet, dass einem Bezeichner seine Definition statisch (d.h. unveränderlich und zur Übersetzungszeit) zugeordnet wird.

Erläutern Sie dies an folgendem Beispiel, in dem Sie angeben, welches `y` – innerhalb der Funktion `f`, die als `pf` an `g` übergeben wurde – verändert wird. Was wird ausgegeben?

```
typedef void Func (int);

int y = 1;        // globales y
```

[28] Statische Bindung ist in C++ die Regel, in manchen fortgeschrittenen – und später noch zu behandelten – Konstrukten wird jedoch statt dessen *dynamische Bindung* eingesetzt.

```
void f (int x) {
  y = x + y;
  cout << y << endl;
}

void g (Func pf) {
  int y = 11;     // lokales y
  pf (y);
  cout << y << endl;
}

int main () {
  int y = 111;
  g (f);
  cout << y << endl;
}
```

12. Was wird von obigem Programm ausgegeben, wenn `f` einen *Referenzparameter* hat und diesen verändert?

```
void f (int & x) {
  x = x + y;
  cout << y << endl;
}
```

Aufgabe 5

1. Geben Sie jeweils eine Typ– und Variablendeklarationen an für eine 2×3 und eine 3×2 Matrix. Zeichnen Sie ein Schaubild, das verdeutlicht, wie die Werte der beiden Matrizen entsprechend Ihrer Definitionen im Speicher abgelegt sind. Geben Sie an mit welchem Ausdruck auf ein Matrix–Element zugegriffen wird.
2. Schreiben Sie ein Programm in dem eine 3×4 Matrix so mit Werten belegt wird, dass a[i][j] den Wert $10i+j$ hat.
3. Ergänzen Sie dieses Programm um eine Prozedur, welche die Summe aller Werte in dieser Matrix ausdruckt. Die Funktion soll auf die Matrix als globale Variable zugreifen.
4. Gelegentlich ist es lästig, dass in C++ der erste Index eines Feldes den Wert "0" hat und nicht "1". Dem kann durch einen eigenen Typ für Matrizen abgeholfen werden.

   Definieren Sie einen Verbundtyp `Matrix_3_2` mit den Methoden

   `float wert (int, int)` und

   `void setze (int, int, float).`

   In einer Variablen von diesem Typ soll eine 3×2 Matrix abgelegt werden. Die beiden Methoden sollen, mit den üblichen Angaben der Indexwerte beginnend mit 1, den Zugriff (Speichern und Lesen) auf die Matrixelemente ermöglichen (intern werden sie die Index–Parameter auf die Indexwerte mit Null beginnend umsetzen):

   ```
   Matrix_3_2 a;
   float x;
   a.setze (1, 1, 5.0);
   x = a.wert (1,1);
   ```

   Hier wird *das erste Element der ersten Zeile* $a_{1,1}$ mit dem Wert 5 belegt und dieser Wert anschließend ausgelesen.
5. Ergänzen Sie das im Skript vorgestellte Programm zur Gauss–Elimination um die notwendigen Test– und Zeilentausch–Operationen.

Aufgabe 6

Schreiben Sie eine Funktion `forAll_in_a (Func)`, die eine beliebige andere Funktion f auf alle Elemente einer globalen 2×3–Matrix `a` anwendet. Beispielsweise soll `forAll_in_a(f)` alle Elemente in `a` verdoppeln, wenn f als `float f(float x){return 2*x;}` definiert ist.

Kann die Funktion f wirklich beliebig sein? Geben Sie Beispiele für gültige Definitionen an!

Aufgabe 7

1. Erweitern oder modifizieren Sie die im Skript vorgestellte Funktion zur Ausdrucksanalyse derart, dass sie Leerzeichen innerhalb eines Ausdrucks verarbeiten kann.
2. Erweitern oder modifizieren Sie die Ausdrucksanalyse so, dass sie nicht nur Ziffern, sondern auch aus Ziffern zusammengesetzte Zahlen als Bestandteile eines Ausdrucks verarbeiten kann.

Aufgabe 8

Schreiben Sie eine (rekursive) Funktion `bWert` zur Auswertung von Ausdrücken mit Brüchen. `bWert` soll so etwas wie

```
(+1/8 - ((+1/24 + -10/3) * -12/8))
```

berechnen können. Die Ausdrücke sollen vollständig geklammert sein, aber Leerzeichen enthalten dürfen.

Spezifizieren Sie die zu verarbeitenden Ausdrücke mittels einer Grammatik. Realisieren Sie einen Verbund–Typ `Bruch` zur Darstellung von Brüchen. Die arithmetischen Operationen auf Brüchen sind als überladene Operatoren zu definieren. Definieren Sie einen Namensraum `BruchNS` in dem Sie alle Definitionen platzieren, die zu der Bearbeitung von Brüchen gehören.

Aufgabe 9

Die Funktion `string ohneLeerZ(string);` soll zu einer Zeichenkette die entsprechende Zeichenkette ohne Leerzeichen liefern. Beispiel:

```
string s = ohneLeerZ(" H al  l   o!");
cout << s << endl;                    // Ausgabe: Hallo
```

Zeichenketten können als rekursive Menge aufgefasst werden und auf dieser kann ein rekursiver Algorithmus für `ohneLeerZ` definiert werden.

1. Definieren Sie Zeichenketten als rekursive Menge: Was ist die Basismenge, wie werden neue aus gegebenen einfacheren Zeichenketten gebildet.
2. Definieren Sie eine rekursive Funktion `ohneLeerZ` die diesem Aufbau der Zeichenketten folgt.

9.12 Lösungshinweise

Aufgabe 1

Siehe Skript!

Aufgabe 2

1. Wir nehmen einen Verbund als Darstellung der Menge. Eine Komponente des Verbunds ist ein Feld. Es nimmt die Elemente der Liste auf. Da wir belegte von unbelegten (mit Zufallswerten belegten) Feldkomponenten nicht unterscheiden können, müssen wir auf irgendeine Art kenntlich machen welche Werte im Feld gültig sind. Wir nehmen dazu einfach eine Zählvariable:

```
struct Liste {
   int l[12];  // Listenelemente: l[0]..l[z-1]
   int z;      // Zahl der Listenelemente
};
```

2. Die leere Liste wird durch einen Verbund mit $z = 0$ dargestellt. Die Liste $< 1, -2, 7 >$ wird durch einen Verbund mit $z = 3$ und $1, -2, 7$ in `l[0]`, `l[1]`, `l[2]` dargestellt.

3.
```
struct Liste {
   int l[12];  // Listenelemente: l[0]..l[z-1]
   int z;      // Zahl der Listenelemente
   void leere();
};

void Liste::leere() { z=0; }
```

4.
```
Liste leereListe() { Liste res; res.leere(); return res; }
```

5.
```
struct Liste {
   int l[12];  // Listenelemente: l[0]..l[z-1]
   int z;      // Zahl der Listenelemente
   void leere();
   void anhaenge (int);
};

void Liste::leere() { z=0; }
void Liste::anhaenge (int x) {
  if ( z == 12 ) return; // Liste ist voll
  else { l[z] = x;  ++z; }
}
```

Aufgabe 3

1.
```
int f (int x) {
  if (x==1) return 1;
  else      return f(x-1) + 2*(x-1) + 1;
};
```

2.
```
int f (int n, int k) {
  if (k==0 || n == k) return 1;
  else                return f(n-1, k) + f (n-1, k-1);
};
```

3.
```
string ziffernRek (unsigned int x) {
  if ( x == 0) return "";
  else
    return ziffernRek ( x/10 )
           + char('0' + x%10);
}
```

Aufgabe 4

1. Siehe Skript, Vorlesung, Übung.
2. Siehe Skript, Vorlesung.
3. In einem *idealen* Rechner mit unendlich großem Wertebereich: Endlos–Rekursion.

   In einem *realen* Rechner soll die Funktion für $x < 0$ folgendes liefern:

   $$x \cdot (x-1) \cdot (x-2) \cdot \ldots \cdot (\texttt{INT\_MIN}+1) \cdot \texttt{INT\_MIN} \cdot \texttt{INT\_MAX} \cdot (\texttt{INT\_MAX}-1) \cdot \ldots \cdot 3 \cdot 2 \cdot 1$$
   $$= \left(\prod_{i=\texttt{INT\_MIN}}^{x} i \right) \cdot \left(\prod_{i=1}^{\texttt{INT\_MAX}} i \right)$$

   Wegen Überlaufs bei der Multiplikation wird aber kein korrektes Ergebnis dieses Ausdrucks berechnet.
4. Die rekursive Funktion `f2` wird mehr Speicherplatz benötigen als `f1`, da dort 12–mal eine `int`–Variable angelegt wird: Speicherplatz für den formalen Parameter `x` von 12 Instanzen der Funktion `f2`: `f2(10), f2(9), ... , f2(0), f2(-1).`
5. Es handelt sich um Multiplikation und Addition. `f1` führt die Multiplikation auf die Addition zurück und `f2` die Addition auf die Addition von 1. Die Funktionsdefinitionen sind:

   $$\begin{aligned} 0 \cdot b &= 0 \\ (n+1) \cdot b &= b + n \cdot b \end{aligned}$$

   $$\begin{aligned} a + 0 &= a \\ a + (n+1) &= (a+n) + 1 \end{aligned}$$

   Vorbedingungen: bei `f1` muss der erste, bei `f2` der zweite Parameter ≥ 0 sein, damit die Rekursion wie vorgesehen endet und die Funktion korrektes Ergebnis liefert.

   Äquivalente iterative Funktionen inklusive Testprogramm sind:

```
#include <iostream>
using namespace std;
int f1 (int a, int b) {   // rekursiv
  if (a == 0) return 0;
  return b + f1(a-1, b);
}
int f1_i (int a, int b) { // iterativ
  int res = 0;
  while (a != 0) {
    res = res + b;
    a--;
  }
  return res;
}
//-----------------------------------
int f2 (int a, int b) {   // rekursiv
  if (b == 0)  return a;
  return 1 + f2(a, b-1);
}
int f2_i (int a, int b) { // iterativ
  int res = a;
  while (b != 0) {
    res++;
    b--;
  }
  return res;
}
//-----------------------------------
int main () {
  int x,y;
```

```
  cin >> x;
  cin >> y;
  cout << f1 (x, y) << " " << f1_i (x, y) << endl;
  cout << f2 (x, y) << " " << f2_i (x, y) << endl;
}
```

6. `f` mit der Hilfsfunktion `ff` addiert, es ist äquivalent zu `f2`.
7. Jede Schleife der Form

   ```
   do
     A(x);
   while (P(x));
   ```

   kann in die äquivalente rekursive Funktion

   ```
   void wF (T x) {
     A(x);
     if (P(x))
       wF(x);
   }
   ```

   umgesetzt werden.
8. Eine Funktion kann *nicht* an sich selbst als aktuellen Parameter übergeben werden, da der Typ einer Funktion niemals der gleiche sein kann wie der Typ einer ihrer Parameter. Ein `f` vom Typ `Func` mit `Func`–Parameter:

   ```
   typedef ... Func ...;
   ... f (... Func, ...);
   ```

   kann es nicht geben. `f` kann hier niemals den Typ `Func` haben – wie auch immer dieser definiert ist – und somit nicht an sich selbst übergeben werden. [29]
9. Natürlich kann eine Methode einer Verbundvariablen diese Variable als aktuellen Parameter haben. Beispiele sind leicht zu konstruieren.
10. Eine rekursive Funktion, von der Form:

    ```
    T1 f (T2 x) {
       if (P(x))
         return H(x);
       else return f(R(x));
    }
    ```

    kann in die äquivalente Schleife

    ```
    T1 f (T2 x) {
       while (!P(x)) {
         x = R(x);
       }
       return  H(x);
    }
    ```

    umgesetzt werden. Leider sind solche einfach–rekursiven Funktionen sehr selten. Eines der wenigen sinnvollen Beispiele ist die GGT–Berechnung:

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

    struct T {
      int a;
      int b;
    };
    ```

[29] Mit einem *Cast* (im C–Stil) kann man allerdings die Übergabe erzwingen.

```
bool P (T x) { return x.a == x.b; }
int  H (T x) { return x.a; }

T R (T x) {
  T res;
  if (x.a > x.b) {
    res.a = x.a - x.b;
    res.b = x.b;
  } else {
    res.b = x.b - x.a;
    res.a = x.a;
  }
  return res;
}
//-----------------------------------
int fr (T x) { // rekursive Version
  if (P(x))
    return H(x);
  else return fr(R(x));
}
//-----------------------------------
int fi (T x) { // iterative Version
  while (!P(x)) {
    x = R(x);
  }
  return  H(x);
}
//-----------------------------------
int main () {
  int a, b;
  T   x;
  cin >> a;
  cin >> b;
  x.a = a; x.b = b;
  cout << fr (x) << endl;
  cout << fi (x) << endl;
}
```

Achtung! Die Äquivalenz des Paars

```
int f1 (int a, int b) {
  if (a == 0) return 0;
  return b + f1(a-1, b);
}
int f1_i (int a, int b) {
  int res = 0;
  while (a != 0) {
    res = res + b;
    a--;
  }
  return res;
}
```

von oben (Nr. 5) kann *nicht* verallgemeinert werden!

11. C++ benutzt *statische Bindung*. Der Bezeichner `y` ist frei in `f`, `f` ist eine freie Funktion. `y` bezieht sich darum immer auf das globale `y`, egal wann und wo `f` aufgerufen wird. Durch Nachdenken und/oder Experimentieren lässt sich herausfinden, welche Werte vom Programm ausgegeben werden.

12. Ausgabe bei `f` mit *Referenzparameter*: Durch Nachdenken und anschließendes Experimentieren lässt sich das herausfinden.

Aufgabe 5

1. Eine 2×3 Matrix hat 2 Zeilen und 3 Spalten (= 3 Elemente pro Zeile). Sie kann also dargestellt werden durch

```
typedef float Zeile[3]; // Eine Zeile hat 3 Elemente
typedef Zeile Matrix[2]; //Eine Matrix hat 2 Zeilen
Matrix m;
```

oder

```
typedef float Zeile[3]; // Eine Zeile hat 3 Elemente
Zeile m[2]; //Die Matrix hat 2 Zeilen


float m[2][3]; // m hat 2 Komponenten, mit jeweils 3 Subkomonenten
```

oder

```
typedef Matrix[2][3];
Matrix m;
```

Die Definitionen für die 3×2–Matrix werden entsprechend konstruiert. Der Zugriff auf die Elemente wird im Skript ausführlich beschrieben.

2. Das Programm in dem eine 3×4 Matrix so mit Werten belegt wird, dass `a[i][j]` den Wert $10i + j$ hat, muss zwei geschachtelte Schleifen von folgender Art enthalten:

```
for (int i = 0; i<3; i++)
  for (int j = 0; j<4; j++)
     a[i][j] = 10*i + j;
```

3. Matrizen sollten jetzt keinerlei Schwierigkeiten mehr bereiten. Wenn doch, dann wiederholen Sie den Stoff bitte noch einmal!

4. Etwa wie folgt:

```
struct Matrix_3_2 {
   float v[3][2];
   float wert (int, int);
   void setze (int, int, float);
};

float Matrix_3_2::wert (int i, int j) {
   return v[i-1][j-1];
}

... setze entsprechend ...
```

5. Die Ergänzung des im Skript vorgestellten Programms zur Gauss–Elimination um die notwendigen Test– und Zeilentausch–Operationen sollte keine Probleme bereiten.

Aufgabe 6

```
#include <iostream>
using namespace std;
// Eine (m x n) Matrix hat
// - m Zeilen, d.h m Elemente in einer Spalte
// - n Spalten, d.h. n Elemente in einer Zeile

const int M = 2;
const int N = 3;

typedef int   Zeile [N];  // N Elemente in einer Zeile
typedef Zeile Matrix[M];  // M Zeilen in der Matrix

// Aequivalent aber weniger uebersichtlich waere
// typedef int Matrix[M][N];

// Eine Testmatrix
Matrix a = {{10, 11, 12}, {21, 22, 23}};

// Testfunktionen
```

```
int doppel (int x) { return 2*x; }
int incr   (int x) { return x+1; }

// Typ der Funktionen
typedef int Func (int);

void forAll_in_a (Func f) {
  for (int i=0; i<M; ++i)
    for (int j=0; j<N; ++j)
      a[i][j] = f (a[i][j]);
}

int main () {
  forAll_in_a (doppel);
  forAll_in_a (incr);
  for (int i=0; i<M; ++i) {
    for (int j=0; j<N; ++j)
      cout << a[i][j] << " ";
    cout << endl;
  }
}
```

Parameter- und Rückgabetyp der Funktion f müssen dem Typ der Matrix-Elementen entsprechen! (`<int>` im Beispiel oben.)

Aufgabe 7

1. Leerzeichen in Ausdrücken.
   Zunächst sollten Sie beachten, dass Zeichenketten die Leerzeichen enthalten nicht einfach mit `cin` eingelesen werden können:

   ```
   string a;    //Eingelesene Zeichenkette
   //STATT, cin >> a; :
   getline (cin, a);
   ```

   Ansonsten müssen einfach an allen relevanten Stellen die Leerzeichen überlesen werden. Beispielsweise in einer Schleife:

   ```
   while (zeichen.at(pos) == ' ') pos++;
   ```

   die an die richtigen Stellen im Programm platziert wird.

2. Zahlen statt Ziffern.
   Die Umsetzung einer Zeichenkette in eine Zahl wurde in der Vorlesung besprochen. Eine entsprechende Funktion, z.B.:

   ```
   int zahlWert (string s, int & pos) {
     int res = 0;
     while (pos < s.length() && istZiffer (s.at(pos))){
       res = 10*res + (s.at(pos) - '0');
       pos++;
     }
     return res;
   }
   ```

   kann hier eingesetzt werden.

Aufgabe 8

Kein Lösungshinweis.

Aufgabe 9

Kein Lösungshinweis.

Literatur

[1] Ulrich Breymann
C++ Einführung und professionelle Programmierung
6. Auflage Hanser 2001

[2] Bjarne Stroustrup
The C++ Programming Language
Third Edition
Addison–Wesley 1997

[3] Stanley B. Lippman, Josée Lajoie
C++ Primer
Addison–Wesley 1998

[4] Nicolai M. Josuttis
The C++ Standard Library
Addison–Wesley 1999

Stichwortverzeichnis