Vorlesungsmodul Programmieren 1
- VorlMod Prog1 -

Matthias Ansorg

02. Oktober 2001 bis 23. Mai 2003

Zusammenfassung

Studentische Mitschrift zur Vorlesung Programmieren 1. Die Gliederung richtet sich vollständig nach dem offiziellen Skript zur Vorlesung [1]. Bis inkl. Kapitel 1.3 ist der Inhalt ein Mitschriebe zur Vorlesung Programmieren 1 bei Prof. Lauwerth im Wintersemester 2001/2002 an der FH Gießen-Friedberg. Das vorliegende Dokument enthält alle Aufgabenlösungen zu [1], auch ein paar Fehlerkorrekturen (Kapitel C) und ergänzende Erklärungen bei schwierig zu verstehendem Stoff. Es sollte stets zusammen mit 52 C++-Quelldateien geliefert werden, in denen alle Programme enthalten sind, die in den Übungsaufgaben aus [1] zu schreiben waren.

- **Bezugsquelle:** Das vorliegende Dokument steht im Internet zum freien Download bereit: `http://matthias.ansorgs.de/InformatikDiplom/Modul.Prog1.Letschert/Prog1.pdf`.
- **Lizenz:** Dieses Dokument ist public domain, darf also ohne Einschränkungen oder Quellenangabe für jeden beliebigen Zweck benutzt werden, kommerziell und nichtkommerziell; jedoch enthält es keinerlei Garantien für Richtigkeit oder Eignung oder sonst irgendetwas, weder explizit noch implizit. Das Risiko der Nutzung dieses Dokumentes liegt allein beim Nutzer selbst. Einschränkend sind außerdem die Urheberrechte der verwendeten Quellen zu beachten.
- **Korrekturen:** Fehler zur Verbesserung in zukünftigen Versionen, sonstige Verbesserungsvorschläge und Wünsche bitte dem Autor per e-mail mitteilen: Matthias Ansorg, ansis@gmx.de.
- **Format:** Die vorliegende studentische Mitschrift wurde mit dem Programm LyX (graphisches Frontend zu LaTeX) unter Linux erstellt und als PDF-Datei exportiert.
- **Dozent:** Prof. Dr. Letschert.
- **Verwendete Quellen:** basierend auf [1].
- **Tipps zur Klausur:**

Inhaltsverzeichnis

1 Erste Programme

C++ wurde 1998 durch ISO standardisiert (sog. ANSI-Standard); diese Programme können also weltweit ausgetauscht werden. Java ist eine neuere Programmiersprache, die ähnlich C++ ist, aber natürlich einige Besonderheiten hat. Java jedoch hat den Nachteil, dass es nicht durch einen internationalen Standard festgelegt ist: Microsoft will mit C# (»C-Sharp«) ein Konkurrenzprodukt durchsetzen. Java kann also durchaus sich in der Praxis nicht durchsetzen.

Die objektorientierten Eigenschaften von C++ werden hauptsächlich erst im 2. Semester dargestellt.

Jedes C++-Programm ist eine Sammlung von Funktionen, von denen eine `main()` heißt. `main()` ist das Hauptprogramm. Beispiel:

```
// Programm führt Ein- und Ausgabe durch:
#include <iostream>
// Standardbezeichner verwenden:
using namespace std;
int main()
// Blöcke werden in { } gefasst:
  {
  // führt Ausgabe auf dem Bildschirm durch;
  // Zeichenketten stehen immer in ""
  // einzelne Zeichen können in '' stehen.
  // Ausgabe eines ": "\""
```

```
  // Ausgabe eines \: "\\"
  cout << "Hallo !";
  // endl ist das Zeichen end-of-line / carriage return
  // man könnte äquivalent schreiben:
  // cout << "\n\n";
  cout << endl << endl;
  // verlasse die Funktion und gebe Funktionswert 0
  // zurück, d.h. Verlauf ohne Fehler. Man sollte jeder
  // Funktion gemäß ANSI-Standard einen Rückgabewert geben,
  // auch main(), trotz dass dies nicht zwingend ist.
  return 0;
  }
```

Im Beispielprogramm vorkommende Schlüsselwörter (»Vokabeln«; können nicht verändert werden, ohne die Semantik zu ändern): `using`, `namespace`, `std`, `int`, `main`, `cout`, `endl`, `return`. Kommentare werden mit `//` am Anfang jeder Zeile geschrieben. Kommentare sind sehr wichtig, weil Sourcecode i.A. mehr gelesen als geschrieben wird. Beim Arbeiten in einer Firma z.B. gibt es code reviews; jeder reviewer muss einverstanden sein, damit der Code akzeptiert wird. Escape-Sequences (»Fluchtzeichen«) sind Steuerzeichen, die nicht direkt über die Tastatur erreicht werden können. Dazu gehören auch Zeichen `"\ooo"` (dreistellige Oktalnummer des ASCII-Zeichens) und `"\xXX"` (`x` und zweistellige Hexadezimalnummer des ASCII-Zeichens).

Der ASCII-Code besteht aus 7 Bit pro Zeichen und 1 Bit parity. Deutsche Umlaute sind nicht Teil des ASCII, sondern der quasi-Standard-Erweiterung durch IBM, die das parity bit dazu missbraucht.

Formulierung dieses Algorithmus in der Umgangssprache (als Pseudocode; dies ist ein Hilfsmittel, eine Lösung aufzuschreiben, bevor man programmiert):

```
main()
  begin
    gebe Text "Hallo !" am Bildschirm aus;
    gebe zwei Zeilenvorschübe am Bildschirm aus;
    gebe 0 als Funktionswert zurück;
  end
```

Pseudocode ist die Art der Notation, mit der der Software-Entwickler dem Programmierer mitteilt, wie er ein Programm schreiben soll. Es gibt Ansätze zu Programmen, die Pseudocode in verschiedene Programmiersprachen übersetzen, also die Arbeit der Programmierer abnehmen. In der Literatur vorkommender Pseudocode richtet sich standardmäßig sehr nach Pascal-Notation. Man kann jedoch auch eigene Pseudocode-Worte definieren.

Beispiel eines C++-Programms mit zwei Funktionen: »Funktionstabelle« mit $y = f(x) = x^2$.

Die Notation in Pseudocode ist sinnvoll und empfehlenswert, weil man sich dann noch nicht mit den Anforderungen der Programmiersprache beschäftigen muss. Der Pseudocode kann als Kommentar beim Erstellen des C++-Codes übernommen werden (empfehlenswert, da so automatisch das Programm dokumentiert wird). Pseudocode:

```
// Hauptalgorithmus
main()
  begin
    // nochmal ist eine Marke
    nochmal:
    hole x von der Tastatur;
    berechne y=f(x);
    gebe x und y am Bildschirm aus;
    // wenn x=0 war, wird main() beendet:
    falls x ungleich 0 ist, gehe zu nochmal;
    // return 0:
    gebe 0 als Funktionswert zurück;
  end;
// f(x) als Teilalgorithmus
f(x)
  begin
```

```
  // return x*x;
  gebe x*x als Funktionswert zurück;
end
```

Es ist eine generelle Strategie, möglichst viele Aufgaben an andere Funktionen »abzudrücken« statt direkt selbst zu erledigen; deshalb wurde hier die Funktion y=f(x) verwendet, statt direkt y=x*x zu berechnen. Dies verbessert auch die Wartbarkeit des Programms, weil nur die Funktion und nicht das Hauptprogramm geändert werden muss; im Sinne von »don't touch running software«. Deshalb sollte man vor einer Änderung laufender Software immer eine Sicherungskopie durchführen. Heute programmiert man zuerst auf eine saubere Lösung und Struktur hin, nicht auf Geschwindigkeit; das tuning des Programms erfolgt wenn nötig im Anschluss.

Schreibtischtest, d.h. Ausführen des Programms mit dem menschlichen Gehirn als Prozessor, mit Hilfe einer Wertetabelle. Ein Schreibtischtest eines C++-Programms kann durchaus eine Aufgabe der Klausur sein.

Eingabe	x	y	Ausgabe
3	3	9	3 9
-4	-4	16	-4 16
0	0	0	0 0
return 0;			
Programmende			

Erstellung eines C++-Programms aus dem obigen Pseudocode:

```
// Name: xfkt.cpp
#include<iostream>
using namespace std;
// Deklaration der Funktion f:
// eine Funktion, die ein Argument vom Typ float hat und einen
// Wert vom Typ float zurückgibt.
float f(float x);
// Deklaration der Funktion main:
// eine Funktion ohne Argument, die einen Wert vom Typ int
// zurückliefert; direkt mit Deklaration und Definition
// (der Block)
int main()
{
  // Deklaration von zwei Variablen vom Typ float
  float x,y;
  // Markendeklaration:
  nochmal:
  // hole x von der Tastatur
  cout << endl << "x (Stop für x=0.0)?";
  // Zuweisung der Eingabe an Variable x:
  cin >> x;
  // Aufruf der Funktion f(x), ablegen des Wertes in y
  y=f(x);
  cout << "x=" << x << " y=" << y;
  if(x!=0.0) goto nochmal;
  return 0;
}
// Definition der Funktion f, die oben deklariert wurde
float f(float x)
{
  return x*x;
}
```

Die Verwendung von `goto` ist unbedingt zu vermeiden! Es wird also bald ersetzt werden, so wie Schleifen behandelt werden. Die Verwendung von `namespace std` heißt, dass nicht vor jeder Anweisung `cin` bzw. `cout` »`std:`« wiederholt werden muss. Jede Definition einer Funktion ist auch eine Deklaration; deshalb könnte statt der anfänglichen Deklaration der Funktion `f` auch sofort die Definition stehen; dies ist jedoch unüblich, weil durch die vorhergehende Definition erreicht wird, dass der Sourcecode von oben nach unten gelesen werden kann (nicht wie Pascalprogramme von unten nach oben).

Die Eingabe zu obigem Programm kann auch als String »`5.0 -3.0 0.0`« o.ä. erfolgen; die `cin`-Funktion holt sich bei jedem Durchlauf dann aus dem Tastaturpuffer sukzessive die Argumente, die sie braucht.

Die Einrückung des Sourcecodes ist wichtig und wird auch in der Klausur bewertet, denn diese Schreibweise ist wichtiger Standard in der Industrie.

Schreibtischtest dieses C++-Programms:

	Eingabe / Ausgabe	x	y
		undefiniert	undefiniert
	x (Stop für x=0.0)? 5.0	5.0	25.0
	x=5.0 y=25.0	5.0	25.0
	x (Stop für x=0.0)? -3.0	-3.0	9.0
	x=-3.0 y=9.0	-3.0	9.0
	x (Stop für x=0.0)? 0.0	0.0	0.0
	x=0.0 y=0.0	0.0	0.0
Programmende			

1.1 Algorithmen und Programme

Ein Algorithmus beschreibt durch eine Folge von Anweisungen $A_1, A_2, A_3, \ldots$ das Wie, wie man vom Problem zu einer Lösung kommt, analog bei unseren Algorithmen von der Eingabe zur Ausgabe. Ein Programm ist ein Algorithmus in einer Programmiersprache. Darstellung durch ein T-Diagramm:

Problem	Algorithmus	Lösung
	Prozessor	

Syntax: legt fest, welche Zeichenfolge sich C++-Programm nennen darf, d.h. wo der C++-Compiler keine Fehlermeldung ausgibt. Syntaxfehler sind all die Fehler, die der Compiler erkennt. Im Studium wird von Anfang an kompiliert mit `g++ -pedantic -Wall <datei.cpp>`, um sich hart an den ANSI-Standard zu halten (`-pedantic`) und entsprechende Warnungen zu sehen (`-Wall`).

Semantik: legt die Bedeutung fest von dem, was man geschrieben hat.

Beispiel zu Syntax und Syntaxfehlern im Deutschen

- »ich schreibe ein programm«
- »ich programm schreibe ein«
- »ein programm ich schreibe«
- »schreibe ein programm ich«

Es gibt $4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$ Möglichkeiten, diese vier Worte anzuordnen. Nur der erste Satz ist syntaktisch richtig, jedoch bleibt die Semantik in allen Kombinationen erhalten und verstehbar. Kleine Syntaxfehler bei C++-Programmen in der Klausur sind möglich, werden nicht als Fehler gerechnet.

Beispiel zur Semantik im Deutschen

- »Inge und Bernd sahen die Alpen (oder setze »Alten«) als sie nach Italien flogen.«

Dieser Satz ist im Deutschen syntaktisch korrekt, aber semantisch nicht eindeutig: flogen Inge und Bernd, oder die Alpen (bzw. Alten) nach Italien?

Unterschiedliche Syntax bei gleicher Semantik sind auch folgende Ausdrücke:
`2+4` Infix-Notation
`2,4,+` Postfix-Notation
`+(2,4)` Prefix-Notation

Beim kompilieren mit `g++` (aus der `gcc`: GNU Compiler Collection) ergibt sich folgendes Verfahren:

- `g++` übersetzt den C-Quellcode in Assembler; der Assemblercode kann mit g++ -S <datei.cpp> in einer Datei abgelegt werden.
- Der Assembler (erkennbar an der Ausgaben `as` beim Compilieren) übersetzt den Assemblercode in ein Maschinenprogramm.
- Der Linker `ld` (erkennbar an der Ausgaben `ld` beim Compilieren) überführt das Maschinenprogramm in eine vom Betriebssystem ausführbare Datei.

Beispiel eines kleinen Programms in mehreren höheren Programmiersprachen In C und C++ und Java

```
//...
x=x+y;
if (x<y-5)
  z=2.5*sin(x)/y;
y=0;
//...
```

In Pascal

```
#...
x:=x+y;
if (x<y-5) then
  z:=2.5*sin(x)/y;
y:=0;
#...
```

Beispiel: Aufsummieren von Zahlen. Es soll eine beliebige Anzahl von ganzen Zahlen aufsummiert werden, bis eine 0 eingegeben wird. 0 wird als sog. »Stopper« verwendet. Bereich von Integer: `[INT_MIN, INT_MAX]`. Definiert in der Datei `limits.h`, also abhängig vom jeweiligen Compiler auf dem jeweiligen Betriebssystem. `g++` nimmt z.B. 4 Byte pro Integer, der Borland C Compiler 2 Byte pro Integer.

```
#include<iostream>
using namespace std;
int main()
{
  int x; // zum Einlesen des aktuellen Wertes
  int s; // enthält die aktuelle Summe
  s=0;
  nochmal:
  // lese x;
  cin >> x;
  s=s+x; // aufsummieren
  if (x!=0) goto nochmal;
  cout << "Summe ist " << s << endl;
  return 0;
}
```

Unter Ersetzung des `goto` durch eine `do while`-Schleife:

```
#include <iostream>
// Pause nach #include !
using namespace std;
int main()
{
  int x; // zum Einlesen des aktuellen Wertes
  int s; // enthält die aktuelle Summe; dies ist eine
         // Deklaration ohne Definition. Also:
  s=0;   // die Initialisierung als Definition, weil die
  // Variable zu Beginn einen zufälligen Wert hat. Man könnte
  // auch direkt bei der Deklaration initialisieren: int s=0;
  do
  // steht hier mehr als eine Anweisung (compound statement /
  // block), muss in geschweiften Klammern gefasst werden:
  {
    //lese x;
    cin >> x;
```

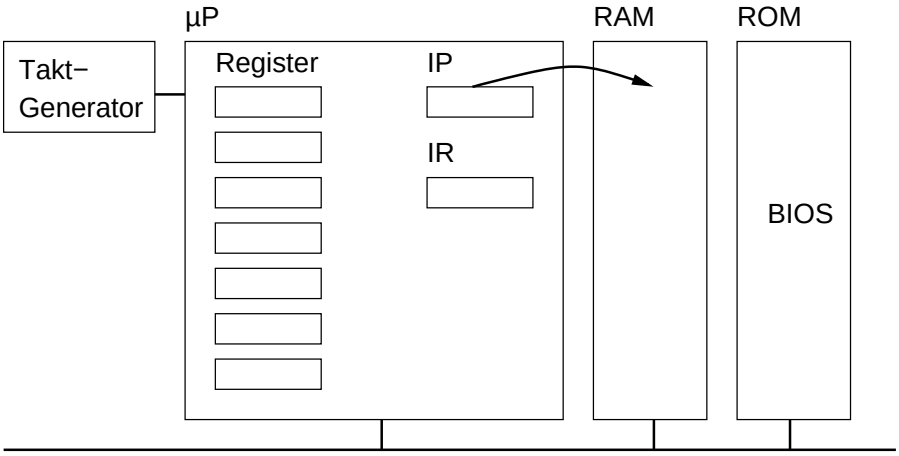

Abbildung 1: Prinzipieller Aufbau der Hardware

```
    s=s+x; // aufsummieren
  } while (x!=0)
  cout << "Summe ist " << s << endl;
  return 0;
}
```

Schreibtischtest:

Eingabe: `2 15 9 12 0`

x	?		2	15	9	12	0
s	?	0	2	17	26	38	

Ausgabe: `Summe ist 38`

Das T-Diagramm zu diesem Programm `sum.cpp` mit dem Benutzer selbst (»Ich«) als Prozessor (denn C-Programme sind für virtuelle Maschinen geschrieben, nicht für Mikroprozessoren. Um als Prozessor einen Mikroprozessor zu verwenden, muss man den Compiler aufrufen; der Prozessor muss stets diejenige Sprache verstehen, in der der Algorithmus geschrieben ist.):

Eingabe `2 15 9 12 0`	`sum.cpp`	Ausgabe `Summe ist 38`
	Ich	

1.2 Computer, Dateien, Programme

Das Arbeiten eines Computers kann man sich erklären am Schalenmodell. Der Zugriff ist nur über die äußerste Schale möglich. Die Schalen von innen nach außen:

Hardware. Siehe Abbildung 1. Der Speicher (RAM, d.h. random access memory, wahlfreier Zugriffsspeicher) ist byteweise adressiert, d.h. man kann auf einzelne Bytes, jedoch nicht auf einzelne Bits zugreifen. Der instruction pointer (IP) im Mikroprozessor nun zeigt immer auf die Adresse im RAM, die den nächsten abzuarbeitenden Befehl enthält. Das instruction register (IR) enthält die nächste auszuführende Anweisung, nachdem sie aus dem RAM hierhin kopiert wurde. Beim Einschalten des Rechners wird der instruction pointer zuerst automatisch auf die Adresse des Programms im ROM gesetzt, das das BIOS enthält. Das BIOS veranlasst die Prüfung des Rechners und bootet dann auch noch den Rechner mit dem installierten Betriebssystem durch den sog. Urlader. Dann wird der instruction pointer vom Urlader auf das im RAM enthaltene Betriebssystem umgesetzt.

Betriebssystem. So wird aus der reinen Hardware z.B. eine Linux-Maschine.

Anwendungsprogramme: Editor / Compiler / eigenes Programm. So wird aus der Linux-Maschine z.B. eine `emacs`-Maschine, eine `g++`-Maschine, eine `sum.cpp`-Maschine. Durch jedes Programm wird der Computer zu einer anderen Maschine, die anderes tun kann und anders bedient werden muss.

1.3 Programmerstellung

Der in dieser Vorlesung bei Prof. Lauwerth verwendete Compiler ist `gcc`. Eine HTML-Dokumentation zum Compiler gcc kann neben anderen Dokumenten im Verzeichnis von Prof. Lauwerth auf Alabama bezogen werden. Als Editor wird der kostenlos erhältliche `PFE` (Programmer's File Editor) verwendet, man darf aber natürlich auch andere Editoren verwenden.

Standardaufruf des Compilers mit Optionen:

`g++ -pedantic -Wall -o <name> %f`

Das heißt: prüfe auf Kompatibilität mit dem ANSI-Standard (`-pedantic`), zeige alle Warnungen (`-Wall`), nenne das Programm `<name>` (`-o`) und verwende als Quelldatei `%f` (nur in `PFE` ausgewertete Option). Vorsicht: niemals unter der Option `-o` den gleichen Dateinamen wie die Quelldatei angeben, sonst wird diese unwiederbringlich überschrieben!

Unter Windows muss der Pfad zum Compiler `g++` mit dem DOS-Befehl `set` der Umgebungsvariablen `PATH` hinzugefügt werden.

Mit dem DOS-Befehl `debug` kann man sich Dateien als Hexdump ausgeben lassen.

1.3.1 Kurzanleitung zu vim, auswendig zu lernen

`:help [<command>]` Die Online-Hilfe

`:help index` Index aller VIM-Befehle

`:help help-tags` alle verfügbaren Argumente zu `:help`

Strg-] Ist Strg-AltGr-9. Zu einem Tag springen, wenn der Cursor zwischen den beiden »|« des Tags steht. Ein Tag ist eine Textsprungmarke. Tags werden in C-Code für jede Funktion mit `ctags` erzeugt.

Strg-t Vom Sprung zu einem Tag zurückkehren.

`:qa!` `vim` ohne Speichern beenden. Auch `:q!`

`:wq` `vim` beenden, dabei alle Dateien speichern.

`:q` Aktuelles Fenster schließen, z.B. Hilfe-Fenster

ESC Befehlseingabe ohne Ausführung abbrechen

`x` In NormalMode: wie Entf in normalen Editoren

`i` In NormalMode: Wechseln in den InsertMode. Beenden mit ESC.

`dw` In NormalMode: delete word, von Cursor-Position bis Wortende

`d$` In NormalMode: delete bis zum Zeilenene

`d`<**num**><**object**> delete number <num> of objects. Objekte können sein: `$` (up to EOL), `w` (word), `e` (word without space) or `d` (line).

`u` in NormalMode: Undo

`U` in NormalMode: Undo für eine ganze Zeile

Strg-r in NormalMode: Redo von Undos

`p` in NormalMode: die letzte Löschung hinter dem Cursor einfügen (»put«)

`r`<**char**> in NormalMode: Ersetze (»replace«) das Zeichen unter dem Cursor mit <char>

`cw` in NormalMode: enter insert mode, deleting the rest of the word (change word)

`c`<**num**><**object**> change num objects from cursor position on.

<**num**>**Shift-g** go to line <num>

Shift-g go to bottom

`/`<**exp**> search for expression forward

`?<exp>` search for expression backward

`n` next search same direction

`N` next search opposite direction

`%` go to matching `(,[,{,},],)` for that one under the cursor

`:s/old/new` substitute one time. Wenn ein Backslash in Suchausdrücken vorkommen soll, so muss dieser escaped werden mit \\!

`:s/old/new/g` substitute all in a line

`:%s/old/new/g` substitute all in a file

`:%s/old/new/gc` substitute all in a file with confirmation

`:<num>,<num>s/old/new/g` substitute between two lines

`:!<command>` execute shell command

`:w <filename>` save under <filename>

`:<num>,<num> w <filename>` save <num> through <num> to <filename>

`:r <filename>` insert <filename> at cursor position

`o` open a line below cursor and enter insert mode

`O` open a line above cursor and enter insert mode

`a` append to character under cursor, entering insert mode

`A` append to EOL, entering insert mode

`R` enter replace mode (overwrite text)

`:set ic` ignore case option for search command

`:set hls` highlight search result

`:set is` incsearch. Sofortige Suche.

`:set cindent` automatisches Einrücken für C-Programme

`:set cinoptions=>3` Option für cindent: 3 Zeichen für eine normale Einrückung

1.3.2 Kurzanleitung zum Debugger `gdb`, auswendig zu lernen

1. Programm kompilieren mit Debug-Information:
   `g++ -g -Wall -pedantic <sourcefile> -o <outputfile>`
2. gdb starten
   `gdb <outputfile>`
3. breakpoint setzen
   `(gdb) break <funktionsname>`
4. Programm laufen lassen bis zum breakpoint
   `(gdb) run`
5. Untersuchungen
   - eine Variable einmal ausdrucken:
     `(gdb) print <variablenname>`
   - einen Ausdruck einmal ausdrucken
     `(gdb) print <ausdruck>`

- Variable bei jedem Schritt beobachten
  `(gdb) display <variable>`
- Variable nicht mehr beobachten:
  `(gdb) undisplay <displaynr>`
- Stack ansehen: backtrace
  `(gdb) bt`

6. Änderungen
   `(gdb) set variable <variable>=<expression>`
   `(gdb) set <variable>=<expression>`
   ändern Variablen des laufenden Programms
   `(gdb) set variable $<name>=<expression>`
   `(gdb) set $<name>=<expression>`
   ändern Hilfsvariablen zum Debuggen

7. Schrittweise weiterlaufen lassen ohne Abstieg in Funktionsaufrufe
   `(gdb) next` für eine Zeile bzw. `(gdb) nexti` für eine Anweisung
   oder mit Abstieg in Funktionsaufrufe
   `(gdb) step` für eine Zeile bzw. `(gdb) stepi` für eine Anweisung
   Die Anweisung, die im nächsten Schritt ausgeführt wird, wird im vorigen Schritt mit ausgegeben.

8. Programm selbständig weiterlaufen lassen bis zum nächsten breakpoint o.ä.:
   `(gdb) c`

1.4 Programme: Analyse, Entwurf, Codierung

Tipps zum Debuggen nach eigener Erfahrung: Man gehe unter allen Umständen systematisch bei der Fehlersuche vor. Man beginnt beim einfachsten Fehler (der einfachsten Aufgabe, die das Programm nicht mehr fehlerfrei erledigt) und behebt ihn, und wiederholt diesen Schritt, bis das Programm fehlerfrei ist. Unter dem X-Window System verwendet man am besten drei teilweise überlappende Fenster gleichzeitig: einen `xterm`, in dem man compiliert und das Programm testet; einen Editor, in dem man den Sourcecode sehen und bearbeiten kann, auch während dem Debugging; und einen `xterm` mit dem Debugger `gdb`.

Checkliste zum Debugging, nach Fehlerhäufigkeit entspr. pers. Erfahrung

1. syntaktische Fehler. Aufgedeckt durch den Compiler.

2. semantische Fehler: Fehler in der Implementierung des Algorithmus, d.i. in Kontrollstrukturen

   (a) Fehler in Sequenzen: return vor Anweisungen, die auch noch ausgeführt werden sollen

   (b) Fehler in der Indizierung: es entstehen zu große Indizes zum Zugriff auf Arrays und strings. Häufig der Fall in falsch geschriebenen Schleifen oder wenn eine Funktion Parameter als Indizes erhält und diese falsch sind. Ein Programm stürzt ab (»SIGABRT: Aborted« bzw. »Abgebrochen«), wenn auf strings mit zu großem Index zugegriffen wird!

   (c) Fehler in Schleifen: falsche Bedingungen

3. semantische Fehler: Fehler im Problemverständnis, d.h. falscher Algorithmus

   (a) fehlendes Verständnis über die Struktur eines von einer Rekursion zu verarbeitenden Elements, d.h. über den Syntax, wie sich dieses Element aus anderen Elementen (Nichtterminalsymbole) und sonstigen Terminalsymbolen aufbaut.

1.5 Variablen und Zuweisungen

1.6 Kommentare

1.7 Die Inklusions-Direktive

Verwendung automatischer Zusicherungen:

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

Auslesen der Wertebereiche integraler Datentypen auf der jeweiligen Maschine:

```
#include<climits>
```

1.8 Datentypen: Zeichen, Zeichenketten, ganze und gebrochene Zahlen

Zu jedem Datentyp gehören bei einer OOP-Sprache Methoden. So sind `at()` und `length()` Methoden, die zum Datentyp `string` gehören und mit jeder Variable vom Typ `string` aufgerufen werden können, z.B. `laenge=vorname.length()`.

Die Methode `at()` liefert ein `char`!

Zeichenketten (C-Strings) werden in "doppelte Hochkommas" gefasst, einzelne Zeichen (`char`) in 'einzelne Hochkommas'. Zeichenketten in doppelten Hochkommas sind immer Literale für C-Strings, d.h. für `char`-Felder, und *nicht* für C++-Strings, d.h. für Instanzen der Klasse `string`.

1.9 Mathematische Funktionen

Stehen nach `#include<cmath>` zur Verfügung.

- `float sqrt(float);`
- `float pow(float, float);`
- `float pow(float, int);`

1.10 Konstanten

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

1.11 Zusammenfassung: Elemente der ersten Programme

Wie kommt es, dass man in C++ Strings mit dem Operator + verknüpfen kann, trotz dass Strings nicht zum Sprachumfang von C++ gehören, sondern erst mit `#include <string>` eingebunden werden müssen? Weil die Definitionen aus der inkludierten Datei die Überladung des Operators + zur Verknüpfung von Strings definieren. Vgl. dazu Skript [1, Kapitel 8.8 »Überladung von Operatoren«].

1.12 C++ und C

Ein- und Ausgabe in C

- float-Wert einlesen:
  `scanf("%f",&f);`
- float-Wert ausgeben:
  `printf(" %f Grad Fahrenheit\n", f);`
- C-String einlesen:
  `scanf("%s", name);`
- C-String ausgeben:
  `printf("Name = %s", name);`

1.13 Übungen

2 Verzweigungen

2.1 Bedingte Anweisungen

- In C++-Programmen muss vor dem `else` einer zweiarmigen `if`-Anweisung ein Semikolon stehen!
- Globale Variable werden in C++ automatisch mit 0 initialisiert.

2.2 Flussdiagramme und Zusicherungen

2.3 Arithmetische und Boolesche Ausdrücke und Werte

- Boolesche Variable:
  `bool a=true;`
- In C++ können die Booleschen Werte mit `true` und `false` im Programmtext bezeichnet werden, z.B. `if (true) {}`.
- Beim Vergleich logischer Variablen gilt: `false < true`.
- Logisches NOT in C++: `!`
- Operatorenrangfolge in C++: Skript [1, S. 35]. Es gilt die Rangfolge:
  - unäre Operatoren (NOT, Vorzeichen)
  - Punktrechnung inkl. % (Modulo)
  - Strichrechnung
  - Relationen
  - Gleich und Ungleich
  - AND
  - OR
  - Zuweisung

2.4 Geschachtelte und zusammengesetzte Anweisungen

- Der C++-Sprachstandard definiert, dass ein `else` stets zum ersten vorhergehenden if desselben Programmblocks (d.h. einer Klammerung in `{}`) gehört. Deshalb sind folgende Ausdrücke von unterschiedlicher Bedeutung:

```
if (a>b) if (b>c) b=a; else a=b; // else bezieht sich auf das zweite if
if (a>b) {if (b>c) b=a;} else a=b; // else bezieht sich auf das erste if
```

- Dem Schlüsselwort `if` bzw. `else` folgt stets nur eine Anweisung. Dies kann eine einfache oder eine zusammengesetzte Anweisung sein. Zusammengesetzte Anweisungen sind mit Klammerung in `{}` zu einem Block zusammengefasste beliebige Anweisungen.
- Die Verwendung von Flussdiagrammen mit Zusicherungen bietet sich an, um komplexe Bedingungen zu analysieren; vom Flussdiagramm aus ist die Codegenerierung dann nicht mehr schwer. Vergleiche im Skript [1, S. 37, Abb. 7]. Zusicherungen helfen, bei geschachtelten Bedingungen die nächste notwendige Bedingung zu finden, weil sie den Status von Variablen im Programmfluss angeben.

2.5 Die Switch-Anweisung

- Bei der switch-Anweisung dürfen einer Alternative mehrere zugehörige Anweisungen folgen, d.h. es ist keine Klammerung zu einer zusammengesetzten Anweisung erforderlich.
- Man betrachte folgendes Codefragment:

```
switch (s.at(i)) {
   case '.' :
   case ',' : vK = false; break;
   case '0' : case '1' : case '2' : // do something; break;
} // kein Semikolon am Ende von switch
```

  Die Marken von `'.'` und `','` sind ebenso wie die der Ziffern Mehrfachmarken; das Layout erweckt vielleicht irrtümlich den Eindruck, nach case `'.' :` würde nichts ausgeführt; dem ist nicht so.

2.6 Übungen

3 Schleifen

3.1 Die While Schleife

3.2 N Zahlen aufaddieren

- Dieses Kapitel beschreibt, wie man Schleifen unter Verwendung mathematischer Zahlenfolgen korrekt modellieren kann.
- Wertverlaufstabellen enthalten in einer Zeile die an einer definierten Stelle im Programm (hier unmittelbar vor Prüfung der Schleifenbedingung) gleichzeitig »gemessenen« Werte von Variablen.

3.3 Die For Schleife

Es ist möglich, Variablen an jeder Stelle des Programms zu deklarieren / zu definieren:
`for (int i = 0; i < 5; ++i) cout << "Hallo Nr " << i << endl;`

Dies ist allerdings die Definition einer lokalen Variablen, die nur in der for-Schleife ansprechbar ist.

Der Ausdruck im Kopf der `for`-Schleife (hier `++i;`) wird stets erst am Ende der Schleifendurchlaufs ausgeführt! Deshalb hat die Laufvariable bis unmittelbar vor Ende des Schleifendurchlaufs ihren Initialisierungswert. Siehe Skript [1, Kapitel 3.3 »Die For-Schleife«, S. 47 (S. 52 in PDF)]:

Die Schleife

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

entspricht darum exakt

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

Bei `for`-Schleifen ist es ein guter Tipp zum Finden einer einfachen Schleifenbedingung, sich zuerst zu überlegen, welchen Wert die Zählvariable im Abbruchfall hat und dann diesen Fall in der Schleifenbedingung auszuschließen.

3.4 Die Do-While Schleife

Syntax der `do while` - Schleife:

```
do
   // <anweisung>
while (<bedingung>);
// dieses letzte Semikolon ist syntaktisch erforderlich!
```

3.5 Schleifenkonstruktion: Zahlenfolge berechnen und aufaddieren

Es empfiehlt sich, Schleifen mit System aus einer Wertverlaufstabelle, die den Aufbau der in der Schleife vorkommenden Zahlenfolgen enthält, zu konstruieren. Dabei empfiehlt sich folgendes Vorgehen:

1. Für die Schleife relevante Variablen festlegen
2. Schleifenkörper konstruieren: wie ergeben sich die neuen Werte aus den Werten des vorherigen Schleifendurchlaufs? Dies kann entnommen werden aus einer Wertverlaufstabelle bzw. aus einer gefundenen Schleifeninvariante.
3. Die richtigen Initialisierungswerte der relevanten Variablen festlegen. Entspricht der ersten Zeile einer Wertverlaufstabelle bzw. einer Vorbedingung, so dass die Schleifeninvariante zu Schleifenbeginn gilt.

4. Die Schleifenbedingung formulieren, d.h. die Abbruchbedingung. Aus der Verneinung der Schleifenbedingung und der Schleifeninvariante muss die Nachbedinung (das gewünschte Ergebnis) gefolgert werden können.

3.6 Rekurrenzformeln berechnen

Kann man ein Problem in einer Rekurrenzformel (rekursive Formel) formulieren, so fällt die Konstruktion der Schleife daraus meist nicht mehr schwer.

Es ist sinnvoll, eine Wertverlaufstabelle so zu schreiben, dass eine Zeile stets de Variablenbelegung vor dem Test der Schleifenbedingung wiedergibt. Bei kopfgesteuerten Schleifen ist die erste Zeile nämlich dann immer die Zeile mit den Initialwerten der relevanten Variablen und man kann sich den Zeitpunkt der Abbruchbedingung nach der letzten Zeile legen, d.h. die danach ausgewertete Bedingung so formulieren, dass sie nicht mehr zutrifft.

Die Initialwerte der relevanten Variablen können so gewählt werden, dass der erste Iterationsvorgang (bei dem ggf. Spezialwerte auftreten, wie $x^0 = 1$) bereits vor dem ersten Schleifendurchlauf als abgeschlossen betrachtet werden kann. Damit fällt es einfach, in Abhänigkeit eines Durchlaufzählers (sagen wir, mit dem Startwert $n_0 = 0$) zu sagen, der wievielte Schleifendurchgang schon abgeschlossen wurde: immer der sovielte, wie der Durchlaufzähler am Ende des Durchgangs angibt. In dieses System passt, dass bei einem Initialisierungswert von $n_0 = 0$ eben der 0te Schleifendurchgang schon vor Beginn der Schleife abgeschlossen wurde.

Man betrachte also zum besseren Veständnis die Initialwerte als die Werte, die sich nach dem 0ten Schleifendurchlauf ergeben haben, das sind die ersten Werte der durch die Schleife zu erstellenden (Zahlen-)Folgen. Für die Zahlenfolge $x_n = 1 + 2 + 3 + \ldots + n$ würde also der Initialwert 1 angenommen.

3.7 Berechnung von e

3.8 Die Schleifeninvariante

Die Invariante ist eine Beziehung zwischen den relevanten Variablen einer Schleife, die unabhängig davon, wie sie durch vorherige Schleifendurchläufe verändert wurden, einen konstanten Wert hat (»invariant ist«). Sie gilt zu Beginn und am Ende eines Schleifendurchlaufs, aber natürlich nicht, während die Variablen gerade bearbeitet werden.

Das Ziel der Schleife ist es nun, die Werte so lange unter Einhaltung der Invariante zu verändern, bis ein Wert zum Zielwert verändert wurde.

Gibt es für
de Schleife
Invariante?

3.9 Geschachtelte Schleifen und schrittweise Verfeinerung

Die Top-Down-Methode: Programme werden hierachisch abgearbeitet, d.h. ein Programm oder Unterprogramm ruft wieder Unterprogramme auf, die bestimmte Teilaufgaben erfüllen und dann zum Hauptprogramm zurückkehren. Unterprogramm meint in diesem Zusammenhang auch Kontrollstrukturen wie if und while. Beim Programmentwurf nach Top-Down-Methode sollte man also jedem Unterprogramm eine Aufgabe zuweisen und ihm dafür wieder Unterprogramme mit Teilaufgaben als Hilfe zur Verfügung stellen.

Beim Top-Down-Entwurf verfeinert man schrittweise, d.h. man muss bei der Definition einer Teilaufgabe noch nicht ihre Lösung kennen, sondern behandelt sie als »black box«.

3.10 Übungen

4 Einfache Datentypen

4.1 Was ist ein Datentyp

4.2 Datentypen im Programm

Definition Jede Definition ist auch eine Deklaration. int f; ist eine Variablendefinition. Die Aufgaben einer Variablendefinition sind:

- den nötigen Speicherplatz für die Variable schaffen (das tut eine Deklaration nicht!)
- Auskunft darüber geben, wie die Variable zu interpretieren ist

Allgemeiner: Eine Definition (sei es eine Typ-, Funktions-, Methoden- oder Variablendefinition) beschreibt ein neu eingeführtes Konstrukt.

Deklaration Eine Variablendeklaration gibt an, dass eine Variable in einem Programm existiert und an einer anderen Stelle definiert ist. Bei einer Deklaration werden Name und Typ der Variablen angegeben.

```
extern double pi;
```

Allgemeiner: Eine Deklaration (sei es eine Typ-, Funktions-, Methoden- oder Variablendeklaration) informiert den Compiler über wesentliche Eigenschaften eines neuen Konstruktes, ohne es damit unbedingt vollständig zu beschreiben.

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 ¨ uber den Typ der Funktion. Diese Information benutzt der Compiler um Maschinencode f¨ ur die Aufrufstellen der Funktion zu er-zeugen. Dabei wird auch die Korrektheit der Verwendung gepr¨ uft (richtige Typen und Anzahl der Argumente, korrekter Typ des Ergebnisses, etc.). Allgemein: Eine Deklaration f ¨ uhrt einen Namen mit einem bestimmten Typ in das Programm ein. Definition: Eine Definition ist eine Information des Programms an den Compiler ¨ uber die gesamte Funktion (Typ + Anweisungen). Diese Information benutzt der Compiler um den Maschinen Code f¨ ur die Funktion selbst zu erzeugen. Allgemein: Eine Definition ordnet einem neu ins Programm eingef¨ uhrten Ding einen Namen zu. Eine Deklaration ist also eine unvollst¨ andige Definition, bei der nur der Typ festgelegt wird. Eine Definition legt neben dem Typ auch alle anderen Eigenschaften fest. Meist ist eine Definition auch gleichzeitig eine Deklaration, z.B. ist jede Variablendefinition auch eine Deklaration: `int a[10];`

Literale Stücke des Programmtextes, die in allen Programmen das gleiche bedeuten. Sie bezeichnen Werte von Variablen.

Namen Stücke von Programmtext, deren Bedeutung im Programm festgelegt wird.

4.3 Integrale Datentypen

Der Datentyp signed short Das höchstwertige Bit ist für das Vorzeichen reserviert (Wert 1: −; Wert 0: +).

- 0 wird dargestellt als

  ```
  0000 0000 0000 0000
  ```

- 32767 ist die größte darstellbare positive Zahl

  ```
  0111 1111 1111 1111
  ```

- Eine weitere Addition von −1 führt zu einem negativen Vorzeichenbit, d.i. zum Übergang in den negativen Zahlenbereich. Deshalb ist die betragsgrößte darstellbare negative Zahl −32768:

  ```
  1000 0000 0000 0000
  ```

  Dabei wird eine negative Zahl durch 1 als Vorzeichenbit und anders als die positiven Zahlen durch das Zweierkomplement ihrer Binärzahl (d.h. durch das invertierte Bitmuster) dargestellt. Die Darstellung der negativen Zahlen durch ihr Zweierkomplement hat den Vorteil, dass die Addition positiver Zahlen genauso geschehen kann wie bei zwei positiven Zahlen, unter Vernachlässigung der besonderen Bedeutung des Vorzeichenbits (vgl. die Addition der Bitmuster zu $-1 + 1 = 0$). Deshalb bedeuten hier 15 Bit mit logisch 0 betragsmäßig (fast) dasselbe wie 15 Bit mit logisch 1 bei positiven Zahlen. Fast, weil es einen weiteren kleinen Unterschied gibt:

- −1 ist

  ```
  1111 1111 1111 1111
  ```

Dies ist nicht die Darstellung von 0 im Zweierkomplement, weil die Darstellung von 0 sonst durch ± 0 redundant wäre. Also ist die größte darstellbare negative Zahl um 1 betragsgrößer als die größte darstellbare positive Zahl.

Die einfache binäre Zahlenreihe `0000 0000 0000 0000` bis `1111 1111 1111 1111` schließt sich so zu einem Zahlenkreis, der sich abhängig vom führenden Bit in eine negative und eine positive Hälfte teilt. Die beiden Hälften treffen bei $+0$ und -1 und bei $+32767$ und -32768 aufeinander. unsigned-Datentypen bilden einen ebensolchen Zahlenkreis, jedoch ohne die Unterteilung in positive und negative Hälfte. Die größte darstellbare Zahl im unsigned short entspricht deshalb der -1 im signed short, was auch bei einer entsprechende Zuweisung zwischen `short i=-1; unsigned short j=i;` beobachtet werden kann; unsigned short enthält also einfache Binärzahlen.

Der Datentyp `char` Literale für `char` sind die Zeichen der Tastatur und bestimmte Escape-Sequenzen für nicht-druckbare Zeichen (`\n`, `\b`, `\v` usw.). Daneben hat aber jeder mögliche Wert von einer `char`-Variablen auch ein hexadezimales und oktales Literal. Hexadezimale `char`-Literale beginnen mit `\x` oder `\X` und es folgt eine zweistellige hexadezimale Zahl (Ziffern $0-9$, $A-F$). Oktale `char`-Literale beginnen mit `\` und es folgt eine dreistellige aktale Zahl (Ziffern $0-7$).

4.4 Bitoperatoren

4.5 Der Datentyp Float

4.6 Konversionen

4.7 Zeichenketten und Zahlen

In C++ sind zwei grundverschiedene Typen für Zeichenketten zu unterscheiden: C-Strings (`char`-Felder) und C++-Strings (Klasse `string`). Alle Literale für Zeichenketten (d.h. Text in doppelten Anführungszeichen) sind C-Strings. Deshalb benötigt die folgende Anweisung nur die Inklusionsdirektive `#include<iostream>`, jedoch kein `#include<string>`, denn es wird kein Datentyp `string` verwendet:

```
cout << "Bitte gruesse mich: " << endl;
```

Verkettung von `C++`-Strings geschieht durch den `+`-Operator:

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

4.8 Aufzählungstypen: enum

Aufzählungstypen sind das erste Beispiel einer Typdefinition; dabei wird die Typdefinition einem Typbezeichner zugewiesen; dieser ist noch keine Variable, die im Programm angesprochen werden könnte, sondern dienst als Typbezeichner für Variablendeklarationen, wie in folgendem Codefragment:

```
enum Farbe {rot, gelb, gruen, blau}; // neuer Typ
Farbe klecks = gelb; // Variablendefinition
                     // mit Initialisierung;
```

4.9 Namen für Typen: typedef

4.10 Übungen

5 Felder und Verbunde

5.1 Felder sind strukturierte Variablen

Vorsicht: Weder der Compiler noch das Programm zur Laufzeit merken, wenn mit zu großen Indizes auf Werte außerhalb der Grenzen des Feldes zugegriffen wird (C++ ist ja kein ADA). Liest man hier versehentlich Werte aus, so wird einfach das Bitmuster der diesem Index eigentlich entsprechenden Speicherstelle als vom Typ der Feldkomponenten interpretiert und ausgegeben.

5.2 Indizierte Ausdrücke, L und R Werte

5.3 Suche in einem Feld, Feldinitialisierung, Programmtest

Die Initialisierung von strukturierten Datentypen (Felder, Strukturen) geschieht stets in geschweiften Klammern. Dies kann man sich analog dazu merken, dass eine strukturierte Anweisung (d.i. eine zusammengesetze Anweisung) auch in geschweifte Klammern gefasst wird.

5.4 Sortieren, Schleifeninvariante

5.5 Zweidimensionale Strukturen

Zugriff auf zweidimensionale Arrays: üblicherweise werden zweidimensionale Arrays (z.B. Matrizen) in der Zeilenform dargestellt, d.h. als Feld von Zeilenfeldern:

```
{{a11,a12,a13},
 {a21,a22,a23},
 {a31,a32,a33}}
```

Der Zugriff auf ein Element `ayx` erfolgt entsprechend der hierarchischen Strukturierung des Feldes

```
Feld
   → Zeilenfeld1
      → Element1
```

in der Hierarchie absteigend mit a[zeilenfeldnummer][elementnummer], also vereinfacht mit a[zeilennummer][spaltennummer], in mathematischen Koordinaten also a[y][x] (also mit vertauschten Koordinaten gegenüber mathematischer Notation $a\,(x|y)$).

Beispiel Matrixmultiplikation: Die Größe von Matrizen wird immer als Zeilen × Spalten angegeben[1]. Das Ergebnis der Multiplikation einer $M \times K$- und einer $K \times N$-Matrix ist eine $M \times N$-Matrix. Beispiel aus dem Programm, wobei $M = N = 2$:

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \end{pmatrix} \times \begin{pmatrix} 0 & 1 \\ 1 & 2 \\ 2 & 3 \end{pmatrix} = \begin{pmatrix} 8 & 14 \\ 16 & 28 \end{pmatrix}$$

5.6 Beispiel: Pascalsches Dreieck

Beispielprogramm siehe `TestPascalDreieck.cc`

5.7 Beispiel: Gauss Elimination

5.8 Verbunde (`struct` Typen)

Nach der Deklaration eines Verbundes muss stets ein Semikolon folgen, denn die geschweiften Klammern meinen hier ja keine (!) zusammengesetzte Anweisung.

5.9 Übungen

6 Funktionen und Methoden

6.1 Funktionen

Funktionen mit dem Rückgabewert `void` brauchen kein `return;` am Ende der Funktionsdefinition. Dies ist völlig unsinnig, da die Funktion hier ohnehin selbst mit Ende der Definition enden würde.

Es gibt mehrere Arten der Parameterübergabe:

- Wertparameter (call by value): die Änderungen der lokalen Variable in der Funktion haben keine Auswirkung auf die übergebene Variable. Syntax:

[1] Die Funktion im Programm LyX, in dem dieses Dokument geschrieben wurde, verlangt jedoch: `math-matrix <spalten> <zeilen>`

```
function(int);      // nur Deklaration
function(int var);  // nur Deklaration
function(int var) { // Definition
   // ...
}
```

- Referenzparameter (call by reference): die Änderungen der lokalen Variablen in der Funktion entsprechen den an der übergebenen Variablen durchgeführten Änderungen.

```
function(int &);      // nur Deklaration
function(int &var);  // nur Deklaration
function(int &var) { // Definition
   // ...
}
```

- Pointerparameter (call by address).

6.2 Freie Funktionen

6.3 Methoden

6.4 Funktionen und Methoden in einem Programm

6.5 Funktionen und schrittweise Verfeinerung

Im Punkt »Die Funktion `prim`« des Skriptes [1, Kap. 6.5]:

```
bool prim (int n) { // ist n prim?
   // folgende Anweisung ist unnötig, da die for-Schleife
   // keinmal durchlaufen wird fuer n==2, denn 2<2 ist nicht
   // erfuellt
   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;
}
```

6.6 Übungen

7 Programmstatik und Programmdynamik

7.1 Funktionsaufrufe: Funktionsinstanz, Parameter, Rückgabewert

7.2 Sichtbarkeit und Lebensdauer von Parametern und Variablen

7.3 Lokale Variable, Überdeckungsregel

7.4 Globale Variablen und Prozeduren

Seiteneffekte nennt man die Effekte (das sind Änderungen an Variablen), die eine Funktion durchführt, die man aber nicht aus dem Funktionsaufruf erkennen kann. Die einzige Möglichkeit dazu ist die Änderung globaler Variablen durch die Funktion, denn diese allein kann sie außer ihrem eigenen Rückgabewert nach außen hin beeinflussen.

7.5 Funktionen: Sichtbarkeit und Lebensdauer

7.6 Methoden: Sichtbarkeit und Lebensdauer

7.7 Wert und Referenzparameter

7.8 Felder als Parameter

Referenzparameter sind Referenzen auf die aktuellen, beim Aufruf eingesetzten Variablen und werden als solche an die Funktion übergeben (call by reference). Referenzen des Namens `reference` auf eine Variable vom Typ `type` werden geschrieben als:

```
type &reference;
```

Jede Änderung von reference im Programmtext ist damit eigentlich eine Änderung der Variablen, auf die die Referenz zeigt. Referenzen werden mit einem Namen angesprochen, deshalb sind Deklarationen wie

```
void f (int &x[10])
// aufgrund Operatorenrangfolge == void f (int & (x[10])
```

unzulässig, denn das würde ein Feld mit Referenzen als Komponenten definieren. Um ein Feld als Referenzparameter zu übergeben, ist deshalb zu schreiben:

```
void f (int (&x)[10])
```

Das garantiert, dass die Referenz als `x` angesprochen wird und verweist auf eine Variable vom anonymen Typ »Feld von 10 Integers«. Der Zugriff auf dieses Feld erfolgt mit `x[i]`.

7.9 Namensräume

`using`-Deklarationen

- Sie werden äquivalent zu Deklarationen an dieser Stelle im Programmtext behandelt.
- Deshalb kann eine `using`-Deklaration in einer Funktion genauso wie eine lokale Deklaration eine globale Variable abdecken. Der Compiler findet diese Deklaration bei der Suche »von innen nach außen« zuerst. Beispiel: Im Sourcecode zu dieser Aufgabe `Aufg.BimPam.cc` in Funktion `f()`: die Deklaration `using Bim::b;` überdeckt die Sichtbarkeit der globalen Variablendeklaration `int b = 3`.
- Deshalb kann eine `using`-Deklaration aber auch in Konflikt kommen mit der Deklaration einer lokalen Variable, wenn sie in der Funktion gemacht wird.
  Beispiel: Skript [1, S. 152] Funktion `f1()`: Kollision von lokalem `b` mit `NS::b`. Der Sourcecode ist auch enthalten in `TestNamespace.cc`.

`using`-Direktiven

- Sie werden äquivalent zu dem Stück des Programmtextes mit den entsprechenden Definitionen an seiner ursprünglichen Stelle, jedoch nach Entfernung der `namespace <name> { ... }`-Klammerung behandelt.
- Deshalb kann durch eine `using`-Direktive nie eine andere Variable abgedeckt werden; wird im `namespace` eine globale Variable definiert, so gerät sie nun in Konflikt mit namensgleichen globalen Variablen (nur wenn sie verwendet wird!) statt diese zu überdecken.
  Beispiel: Skript [1, S. 153] Funktion `f2()`: Es besteht eine Zweideutigkeit, ob `::c` oder `NS::c` nach `using namespace NS;` gemeint ist. Der Sourcecode ist auch enthalten in `TestNamespace.cc`.
- Deshalb kann aber auch eine using-Direktive nie in Konflikt kommen mit Deklarationen lokaler Variabler in Funktionen, in denen die Direktive gilt. Denn: diese lokalen Variablen decken bei Namensgleichheit die als global definiert betrachteten Variablen des `namespace` ab.
  Beispiel: Skript [1, S. 153] Funktion `f2()`: `NS::b` ist durch lokales `b` abgedeckt, es gibt keine Kollision. Der Sourcecode ist auch enthalten in `TestNamespace.cc`.

7.10 Übungen

8 Techniken und Anwendungen

8.1 Rekursion

8.2 Rekursion als Kontrollstruktur, Rekursion und Iteration

»Rekursion ist also die mächtigste Kontrollstruktur. Mit ihr lassen sich alle Probleme lösen, die sich mit Schleifen lösen lassen und noch einige mehr.« [1, S. 157; entspr. S. 163 in PDF].

8.3 Rekursion als Programmiertechnik, rekursive Daten

Verfahren der Entwicklung eines rekursiven Algorithmus auf rekursiven Daten:

1. Die rekursive Struktur der Daten erkennen: einen Ansatz zur rekursiven Einteilung der Daten in einfachere Komponenten gleicher Art machen, die einfachste (»elementare«) Komponente festlegen. So gibt es z.B. mehrere Möglichkeiten, wenn eine Zeichenkette in eine Zahl umgewandelt werden soll, von denen nicht alle zu einem umsetzbaren Algorithmus führen müssen. Eine Zeichenkette ist:
   (a) entweder ein Zeichen oder ein Zeichen, dem eine Zeichenkette folgt
   (b) entweder leer oder eine Zeichenkette, der ein Zeichen folgt
2. Nun wird ein Algorithmus entworfen, der zu der gefundenen rekursiven Zerlegung passt.

8.4 Rekursive Daten: logische kontra physische Struktur der Daten

8.5 Vor und Nachbedingungen

8.6 Funktionen als Parameter

8.7 Typen mit Ein- /Ausgabe-Methoden

Die sogenannte Determinante (wörtl. die »Bestimmende«) von zwei Vektoren ist $D = \left|\vec{a}\vec{b}\right|$. Ihr Wert bestimmt, ob die beiden Vektoren kollinear sind ($\vec{a} = x\vec{b} \Rightarrow D = 0$) oder nicht ($\vec{a} \parallel \vec{b} \Rightarrow D \neq 0$). Warum ist das so? Wenn zwei Vektoren kollinear sind, so heißt das ausgeschrieben: es gibt ein x mit:

$$\begin{aligned}
\vec{a} &= x\vec{b} \\
\Leftrightarrow \begin{pmatrix} a_x \\ a_y \end{pmatrix} &= x \cdot \begin{pmatrix} b_x \\ b_y \end{pmatrix} \\
\Leftrightarrow a_x &= x \cdot b_x \\
\wedge a_y &= x \cdot b_y \\
\Leftrightarrow \frac{a_x}{b_x} = &\ x \ = \frac{a_y}{b_y} \\
\Leftrightarrow a_x \cdot b_y &= a_y \cdot b_x \\
\Leftrightarrow D = a_x \cdot b_y - a_y \cdot b_x &= 0
\end{aligned}$$

Der Schritt »Wir setzen voraus, dass die Geraden nicht parallel sind und lösen das Gleichungssystem nach s auf« ausführlich: Da wir hier ebene Vektoranalysis betreiben, schneiden sich die Geraden stets, wenn sie nicht parallel sind, d.h. es das Gleichungssystem hat unter dieser Voraussetzung stets eine eindeutige Lösung für s. Bestimmung:

$$sa_{1x} - ta_{2x} = p_{2x} - p_{1x} \quad (1)$$
$$sa_{1y} - ta_{2y} = p_{2y} - p_{1y} \quad (2)$$

In Gleichung 2 kann die Unbekannte t eliminiert werden, indem sie mit $\frac{a_{2x}}{a_{2y}}$ erweitert wird und durch die Differenz »Gleichung 1 - Gleichung 2« ersetzt wird. Das LGS wird damit zu:

$$
\begin{aligned}
sa_{1x} - ta_{2x} &= p_{2x} - p_{1x} && (3)\\
s \cdot \left(a_{1x} - a_{1y}\frac{a_{2x}}{a_{2y}}\right) &= (p_{2x} - p_{1x}) - (p_{2y} - p_{1y})\frac{a_{2x}}{a_{2y}} && (4)\\
\Leftrightarrow s \cdot \left(\frac{a_{1x}a_{2y} - a_{1y}a_{2x}}{a_{2y}}\right) &= p_{2x} - p_{1x} - \frac{a_{2x}\,(p_{2y} - p_{1y})}{a_{2y}} && (5)\\
\Leftrightarrow s &= \frac{a_{2y}\,(p_{2x} - p_{1x}) - a_{2x}\,(p_{2y} - p_{1y})}{a_{2y}} \cdot \frac{a_{2y}}{a_{1x}a_{2y} - a_{1y}a_{2x}} && (6)\\
&= \frac{-1\,(a_{2y}\,(p_{2x} - p_{1x}) - a_{2x}\,(p_{2y} - p_{1y}))}{-1\,(a_{1x}a_{2y} - a_{1y}a_{2x})} && (7)\\
&= \frac{D\,(\vec{p}_2 - \vec{p}_1, -\vec{a}_2)}{D\,(\vec{a}_1, -\vec{a}_2)} && (8)
\end{aligned}
$$

Die Erweiterung des Bruches mit -1 in Gleichung 7 ist eigentlich unnötig, wurde aber durchgeführt, um das im Skript angegebene Ergebnis $s = \frac{D(\vec{p}_2 - \vec{p}_1, -\vec{a}_2)}{D(\vec{a}_1, -\vec{a}_2)}$ zu erhalten statt dem ebenfalls richtigen $s = \frac{D(\vec{p}_2 - \vec{p}_1, \vec{a}_2)}{D(\vec{a}_1, \vec{a}_2)}$.

8.8 Überladung von Operatoren

8.9 Übungen

A Standardbibliothek

A.1 Zeichenketten (Strings)

A.2 Ausgabeformate

Ein horizontaler Tabulator (`cout << '\t'`) ist in C++ (mindestens unter Linux) 8 Zeichen lang.

B Lösungshinweise

B.1 Lösungen zu Kapitel 1

B.1.1 Aufgabe 1

1. Syntaktische Fehler (bedeuten die fehlerhafte - vom Sprachstandard abweichende - Notation eines Programms. Es sind all die Fehler, die vom Compiler erkannt werden, zum Beispiel:

   ```
   int main
      [
       const float pi=3.192;
      ]
   ```

   Statt der Verwendung der geschweiften Klammern `{}`. Semantische Fehler sind korrekt formulierte (vom Compiler akzeptierte) falsche (nicht dem Programmzweck entsprechende) Anweisungen. Hier sollte z.B. die Kreiszahl π definiert werden, es wurde jedoch ein falscher Wert verwendet.

2. Siehe `Aufg.SummeProdukt.cc`.

3. Eine mögliche Folge der Zuweisungen:

   ```
   f2=2.0*f1;
   f3=2.0*f2;
   ```

4. Nach den einzelnen Fällen gegliedert:

   (a) Ja, gleiche Wirkung.

(b) Nein, nicht die gleiche Wirkung, denn bei den einzelnen Anweisungen entsteht aufgrund `a=a-a` stets 0 als Ergebnis. Der aktuelle Wert von `a`, der in der Zuweisung `a=(b-a)*2;` verwendet wird, wurde durch `a=b;` überschrieben und steht in den einzelnen Anweisungen daher nicht mehr zur Verfügung.

(c) Ja, gleiche Wirkung aufgrund des Distributivgesetzes und der Regel »Punkt- vor Strichrechnung«.

(d) Nicht die gleiche Wirkung. Es entsteht zwar für `a` jeweils dasselbe Ergebnis, `b` hat jedoch im zweiten Fall am Ende einen anderen Wert als im ersten Fall, d.h. einen gegenüber dem Originalwert veränderten Wert. Dies ist auch eine Wirkung der Anweisungen, und sie unterscheidet sich zwischen beiden Möglichkeiten.

B.1.2 Aufgabe 2

Siehe `Aufg.StringOp.cc`

B.1.3 Aufgabe 3

1. Der Compiler findet syntaktische Fehler, der Programmierer muss die semantischen Fehler finden (Fehler, die dazu führen, dass ein korrekt kompiliertes Programm nicht tut, was der Programmierer will).

2. Bedeutungen, Täter und Zielsetzungen:

   Analysieren. Die Untersuchung einer Problemstellung, die durch ein Programm gelöst werden soll, mit dem Ziel, herauszufinden, was man überhaupt tun soll und dies in einem Entwurfsdokument aufzuschreiben. Durchgeführt von Softwareentwicklern [nicht von Programmierern!].

   Übersetzen. Überführen von einer Sprache L1 in einer Sprache L2. Meist meint man damit das compilieren (Übersetzen eines Quellprogramms aus der Programmiersprache in ein Maschinenprogramm in Maschinensprache), durchgeführt vom Compiler. Ziel ist, dass ein Quellprogramm am Ende als ausführbares Maschinenprogramm vorliegt.

   Entwerfen. Die Überlegung, wie eine analysierte Problemstellung in ein Programm umgesetzt werden kann. Durchgeführt von Softwareentwicklern [nicht von Programmierern!]. Ziel ist das Entwurfsdokument.

   Editieren. Tätigkeit des Veränderns einer Datei (z.B. ein Quellprogramm). Vom Programmierer durchgeführte Tätigkeit mit dem Ziel, das Quellprogramm entsprechend seinen Vorstellungen zu ändern oder zu erstellen.

3. Vergleich der Bedeutung von »Variable« und »Konstante«

   - Variable

     Mathematik: bezeichnet einen veränderlichen (»variablen«) Wert.

     Physik: bezeichnet einen Parameter.

     Programmierung: bezeichnet einen Behälter für wechselnde Werte eines bestimmten Typs, oder genauer je nach Kontext den Behälter (lvalue) oder den Wert selbst (rvalue).

   - Konstante

     Mathematik: bezeichnet einen festen Wert

     Physik: bezeichnet eine in der Natur beobachtete unveränderliche Größe.

     Programmierung: bezeichnet eine unveränderliche Variable im Sinne der Programmierung. Sie besteht auch aus Behälter (Speicherzelle) und Wert (Inhalt der Speicherzelle), aber der Wert kann nicht verändert werden.

4. Bedeutung von $x = 2 \cdot x$

   (a) in der Mathematik: eine Gleichung (Bedingung), die die Variable x erfüllen muss, so dass sich eine Einschränkung ergibt, welche Werte x annehmen kann. Hier gilt äquivalent $x = 0$.

   (b) in der Programmierung: weise der Variablen x einen neuen Wert zu, nämlich den, der dem Doppelten ihres bisherigen Wertes entspricht.

5. »Höhere Programmiersprachen« sind problemorientierte Programmiersprachen, d.h. sie orientieren sich an der Denk- und Vorstellungswelt des Menschen statt an der Funktion der Maschine. Eine höherer Programmiersprache abstrahiert mehr von der Hardware als eine nicht so hohe Programmiersprache (wie Assembler).

6. Compiler übersetzen Quellprogramme (in einer (höheren) Programmiersprache geschrieben) in Maschinenprogramme (in Maschinencode geschrieben). Die zu übersetzenden Dateien werden per Kommandozeilenparameter übergeben, die ausführbare Datei schreibt der Compiler in eine Datei, die standardmäßig `a.out` heißt (bei `g++`), deren Namen aber auch mit der Option `-o <dateiname>` angegeben werden kann.

7. C++ ist eine Sprache für eine virtuelle Maschine. Sie kann nicht auf einer real existierenden Maschine ohne vorherige Übersetzung ausgeführt werden, ist also keine Maschinensprache im eigentlichen Sinn.

8. Die Zuweisung `x=1;` bzw. `y=1;` ist im strikten Sinne falsch, denn hier wird einem `float`-Wert ein Integer zugewiesen. Die internen Darstellungen beider Datentypen sind aber inkompatibel, so dass der Compiler automatisch eine Typkonversion durchführt und `x` schließlich den `float`-Wert `1.0` besitzt (wie nach `x=1.0;`). Die Zuweisung `x=x+y;` ist dann wieder ohne Typkonversion möglich, weil sowohl `x` als `y` vom Typ `float` sind. Dasselbe gilt für `a=1;`, `b=2;`, `a=a+b;`.

9. Der Compiler muss wissen, wieviel Byte eine `float`-Zahl belegt, um die Adressen dieser Variablen richtig verwalten zu können und in Zuweisungen die vollständige `float`-Zahl auszuwerten. Der Prozessor und die Software für Ein- und Ausgabe müssen es ebenso wissen.
   Die Länge von `floats` kann zwar jeder selbst einstellen (im Compiler ist sie als Vielfaches der word-Länge definiert), was jedoch aus Kompatibilitätsgründen nicht ratsam ist. Sie wird festgelegt durch den Prozessorhersteller (oft Intel) durch die Definition der arithmetischen Operationen, die der Prozessortyp ausführen kann, und durch Compilerbauer (oft Microsoft), die weitere Datentypen emulieren können.

B.2 Lösungen zu Kapitel 2

B.2.1 Aufgabe 1

1. Wert von `max`:

   i `max==4` nach der Zuweisung `max=b;`

   ii `max==4` nach der Zuweisung `max=a;`

   iii `max==3` nach der Zuweisung `max=b;`

2.

   statisch Die Entscheidung wird während der Erstellung des Programms getroffen.

   dynamisch Die Entscheidung wird während der Laufzeit des Programms getroffen.

3. Der Programmverlauf ist bei Verzweigungen deshalb »dynamisch«, weil er nicht nur vom Quelltext, sondern auch vom booleschen Wert der Bedingungen abhängt, der wiederum z.B. von Eingaben des Benutzers, Messgrößen oder (Pseudo-)Zufallswerten abhängen kann. Diese Größen wurden im Quelltext nicht festgelegt, es sind dynamische Einflüsse.

4. »Zustand eines Programms« meint die aktuelle Belegung seiner Variablen. Sie wird durch Zuweisungen verändert.

5. Nur die Anweisungsfolge (ii) funktioniert wie gewünscht; in Anweisungsfolge (i) ist der Maximalwert für a=b nicht definiert, in Anweisungsfolge (iii) wird stets b als Maximalwert verwendet.

6. Siehe `Aufg.MaxAus4.HilfsvarN.cc`

7. Siehe `Aufg.MaxAus4.Hilfsvar2.cc`

8. Siehe `Aufg.MaxAus4.Hilfsvar1.cc`

9. Siehe `Aufg.MaxAus4.Hilfsvar0.cc`

B.2.2 Aufgabe 2

1. `if (a>b) if (c>d) x = y; s = t; else u = v;`? Dem Schlüsselwort if darf nur eine einzige (einfache oder zusammengesetzte) Anweisung folgen, aber nicht mehrere, wie hier.
   `if (a>b) if (c>d) {x = y; else u = v;}` ? else kann nicht ohne vorheriges if in einem Block stehen; dies ist syntaktisch unzulässig. Stattdessen war wohl gemeint: `if (a>b) if (c>d) {x = y;} else u = v;`
2. `if (<bedingung>) <anweisung1>; else <anweisung2>;`
   ist äquivalent mit
   `if (<bedingung>) <anweisung1>; if (!(<bedingung>)) <anweisung2>;`
3. Vereinfachungen
   (a) `b = (x == y); if ( b == true ) x = x + y; else x = y + x;`
   ist äquivalent mit
   `if (x==y) x=x+y; else x=y+x;`
   ist äquivalent mit
   `x=x+y;`
   (b) `if (x >= y) x = x; else x = 2*x;`
   ist äquivalent mit
   `if (x<y) x=2*x;`
   (c) `if ((x >= x) == true) x = x; else x = 2*x;`
   ist äquivalent mit
   `<keine Anweisung>`
   (d) `if (((x >= x) == true) == false) x = x; x = 2*x;`
   ist äquivalent mit
   `if (x<x) x=x; x=2*x;`
   ist äquivalent mit
   `x=2*x;`
4. `b=3; a=0;`

B.2.3 Aufgabe 3

1. `a==3; b==6`
2. `a==3; b==6`
3. `a==3; b==4`
4. `a==3; b==6`
5. `a==3; b==6`
6. `a==3; b==6`
7. `a==3; b==4`
8. `a==3; b==6`
9. `a==3; b==4`
10. `a==6; b==4`
11. `a==6; b==4`
12. `a==6; b==4`

B.2.4 Aufgabe 4

1.

```
a=6;
```

2. Nach

$$\begin{aligned} a_1 = 2b &\Leftrightarrow b = \frac{1}{2}a_1 \\ a_2 &= 2\left(b+1\right) \\ &= 2\left(\frac{1}{2}a_1 + 1\right) \\ &= a_1 + 2 \end{aligned}$$

(a_1: alter Wert von a; a_2: neuer Wert von a) folgt als Anweisung

```
a=a+2;
```

3.

```
a=-1*a;
```

4.

```
c=abs(a)+abs(b);
```

5.

```
c=0.5*(a+b);
```

6.

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

Der zweite Teil ist ebenso notwendig, denn der Wert von `a` ändert sich ja nicht selbständig, sondern muss auch durch die gefragten Anweisungen verändert werden.

7.

```
x=x+a+1; a=a+1:
```

Der zweite Teil ist ebenso notwendig, denn der Wert von `a` ändert sich ja nicht selbständig, sondern muss auch durch die gefragten Anweisungen verändert werden; damit `x` sich in der ersten Anweisung auf den alten Wert von `a` beziehen kann, muss diese Reihenfolge der Anweisungen eingehalten werden. Ansonsten wäre möglich:

```
a=a+1; x=x+a;
```

B.2.5 Aufgabe 5

Siehe `Aufg.AfolgtB.cc`

B.2.6 Aufgabe 6

1.

```
if (c=='A') {
  cout << "A\n";
}
if (c=='B') {
   cout << "B\n";
}
else
   cout << "weder A noch B\n";
```

2.

```
if (c=='A') {
  cout << "A\n";
}
if (c=='B' || c!='A') {
   cout << "B\n";
}
cout << "weder A noch B\n";
```

3.

```
if (c=='A') {
   cout << "A\n";
}
if (c=='B') {
   cout << "B\n";
}
```

B.2.7 Aufgabe 7

1.

```
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.

- Die Anweisungen hinter der Marke default müssen mit break; enden, sonst werden alle folgenden Anweisungen im switch-statement auch ausgeführt. Ebenso müssen die Anweisungen hinter den Marken für kleine Vokale mit break; enden, die für große Vokale sollten damit enden, müssen aber nicht.
- Anführungszeichen: es muss "kleiner Vokal" und "großer Vokal" heißen, statt "kleiner Vokal' ' und "großer Vokal' '.
- Die Marken `"o"` und `"O"` müssen mit aufgeführt werden.
- Bei `cout` muss in allen Fällen der Operator `<<` verwendet werden.
- Das letzte `case`-Schlüsselwort ist unzulässig, denn es werden keine Markennamen angegeben.

B.2.8 Aufgabe 8

Siehe `Aufg.MinMaxAus4.cc`

B.3 Lösungen zu Kapitel 3

B.3.1 Aufgabe 1

Lsg. noch n
im Progra
geprüft

1. `a==-1; b==0`
2. `a==3; b==5;`
3. `a==0; b==1`

B.3.2 Aufgabe 2

Lsg. noch n
im Progra
geprüft

Die Schleife endet (d.h., wird das letzte mal durchlaufen), wenn vor diesem Durchlauf die Variablenbelegung folgende Bedingungen erfüllt:

1. innere Schleife terminiert für $a \leq 5$
   äußere Schleife terminiert für $a \geq 5$
2. innere Schleife terminiert nie
   äußere Schleife terminiert nie, denn: Voraussetzung für eine Terminierung ist, dass die innere, nicht terminierende Schleife, nicht betreten wird (also $a \geq 6$ bei Schleifeneintritt) und dass die am Ende geprüfte Bedingung nicht erfüllt ist, so dass die Schleife nicht noch einmal betreten wird (also $a < 5$). $(a \geq 6) \wedge (a < 5) = 0$, d.h. es können nie beide Bedingungen gleichzeitig zutreffen. Es gibt also keinen Fall, für den die Schleife ein letztes Mal durchlaufen wird; da sie jedoch immer ein erstes Mal durchlaufen wird, terminiert sie nie.

B.3.3 Aufgabe 3

Siehe `Aufg.SummeUndMittelwert.cc`

B.3.4 Aufgabe 4

Siehe `Aufg.Summe2i.cc`

B.3.5 Aufgabe 5

Siehe `Aufg.NaeherungPi.cc`
Siehe `Aufg.NaeherungEX.cc`

B.3.6 Aufgabe 6

```
// ...
int a = a_1;
int i = 1;
s = a_1;
while (i != n) {
   s = s + a_1 + i*c;
   i = i + 1;
}
// ...
```

B.3.7 Aufgabe 7

Beweis nach der vollständigen Induktion Behauptung: Die Formel

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1 \tag{9}$$

ist richtig für alle $n \in \mathbb{N}$. Beweis nach der vollständigen Induktion:

1. Sei $M \subseteq \mathbb{N}$ die Teilmenge, für die Gleichung 9 richtig ist. Durch die vollständige Induktion wird nun bewiesen, dass die $M = \mathbb{N}$, dass also die Behauptung richtig ist, dass die Gleichung für alle $n \in \mathbb{N}$ gilt.

2. Ist $1 \in M$? Ja, denn $n = 1$ erfüllt die Gleichung 9:

$$\sum_{i=0}^{1} 2^i = 2^0 + 2^1 = 3 = 2^{1+1} - 1$$

3. Unter der Voraussetzung, dass Gleichung 9 wahr ist für $n \in M$, zeige man, dass sie auch für $n+1$ wahr ist, dass also gilt:

$$\begin{aligned} \sum_{i=0}^{n} 2^i &= 2^{n+1} - 1 \\ \Rightarrow \sum_{i=0}^{n+1} 2^i &= 2^{n+2} - 1 \end{aligned}$$

   Dazu:

$$\begin{aligned} \sum_{i=0}^{n} 2^i &= 2^{n+1} - 1 \\ \Leftrightarrow 2^0 + 2^1 + \ldots + 2^n &= 2^{n+1} - 1 \\ \Leftrightarrow 2^0 + 2^1 + \ldots + 2^n + 2^{n+1} &= 2^{n+1} + 2^{n+1} - 1 \\ \Leftrightarrow \sum_{i=0}^{n+1} 2^i &= 2^1 \cdot 2^{n+1} - 1 \\ &= 2^{n+2} - 1 \end{aligned}$$

   Damit wurde also nach der vollständigen Induktion bewiesen, dass die Formel für jedes $n \in \mathbb{N}$ gilt, dass also $M = \mathbb{N}$, denn: die Formel gilt für $k = n + 1$, wenn sie für n gilt; da sie außerdem für $n = 1$ gilt, folgt: sie gilt auch für $k = 1 + 1 = 2$; da sie also für $n = 2$ gilt, gilt sie auch für $k = 2 + 1 = 3$ usw., d.h. sie gilt tatsächlich für jedes $n \in \mathbb{N}$.

Prüfung mit einem C++-Programm und Vergleich
Siehe `Aufg.Summe2iInduktion.cc`

- Induktionsverankerung und Schleifeninitialisierung: Die Schleifeninitialisierung geschieht mit dem Summand für den Index $i = 0$, nämlich $2^0 = 1$. Diese erste, statisch berechnete Summe $\sum_{i=0}^{0} 2^i = 1 = 2^{0+1}-1$ wird also zur Berechnung der Summe vorausgesetzt. Die Induktionsverankerung wurde hier bei $n = 1$ gewählt; da aber die Gleichung auch für $n = 0$ gilt, hätte sie ebensogut hier gewählt werden können, wobei dann die Gültigkeit für $\mathbb{N}_0$ bewiesen wäre. Dann entspricht die Induktionsverankerung genau der Schleifeninitialisierung.

- Induktionsschritt und Schleifenkörper: Im Schleifenkörper wird jeweils die Summe für eine um 1 größere Summationsgrenze n gebildet und geprüft, ob die Formel für dieses n (im Programm die Variable i) gilt. Dies entspricht dem Verfahren der vollständigen Induktion, denn hier wird auch in einer Schrittweite von $k = n + 1$ geprüft, ob die Formel noch gilt.

B.3.8 Aufgabe 8

Die Tabelle enthält alle ungeraden Zahlen. Jede Zeile enthält eine Spalte mehr als die vorige Zeile. Regelmäßigkeit der Summe einer Zeile: Die Summe der i-ten Zeile ist i^3.
Siehe `Aufg.TabelleUngeradeZahlen.cc`

B.3.9 Aufgabe 9

Siehe `Aufg.WurzelHeron.cc`

B.3.10 Aufgabe 10

Siehe `Aufg.FolgeFx.cc`

B.3.11 Aufgabe 11

Siehe `Aufg.GeradeZahl.cc`
Siehe `Aufg.TeilerProbe.cc`

B.4 Lösungen zu Kapitel 4

B.4.1 Aufgabe 1

1. `int` ist keine Teilmenge von float, da die Darstellungen der Bitmuster inkompatibel sind (eine Verrechnung ist nicht ohne Konversion möglich). Das heißt, das Bitmuster einer Variablen vom Typ int ist nicht automatisch ein Bitmuster desselben Werts einer Variablen vom Typ float, also ist int keine Teilmenge von `float`.

2. Ja. Nur müsste der Compiler für diese Maschinen so gebaut sein, dass er die `float`-Operationen softwaremäßig emuliert, so wie heutige C++-Compiler z.B. den `bool`-Datentyp softwaremäßig emulieren.

3. `'a'` ist ein `int`-Literal und kann daher nicht dem Datentypen string zugewiesen werden; `"s"` ist ein `string`-Literal und kann daher nicht dem Datentypen `char` zugewiesen werden.

4.

   - `10`: Die Zahl 10 als (mathematischer) Wert.
   - `010`: Die Zahl 8 als (mathematischer) Wert. Dies ist ein int-Literal im Oktalsystem, erkennbar an der führenden 0.
   - `0x10`: Die Zahl 16 als (mathematischer Wert. Dies ist ein int-Literal im Hexadezimalsystem.
   - `'1'`: Die Zahl 49 als (mathematischer) Wert. Werte entsprechend der korrekten Interpretation von Bitmustern; da der Datentyp char wie ein Integer interpretiert wird (z.B. bei arithmetischen Operationen), hat er auch Zahlen als Werte.

5. Ein `short` hat zwei Bytes, da die größte mit 2 Bytes darstellbare Binärzahl $2^{16} - 1 = 65535$ ist, das ist genau dem höchsten Wert, den ein unsigned short annehmen kann entsprechend der Ausgabe des Programms.

6. Gemeint ist: erhöhe im letzten Schleifendurchgang am Ende der Schleife durch `++c;` die Variable `c` auf 128, so dass die Schleifenbedingung `c<128` nicht mehr erfüllt ist. Tatsächlich geschieht ein Wertüberlauf: Der Wertebereich einer char-Variablen ist $[-128; 127]$. Wird ++c; auf $c = 127$ angewandt, so ist danach $c = -128$. Möglichkeit zur Behebung: `unsigned char c=32;`.

7. `int`-Werte, denn die Darstellung der ganzen Zahlen in `int`-Werten ist mathematisch exakt. `float`-Werte dagegen werden im Normalfall nur mit endlicher Präzision dargestellt (z.B. π), nur ganze Zahlen und Dezimalbrüche bis zu einer gewissen Zahl Nachkommastellen werden auch hier mathematisch exakt dargestellt.

8. Ein Werteüberlauf ohne Warnung durch Compiler oder Programm. Ein unsigned Integer beliebiger Größe enthält anschließend das Bitmuster für 0 (da die zur korrekten Darstellung nötige führende 1 nicht mehr in den für diesen Integer reservierten Speicher passt), ein signed Integer enthält anschließend das Bitmuster der betragsgrößten darstellbaren negativen Zahl.

9. Tut man dies statisch, so erkennt der Compiler dies und weist die Operation als auf diesem Datentyp nicht definiert mit einer Fehlermeldung zurück. Tut man dies dynamisch, so gibt es einen Programmabsturz; im Prozessor, der diese Operation ausführen musste, wird dann ein Flag-Register für diesen Fehler gesetzt.

noch n
mit ei
Programm
getestet

10. Nicht korrekt. Namen dürfen in C++ nie bloß aus Ziffern bestehen (wie hier die Elemente des Enum), denn solche Zeichenkombinationen werden als `int`-Literale interpretiert.

11.

```
enum farbe {rot, gelb, nachts}; // OK
Farbe gelb; // gelb ist ein schon benutzter Bezeichner
            // was waere sonst gelb=farbe(1+gelb) ?
            // farbe muss klein geschrieben werden;
int rot;    // rot ist ein schon benutzter Bezeichner
```

12. `float a; int 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==0.0;`
    - `a = float(1/2 * j); a==0.0;`
    - `a = float(1/2) * j; a==0.0;`
    - `a = float(1)/2 * j; a==1;`
    - `a = 1/float(2) * j; a==1;`

13. `(true && (30 % 7)) / 5`. Dieser Ausdruck ist in C++ syntaktisch korrekt, da nur integrale Datentypen vorkommen, nur in verschiedener Interpretation als Wahrheits- oder Zahlwerte. Der Wert des Ausdrucks ist 0, denn die abschließende Ganzzahldivision ist, da der Dividend ein Wahrheitswert ist, entweder `0 / 5` oder `1 / 5`.

14.

```
a = int (x);        // Warnung int -> enum
a = x;              // verbotener Typemix von AA und XX
a = x + 1;          // Warnung int -> enum
a = int(x) + 1;     // Warnung int -> enum
a = AA(int(x) + 1); // OK
```

B.4.2 Aufgabe 2

Siehe `Aufg.ASCII-Code.cc`

B.4.3 Aufgabe 3

Siehe `Aufg.Wahrheitstafel.cc`

B.4.4 Aufgabe 4

- (I, WE, RV): `int i=1; float f=i;`
- (E, WE, RE):

```
int i=2;
enum topfteile (deckel, griff, topf);
topfteile pfanne;
pfanne = topfteile(i);
```

B.4.5 Aufgabe 5

1. Der Compiler könnte nicht mehr (wie bisher durch das Verbot eines Typenmix verschiedener enum-Datentypen) mögliche Programmierfehler aufdecken. Wenn bei enum-Typen der Operator `++` eine Variable auf den nächsten enum-Wert setzt (statt sie als Integer um 1 zu erhöhen), so ist das bei Variablen vom Typ Farbe wie hier nicht mehr möglich.

2. Literale sind Zeichenkombinationen, die stets einen festen Wert bezeichnen. Bei enum-Typen definiert der Programmierer jedoch selbst, welche Werte im enum-Typ vorkommen, d.h. welchen Integer-Zahlen die Namen entsprechen. Die Einführung solcher Literale wäre überdies sinnlos, weil die Idee von enum-Typen ja gerade ist, Werte mit (selbst definierten) Namen statt mit (vordefinierten) Literalen zu bezeichnen.

B.4.6 Aufgabe 6

Siehe `Aufg.Bitmuster.cc`

B.4.7 Aufgabe 7

Siehe `Aufg.Zahlsystemwandlung.cc`

B.5 Lösungen zu Kapitel 5

B.5.1 Aufgabe 1

1. Beispiel, bei dem $L = R;$ und $L = R;\ L = R;$ eine unterschiedliche Wirkung haben: $i = i + 1;$.
   Bedingungen für gleiche Wirkung (in den Lösungshinweisen wird die Bedingung für ungleiche Wirkung angegeben, also sollte eher das beantwortet werden): Gleiche Wirkung von $L_1 = R_1;$ und $L_1 = R_1;\ L_2 = R_2;$ besteht dann, wenn die Variablen gleich sind ($L_1 = L_2$) und diese am Ende jeweils den gleichen Wert hat: $R_1 = R_2$. Wann ist nun der R-Wert identischer Ausdrücke gleich? Genau dann, wenn die Ausdrücke nach Einsetzen ihrer jeweiligen Werte durch Termumformungen ineinander übergehen. In obigem Beispiel ist dies nicht der Fall:
   $$i + 1 \neq (i + 1) + 1 = i + 2$$
   Für die Anweisung `i=(2*i)/(2*i)` wäre dies jedoch der Fall (Erweiterung eines Bruches):
   $$\frac{2i}{2i} = 1 = \frac{2\,(2i)}{2\,(2i)}$$
   In folgenden Fällen ändert sich durch die Zuweisung der L-Wert, so dass $L_1 \neq L_2$ und die zweimalige Ausführung der Zuweisung nicht äquivalent ist: `a[i]=++i;` und sogar mit einer konstanten rechten Seite `a[++i]=5;`.

2. Der Wert von `a` ist `a[10] = {1,8,3,3,5,6,7,8,9,0}` nach den Zuweisungen `i=3; a[3]=3; a[1]=a[8]-1;` (d.i. `a[1]=8;`).

3.
   (a) r-Wert
   (b) l-Wert, r-Wert
   (c) r-Wert
   (d) l-Wert, r-Wert
   (e) r-Wert
   (f) l-Wert, r-Wert
   (g) r-Wert

4.
   - auf einen String kann nicht wie auf ein Feld mit einem Index zugegriffen werden, sondern der Zugriff auf ein Zeichen erfolgt über die Methode `<string-variable>.at(<stelle>)`.
   - Strings sind von variabler Länge

- Strings haben als ganzes einen r-Wert (nämlich die Zeichenkette selbst), der r-Wert eines char-Feldes ist jedoch nicht die Reihe der Komponenten, sondern ein Zeiger auf das erste Feld.
- die Zeichen eines `string` sind keine Zeichen `char`, denn `string s='a';` ist unzulässig.
- Der Datentyp string ist im Gegensatz zu einem `char`-Feld kein strukturierter Datentyp, denn er hat keine Variablen als Bestandteile.

5. Nicht mit den üblichen Zuweisungs- oder Vergleichsoperatoren. Es ist die Definition eigener Routinen nötig.

6.

- B ist ein Variablenbezeichner, kein Typbezeichner!
- Die Verwendung des Index 10 auf einem Feld mit 10 Elementen ist unzulässig; erlaubt sind Werte von $0 \ldots 9$.

```
// korrigierte Version:
typedef  float  A[12];
typedef  int    B[11];
int      C[13];
A        a;
B        b;
a[10] = 12;
b[10] = 22;
C[1]  =  1;
```

7. Siehe `Aufg.FeldVergleich.cc`. Die Ausgabe ist natürlich Ungleich, da die Felder unterscheidbare Variablen sind und daher unterschiedliche Zeiger auf das erste Element haben. Der Wert dieser Zeiger wird im Programm durch den Versuch direkten Feldvergleichs ausgewertet; an dessen Ergebnis kann man erkennen, ob es sich um identische Variablen handelt (die auf dieselben Speicherzellen zugreifen!).

8.

```
int main () {
   int f[10],
       i;
   f[0] = 1;
   // i = 0;
   // while (i<9) {
   //    i = i+1;
   //    f[i] = f[i-1]*i;
   // }
   for (i=1; i<10; ++i) {
      f[i]=f[i-1]*i;
   }
}
```

B.5.2 Aufgabe 2

Siehe `Aufg.Fibonacci.cc`

B.5.3 Aufgabe 3

1. Siehe `Aufg.FeldOhneDuplikate.cc`
2. Siehe `Aufg.kGroesstes.cc`

B.5.4 Aufgabe 4

Siehe `Aufg.N_ueber_K.cc`

B.5.5 Aufgabe 5

1. Testprogramm zur Wertebelegung siehe `Aufg.PruefProgramm.cc`
   Wertebelegung nach Programmlauf:

   - `s1a` ist mit seinen Initialisierungswerten belegt
   - `s1b` ist mit zufälligen Werten belegt (es wurde nicht initialisiert)
   - `s2` ist mit zufälligen Werten belegt (es wurde nicht initialisiert)
   - `a`,`b` sind in jeder Komponente mit den Initialisierungswerten von a belegt
   - `sss` hat den Wert ist mit seinem Initialisierungswert `"s_i"` belegt

2.

   - `.sss` ist keine Komponente im Typ `S2` der Variablen `s2;` daher ist `s2.sss=5;` keine zulässige Anweisung, auch schon deshalb, weil hier einem String ein Integer-Wert zugewiesen wird.
   - `s2=s1a;` ist ebenfalls unzulässig, denn `s2` und `s1a` sind von unterschiedlichem Typ (trotz dass die beiden Typen sich nur in ihrem Namen unterscheiden).

B.5.6 Aufgabe 6

1. Ja, es würde eine implizite Konversion `enum` → `int` erfolgen, wie auch bei sonstiger Verwendung von enum-Werten in Ausdrücken mit Integer-Zahlen.

2. Nein. Der angeführte Code wird zwar vom Compiler akzeptiert, aber bei einem Ausdruck `s.x;` wird `x` nicht als `enum`-Wert ausgewertet, sondern als Variablenbezeichner innerhalb der strukturierten Variable `s`. Dadurch haben beide Namen (für den enum-Wert, für die Variable im `struct S`) unterschiedliche Bedeutung und eine Verwechslung ist durch den Kontext ausgeschlossen. Überhaupt Werte nie Variablenbezeichner sein, denn Variablenbezeichner sind ja die Namen von Wertbehältern.

3.

   (a) Variablendefinition `i`;
   (b) Typdefinition `i`;
   (c) Typdefinition `S`;
   (d) Typdefinition `S`; Variablendefinition `s`;
   (e) Typdefinition `S`; Typdefinition `s`;
   (f) Typdefinition `S`; Typdefinition `s`; Variablendefinition `s1`;
   (g) Typdefinition `A`;
   (h) Variablendefinition `A`;
   (i) Typdefinition `A`;
   (j) Variablendefinition `a`;
   (k) Typdefinition `A`; Variablendefinition `a`;
   (l) Typdefinition `A`; Typdefinition `a`;
   (m) Typdefinition `A`; Variablendefinition `a`;
   (n) Typdefinition `E`; Variablendefinition `A`;
   (o) Typdefinition `E`; Variablendefinition `A`;
   (p) Typdefinition `E`; Typdefinition `A`;

B.6 Lösungen zu Kapitel 6

B.6.1 Aufgabe 1

1. Variable vom Typ `struct` als Komponente in einem `struct`? Ja, dies ist möglich, Aufruf erfolgt dann eben mit zwei »`.`«-Operatoren. Komponente eines `struct S1` vom gleichen Typ `S1`? Nicht möglich, denn das würde zu rekursiv definierten Daten mit nicht endender Rekursionstiefe führen.

2. Sowohl Komponenten als auch Methoden in zwei verschiedenen `struct`-Typen können den gleichen Namen haben, denn sie gehören zu einer Variablen dieses Typs und sind deshalb unterscheidbar.

3. Siehe `Aufg.MaxFunktion.cc`

4. Siehe `Aufg.ggT-Funktion.cc`

5. Siehe `Aufg.kgV-Funktion.cc`

6.

```
bool ist_untergeben (PersonalNr angestellt_Wer, PersonalNr angestellt_Wem, Firma firma) {
   // Suche Feldelement von angestellt_Wer in firma
   // Voraussetzung: es existiert ein solches!
   int aWer; // Feldindex des Angestellten angestellt_Wer
   for (aWer=0; aWer<100; ++aWer) {
      // INV: firma[0] .. firma[aWer-1] ist nicht das
      //      Feld von angestellt_Wer
      if (firma[aWer].nr == angestellt_Wer)
         break; // OK, passendes Feld gefunden
   };
   // ist angestellt_Wem Chef von angestellt_Wer?
   if (firma[aWer].chef == angestellt_Wem)
      return true;
   else
      return false;
}
```

7.

```
string Firma::name(PersonalNr id) {
   // durchsuche das Feld, bis Element eine passende id hat
   for (int i=0; i<100; ++i) {
      // INV: angestellt[0] .. angestellt[i-1] ist nicht das
      //      Feld des Angestellten mit nr==i
      if (angestellt[i].nr == id)
         return angestellt[i].name;
   };
}

bool Firma::ist_untergeben(PersonalNr wer, PersonalNr wem) {
   // Suche Feldelement von wer in angestellt[]
   // Voraussetzung: es existiert ein solches!
   int wer_index; // Feldindex des Angestellten wer
   for (wer_index=0; wer_index<100; ++wer_index) {
      // INV: angestellt[0] .. angestellt[wer_index-1]
      //      ist nicht das Feld von Angestelltem wer
      if (angestellt[wer_index].nr == wer)
         break; // OK, passendes Feld gefunden
   };
   // ist Angestellter wem Chef von Angestelltem wer ?
   if (angestellt[wer_index].chef == wem)
```

```
            return true;
         else
            return false;
      }
```

B.6.2 Aufgabe 2

```
Bruch erweitere(Bruch bruch; int faktor) {
   if (faktor<0)
      if (bruch.vz==plus) bruch.vz=minus;
      else                bruch.vz=plus;
   faktor=abs(faktor);
   bruch.zaehler*=faktor;
   bruch.nenner *=faktor;
   return bruch;
}

Bruch kuerze(Bruch bruch) {
   // da ggT(0,bruch.nenner) nicht bestimmbar:
   if (bruch.zaehler == 0)
      return bruch;
   unsigned int ggt_bruch = ggT(bruch.zaehler,bruch.nenner);
   bruch.zahler = bruch.zaehler / ggt_bruch;
   bruch.nenner = bruch.nenner  / ggt_bruch;
   return bruch;
}

void kuerze(Bruch &bruch) {
   // da ggT(0,bruch.nenner) nicht bestimmbar:
   if (bruch.zaehler == 0)
      return bruch;
   unsigned int ggt_bruch = ggT(bruch.zaehler,bruch.nenner);
   bruch.zaehler = bruch.zaehler / ggt_bruch;
   bruch.nenner = bruch.nenner  / ggt_bruch;
}
void erweitere(Bruch &bruch, unsigned int faktor) {
   if (faktor<0)
      if (bruch.vz==plus) bruch.vz=minus;
      else                bruch.vz=plus;
   faktor=abs(faktor);
   bruch.zaehler*=faktor;
   bruch.nenner *=faktor;
}
void gleichnamig(Bruch &bruch1, Bruch &bruch2) {
   // gleichnamig machen, indem beide Brueche so erweitert
   // werden, dass ihre Nenner das kgV beider Nenner sind.
   unsigned int kgv_nenner = kgV(bruch1.nenner,bruch2.nenner);
   bruch1.zaehler *= kgv_nenner / bruch1.nenner;
   bruch1.nenner  = kgv_nenner;

   bruch2.zaehler *= kgv_nenner / bruch2.nenner;
   bruch2.nenner  = kgv_nenner;
}
```

In der Aufgabenlösung ist das Ergebnis der Funktion »`void gleichnamig`« bereits nach den beiden Aufrufen der Funktion erweitere gültig, die letzten beiden Anweisungen `b1.nenner=n; b2.nenner=n;` sind unnötig.

B.6.3 Aufgabe 3

```
Bruch add(Bruch bruch1, Bruch bruch2) {
   Bruch summenwert;
   int   summenwert_zaehler;
   gleichnamig(bruch1, bruch2);
   // bruch.vz * (-1) liefert (+1) für bruch.vz=plus (==0)
   // bruch.vz * (-1) liefert (-1) für bruch.vz=minus (==-1)
   summenwert_zaehler = bruch1.vz * (-1) * bruch1.zaehler
                      + bruch2.vz * (-1) * bruch2.zaehler;
   if (summenwert_zaehler < 0)
      summenwert.vz = minus;
   else
      summenwert.vz = plus;
   summenwert.zaehler = abs(summenwert_zaehler);
   summenwert.nenner  = bruch1.nenner;
   kuerze(summenwert)
   return summenwert;
}
Bruch sub(Bruch bruch1, Bruch bruch2) {
   Bruch diffwert;
   int   diffwert_zaehler;
   gleichnamig(bruch1, bruch2);
   diffwert_zaehler = bruch1.vz * (-1) * bruch1.zaehler
                    - bruch2.vz * (-1) * bruch2.zaehler;
   if (diffwert_zaehler < 0)
      diffwert.vz = minus;
   else
      diffwert.vz = plus;
   diffwert.zaehler = abs(diffwert_zaehler);
   diffwert.nenner  = bruch1.nenner;
   kuerze(diffwert);
   return diffwert;
}
Bruch mul(Bruch bruch1, Bruch bruch2) {
   Bruch mulwert;
   mulwert.zaehler = bruch1.zaehler * bruch2.zaehler;
   mulwert.nenner  = bruch1.nenner  * bruch2.nenner;
   kuerze(mulwert);
   if (bruch1.vz==minus)
      if (bruch2.vz==minus)
         mulwert.vz=plus;
      else
         mulwert.vz=minus;
   else
      if (bruch2.vz==minus)
         mulwert.vz=minus;
      else
         mulwert.vz=plus;
   return mulwert;
}
Bruch div(Bruch bruch1, Bruch bruch2) {
   Bruch divwert;
   // Division durch Multiplikation mit dem Kehrwert
   divwert.zaehler = bruch1.zaehler * bruch2.nenner;
   divwert.nenner  = bruch1.nenner  * bruch2.zaehler;
   kuerze(divwert);
   if (bruch1.vz==minus)
```

```
      if (bruch2.vz==minus)
         divwert.vz=plus;
      else
         divwert.vz=minus;
   else
      if (bruch2.vz==minus)
         divwert.vz=minus;
      else
         divwert.vz=plus;
   return divwert;
}
```

B.7 Lösungen zu Kapitel 7

B.7.1 Aufgabe 1

1.

```
// Berechne Summe von i=1 bis n ueber i
// Aufruf mit s = sum_i(n);
int sum_i (int n) {
   int sum_i=0;
   for (int i=0; i<n; ++i)
      sum_i+=i;
   return sum_i;
}
```

2.

```
namespace nonstd {
   int a;
}

int a;
int main () {
   using namespace nonstd;
   ++a;         // FEHLER: nonstd::a oder ::a ? Daher:
   ++::a;       // <- explizite Qualifizierung trotz
   ++nonstd::a; // <- using-Direktive noetig
   int a=0;
   ++a;         // lokales a, ueberdeckt nonstd::a und ::a;
   ++nonstd::a; // Ueberdeckung umgehen durch explizite Qualifizierung
}
```

Die drei Variablen `a` können unterschieden aufgrund Zugehörigkeit zu unterschiedlichen `namespace` (nämlich dem unbenannten `namespace` für alle nicht besonders zugeordneten Variablen im Programm und dem `namespace nonstd`) und der Überdeckung der beiden globalen `a` (die unterschiedlichen `namespace` angehören) durch das lokale `a`.

Der Gültigkeitsbereich einer Variablen ist ein statisches Konzept, das den Bereich im Programmtext angibt, in dem die Deklaration einer Variablen gültig ist. Der Gültigkeitsbereich global deklarierter Variablen ist daher der gesamte Programmtext von der Deklaration bis zum Ende. Da `::a` etwas später deklariert wurde als `nonstd::a`, ist sein Gültigkeitsbereich auch etwas kleiner, aber für beide erstreckt er sich bis zum Ende des Programms. Der Gültigkeitsbereich lokaler Variabler (wie hier `a` in main) erstreckt sich nur von der Deklaration bis zum Ende der Funktion. Von außerhalb der Funktion main ist hier kein Zugriff auf das darin deklarierte `a` möglich.

Die Sichtbarkeit von Variablen ist ein statisches Konzept, das den Bereich im Programmtext angibt, in dem eine Variable unter ihrem Namen auch angesprochen werden kann. Er entspricht maximal dem Gültigkeitsbereich und wird nur an solchen Stellen eingeschränkt, wo die Variable durch »lokaler« deklarierte gleiche Namen überdeckt wird. Hier werden die globalen `nonstd::a` und `::a` durch das lokale `a` von

`main()` überdeckt; sie können nicht mehr unter dem Namen `a` angesprochen werden, denn dieser meint nun die lokale Variable a. Eine Umgehung ist durch explizite Qualifizierung des `namespace` möglich.

3.

- `int max1 (int i, int j)` ist korrekt definiert. Da eine Funktion mit return endet, wird `return j;` nicht mehr ausgeführt, wenn `(i > j)` erfüllt wurde.
- `int max2 (int i, int j)` ist korrekt und entspricht `max1`.
- `int max3 (int i, int j)` ist falsch. Hier wird nichts berechnet (es gibt keinen Rückgabewert), sondern nur etwas ausgegeben. Noch dazu wird `j` in jedem Fall ausgegeben, auch wenn `i` Maximum ist.
- `int max4 (int i, int j)` ist falsch. Zwar erfolgt im Gegensatz zu `max3` die korrekte Ausgabe, aber eben keine Berechnung mit Rückgabe des Maximums über return.
- `int max5 (int i, int j)` ist korrekt. Wie bei `max1` und `max2` wird das Maximum als Wert mit `return` zurückgegeben, zusätzlich erfolgt eine korrekte Ausgabe.

4. Die Ausgabe des Programms ist »`0 0`«. Die Funktion liefert den übergebenen Wert (!, es ist in der Funktion keine indizierbare Feldvariable mehr) `j == a[i] == a[0] == 0` zurück. Die Funktion gibt das unveränderte `i == 0` aus: die lokale Variable `i` in `f (int i, int j)` hat zwar den gleichen Namen, ist aber von `i` aus `main()` unterscheidbar und überdeckt es.

5. Die Ausgabe des Programms ist »`0 1`«. Das `i` von `main()` wird in `f` verändert, weil es als Referenzparameter übergeben wurde. Welchen Wert liefert aber `return j;` zurück: das übergebene `a[i] == a[0]` zu Funktionsbeginn oder `a[i] == a[1]` aufgrund des aktuellen, veränderten Wertes von `i`? Die Variable `j` ist in der Funktion `f` eine Referenz auf das Element `a[i] == a[0]`, das beim Funktionsaufruf übergeben wurde. Alle Änderungen an `j` sind stets Änderungen am Speicherbereich vom Referenzziel `a[0]`, egal welchen Wert `i` annimmt - das Referenzziel wird dadurch nicht mehr verändert.

6. `void` ist ein Pseudotyp, der »kein Rückgabewert« meint. Die Funktion wird keinen Wert zurückliefern, wird also nicht wie ein Ausdruck, sondern wie eine Anweisung (nicht: »Befehl«) aufgerufen. Solche Funktionen heißen auch Prozeduren.

7.

```
#include <iostream>
struct S {
   int t;
   int f (S &);
   int g ();
};
int i = 0;
int t = 5;
int S::f(S &x) {
   return t+x.g();  // t aus jeweiliger Variablendeklaration vom Typ struct S
                    // x aus Parameterliste
}
int f () {
   t++;
   return t;        // t aus globaler Variablendeklaration
}
int S::g() {
   i++;
   return i;        // i aus globaler Variablendeklaration, denn weder
                    // Funktionen noch Methoden koennen auf die lokalen Variablen
                    // der aufrufenden Funktion zugreifen
}
int main () {
   S s[2];
```

```
    int i = 0;
    for (int i=0; i<2; i++) {  // i aus for-Schleife
       s[i].t = i;             // i aus for-Schleife
                               // s aus main ()
    }
    cout << s[i%2].f(s[(i+1)%2]) << endl;
                               // i aus main ()
                               // s aus main ()
}
```

Das Programm erzeugt die Ausgabe »1« (vergleiche `Aufg.AusgabeTest.cc`). Beachte: dass eine Methode als Argument den Typ bekommt, von dem sie selbst eine Komponente ist (wie hier `S::f(S &)`) ist zulässig, denn dadurch ergeben sich ja keine unendlich großen Datenstrukturen wie bei rekursiver Definition in einem `struct` (d.h. wenn die Komponente eines `struct` vom Typ sein soll wie der `struct` selbst).

8. siehe `Aufg.WieOft.cc`

9. siehe `Aufg.SwapSort.cc`

10. Es wird getauscht `i==1` mit `a[1]==3`. Denn:

    - Vor dem Funktionsaufruf `swap(i, a[f()])` ist die Wertebelegung:
      `i==0; a=={4,3,2,1}`
    - Bei Auswertung des Ausdrucks `f()` beim Funktionsaufruf wird `i==0` zu `i==1`, `a[f()] ==a[1]`
    - In der Funktion wird also auf `i==1` und `a[1]` als Referenzparameter zugegriffen, ihr Inhalt wird getauscht.
    - `i` hat in der Funktion tatsächlich schon den neuen Wert `i==1`, denn es ist Referenzparameter, d.h. call by reference (Auswertung der mittlerweile veränderten Variablen) statt call by value (Verwendung eines übergebenen Wertes).

11.

    Das Programm ist korrekt. Es erzeugt die Ausgabe:

    ```
    b= 10
    ::b= 3
    ::p= 4
    Bim::b= 14
    Pam::p= 2
    ```

    Siehe `Aufg.BimPam.cc` für kommentierten Sourcecode, welche Deklaration wo sichtbar ist.

B.7.2 Aufgabe 2

siehe `Aufg.Bruchrechnung.cc`

B.8 Lösungen zu Kapitel 8

B.8.1 Aufgabe 1

1. Mathematische Begründung dieser rekursiven Definition:

$$\begin{aligned}(x-1)^2 + 2(x-1) + 1 &= x^2 - 2x + 1 + 2x - 2 + 1 \\ &= x^2\end{aligned}$$

   Lösung siehe `Aufg.QuadratRekursiv.cc`

2. Lösung siehe `Aufg.nkRekursiv.cc`

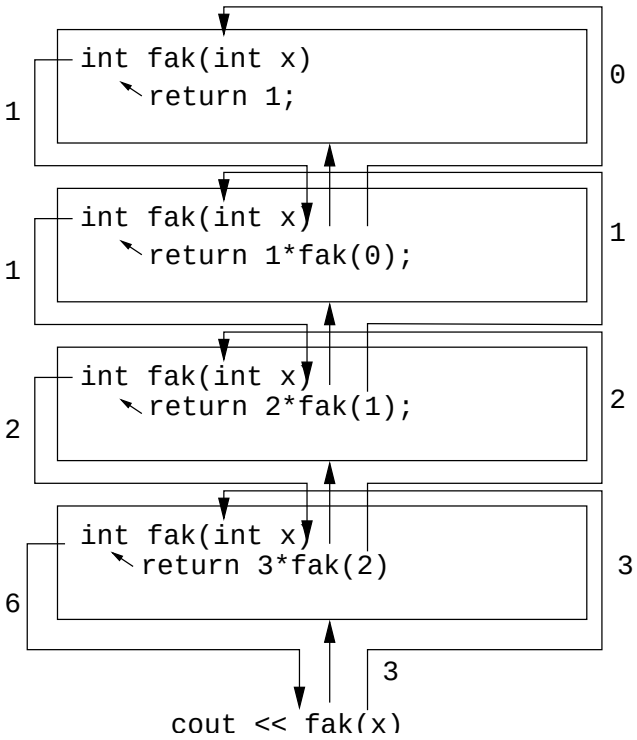

Abbildung 2: Rekursive Berechnung der Fakultät

B.8.2 Aufgabe 2

1. Zeichnung siehe Abbildung 2. Hinweise zum Debugging siehe Kapitel 1.3.2.

2. Kontrollstrukturen sind Möglichkeiten, den Ablauf eines Programms zu steuern, d.h. welche Anweisungen wann und in welcher Reihenfolge ausgeführt werden (vgl. Skript [1, Kap. 8.2, Punkt »Rekursion ist eine Kontrollstruktur«]). Es gibt: Sequenz, Verzweigung, Schleife, Rekursion. Rekursion ist eine Möglichkeit zur bedingten Wiederholung einer Funktion durch sich selbst. Sie ist daher in vielem vergleichbar mit einer Schleife, also auch eine Kontrollstruktur.

3. Mit dem Argument $x < 0$ liefert `fak(x)` aufgrund eines Wertüberlaufs $-32768 - 1 \rightarrow 0$:

$$\prod_{i=\text{INT\_MIN}}^{x} i = (-32768) \cdot (-32767) \cdot \ldots \cdot x$$

   (Bezogen auf die Funktion int `fak (int n)` im Skript [1, Kap. 8.1 »Rekursion« S. 153, entspr. S. 158 in PDF] Punkt »Direkte und indirekte Rekursion«.

4. Beide Funktionen haben gleiche Parameter und gleichen Rückgabewert. Funktion `f1` benötigt Speicherplatz für 11 Integer (`int a[10]` und `int i`) und die Instanz der Funktion selbst; sie erzeugt beim Aufruf keine weiteren Funktionsinstanzen. Funktion `f2` liegt durch rekursiven Aufruf `f2(10)` in maximal 12 Instanzen gleichzeitig vor, von denen jede einen Integer als Parameter hat, der irgendwo Speicher verbraucht (im Stack oder bei Ausführung). Da zu einer Funktionsinstanz noch weitere Informationen gehören (Art und Typ des Parameters usw.) sollte `f2` deutlich mehr Speicher verbrauchen als `f1`.

5.

   - Funktion `f1`:
     $f1\,(a, b) = a \cdot b$
     Vorbedingung: $a \geq 0$, denn sonst endet die Rekursion nicht.
     rekursive Definition:

$$a \cdot b = \begin{cases} 0 & \text{für } a = 0 \\ b + ((a-1) \cdot b) & \text{für } a \neq 0 \end{cases}$$

```
// Vorbedingung: a>=0
int f1(int a, int b) {
   int product=b;
   for (int i=1; i<a; ++i) {
      // INV: product=i*b
      product+=b;
   }
   // product=i*b mit i=a: product=a*b
   return product;
}
```

- Funktion `f2`:
  $f2\,(a,b) = a + b$.
  Vorbedingung: $b \geq 0$, denn sonst endet die Rekursion nicht
  rekursive Definition:

$$a + b = \begin{cases} a & \text{für } b = 0 \\ 1 + (a + (b - 1)) & \text{für } b \neq 0 \end{cases}$$

```
// Vorbedingung: b>=0
int f2(int a, int b) {
   int summe=a
   for (int i=0; i<b; ++i) {
      // INV: summe=a+i
      summe+=1;
   }
   // summe=a+i mit i=b: summe=a+b
   return product;
}
```

6. $f(a,b) = a + b$.

7. Eine `do while`-Schleife führt eine Anweisung (oder zusammengesetzte Anweisung) aus und dann nochmals, wenn eine Bedingung erfüllt ist. Dies kann ersetzt werden durch eine rekursive Funktion, die eine Anweisung oder zusammengesetzte Anweisung ausführt (zuerst, denn das schafft die Ausgangswerte für den nächsten Durchgang) und dann (erst dann!) die Funktion rekursiv nochmals ausführt, wenn eine Bedingung erfüllt ist:

   ```
   do {
      A (x);
   } while (P(x));
   ```

   wird zu

   ```
   void f1(T x) {
      A(x);
      if (P(x)) f1(x);
   }
   ```

8. Ist nicht möglich. Theoretisch entspräche der Aufruf der einer neuen Instanz der eigenen Funktion, deren Name mit einem formalen Parameter aufgerufen wird, zwar der Rekursion, der Compiler scheitert aber an der dazu nötigen Typdeklaration wie in folgendem Beispiel. Man kann nämlich nicht einen Typ verwenden, der noch nicht vollständig deklariert ist, kann ihn also nicht in seiner eigenen Deklaration verwenden.

   ```
   typedef int FunkPar(FunkPar, int);
   int fak(FunkPar g, int x) {
      if (x==0) return 1;
      else return x*g(g,2);
   }
   int main () {
      int x=fak(fak,3);
   }
   ```

9. Dass eine Methode als Parameter eine Variable vom gleichen Typ bekommt wie der struct, in dem die Methode definiert ist, ist möglich, wie unter Kapitel 7 dargelegt. Für structs gilt wohl nicht, dass eine Definition so lange nicht verwendet werden kann, wie sie nicht vollständig definiert ist, denn structs bestehen aus einzeln definierten Komponenten. Die einzeln definierten Komponenten dürfen die Gesamtdefinition verwenden (als Argumente von Methoden, jedoch nicht als Datenkomponententyp, sonst ergeben sich unendlich tief rekursiv definierte Daten). Beispiel: siehe Kapitel 7.

10. Diese Rekursion wiederholt sich so lange, wie die Bedingung `!P(x)` erfüllt ist, und bricht dann ab. Dies entspricht einer `while`-Schleife, denn auch diese wird so lange wiederholt, wie eine Bedingung erfüllt ist. Ein Äquivalent zum Abbruchwert `H(x)` müsste bei der Schleife als Initialisierungswert verwendet werden, aufgrund der unterschiedlichen Berechnungsrichtung von Iteration (von unten nach oben aufbauend) und Rekursion (von oben nach unten absteigend, dann in umgekehrter Reihenfolge die Ergebnisse einsammelnd).

11. In der Funktion `f` wird stets das `y` verändert, auf dessen Definition sich der »freie Bezeichner« `y` im Programmtext bezieht: auf das globale `y`. Die Funktion wird stets so aufgerufen, als stünde sie an ihrer ursprünglichen Stelle im Programmtext. Die Ausgabe ist:
```
12 // aus der Funktion f
11 // aus der Funktion g
111 // aus der Funktion main
```

12. Die Ausgabe ist:
```
1 // aus der Funktion f; das globale, unveränderte y
12 // aus der Funktion g
111 // aus der Funktion main
```

B.8.3 Aufgabe 3

1. Wichtig: Matrizen werden immer (!) als »Zeilen × Spalten« angegeben, z.B. hat eine 3 × 2-Matrix 3 Zeilen und 2 Spalten

```
//2x3-Matrix: 2 Zeilen, 3 Spalten
typedef int Zeile3[3];
typedef Zeile_3 Matrix23[2];
Matrix23 feld3x2;
int x=feld2x3[1][2]; // Zugriff: feld2x3[<zeile>][<spalte>]
//3x2-Matrix: 3 Zeilen, 2 Spalten
typedef int Zeile2[2];
typedef Zeile_2 Matrix32[3];
Matrix32 feld2x3;
int x=feld3x2[2][1]; // Zugriff: feld3x2[<zeile>][<spalte>]
```

   Sequentielle Ablage der Integer-Elemente in den Speicherzellen, angegeben durch die Indizierung des jeweiligen Matrixelements:

Matrix23	[0][0]	[0][1]	[0][2]	[1][0]	[1][1]	[1][2]
Matrix32	[0][0]	[0][1]	[1][0]	[1][1]	[2][0]	[2][1]

2. Siehe `Aufg.Matrix3x4.cc`

3. Siehe `Aufg.Matrix3x4.cc`

4. Siehe `Aufg.MatheMatrix.cc`

5. Siehe `Aufg.GaussSchema.cc`

B.8.4 Aufgabe 4

```
typedef float Func(float);
typedef float Matrix[2][3];
Matrix matrix;
forAll_in_a(Func fkt) {
```

```
   for (int i=0; i<2; ++i)
      for (int j=0; j<3; ++j)
         matrix[i][j]=fkt(matrix[i][j]);
}
```

B.8.5 Aufgabe 5

Siehe `Aufg.AusdrAnalyse.cc`

B.8.6 Aufgabe 6

Beispiel einer Grammatik in EBNF

```
<ausdruck> = [<space>] (<bruch> | <klammerausdruck>) [<space>].
<space> = " " {" "}.
<bruch> = ("+" | "-") <zaehler> ["/" <nenner>].
<zaehler> = <nullziffer> {<nullziffer>}.
<nenner> = <ziffer> {<nullziffer>}.
<nullziffer> = "0" | <ziffer>.
<ziffer> = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
<klammerausdruck> = "(" <ausdruck> <operator> <ausdruck> ")".
<operator> = "+" | "-" | "*" | "/".
```

Lösung siehe `Aufg.AusdrAnalyseBruch.cc`

C Errata

Es folgt eine Liste mit Fehlern in [1].

Kap. 2.1, S.27 in PDF: »Die zweiarmige If-Anweisung ist ein "Wenn-Dann"« ersetzen durch »Die einarmige If-Anweisung ist ein "Wenn-Dann"«.

Kap. 3.1, S.46 in PDF: `cout << "Hallo« < endl;` sollte sein `cout << "Hallo" << endl;`

Kap. 3.2, S.49 in PDF: »eingelesene zahlen addieren« ersetzen durch »eingelesene Zahlen addieren«.

Kap. 3.3, S.50 in PDF: »for (int i = 0; i < 5; ++i)« beim letzten Algorithmus ersetzen durch »for (i = 0; i < 5; ++i)«, denn hier soll ja in der for-Schleife die globale Variable nur initialisiert werden, nicht aber durch lokale Definition die globale Variable überdecken.

Kap.3.5, S.54 in PDF: ersetze »der der« durch »der«.

Kap.3.9, S.62 in PDF: ersetze »einen Auftrag andere Teilprogramme als Gehilfen zuzuweisen.« durch »einen Auftrag und andere Teilprogramme als Gehilfen zuzuweisen.«

Kap.3.9, S.64 in PDF: in Seitenmitte

```
if (... ? n wird von i ohne Rest geteilt ? ...)
   cout << i << endl;
   ++c;
```

ersetzen durch

```
if (... ? n wird von i ohne Rest geteilt ? ...) {
   cout << i << endl;
   ++c;
  }
```

Kap.4.1, S.69 in PDF: »noch die Art der Codierung wird von der Sprache festgelegt wird.« ersetzen durch »noch die Art der Codierung von der Sprache festgelegt wird.«

Kap.4.3, S.73 in PDF: ersetze »mit einer einer führenden Null« durch »mit einer führenden Null«.

Kap.4.3, S.73 in PDF: »'A' hat den ASCII Code 0110000 und 48 in Binäardarstellung ist« ersetzen durch »'0' hat den ASCII Code 0110000, und 48 in Binärdarstellung ist«.

Kap.4.3, S.74 in PDF: »j (= i-2) = 32767« ersetzen durch »(= i-1) = 32767«, so wie diese Ausgabe auch im Programm codiert ist.

Kap.4.3, S.75 in PDF: »cout << "char, unsigned char, int, unsigned int,";« ersetzen durch »cout << "char, unsigned char, short, unsigned short, int, unsigned int,";«.

Kap.4.4, S.75 in PDF: »Länge, sowie und Bitsets« ersetzen durch »Länge, sowie Bitsets«.

Kap.4.5, S.78 in PDF: »(d.h. es die kleinste Zahl« ersetzen durch »(d.h. es ist die kleinste Zahl«.

Kap.4.6, S.79 in PDF: »long : aufuellen« ersetzen durch »long : auffuellen«.

Kap.4.6, S.79 in PDF: »den arithmetische Konversionen, muss« ersetzen durch »den arithmetischen Konversionen, muss«

Kap.4.7, S.80 in PDF:

```
string s; cin << s;
int i; cin << i;
float f; cin << f;
```

ersetzen durch

```
string s; cin >> s;
int i; cin >> i;
float f; cin >> f;
```

Kap.4.7, S.80 in PDF: »s kann man neben« ersetzen durch »in s kann man neben«

Kap.4,7, S.81 in PDF: »der einzelnen Ziffen« ersetzen durch »der einzelnen Ziffern«.

Kap.4.9, S.88 in PDF: »verbessern werden kann.« ersetzen durch »verbessert werden kann.«.

Kap.4.10, S.90 in PDF: »Interger-Zahl« ersetzen durch »Integer-Zahl«.

Kap.5.1, S.93 in PDF: »Ein etwas ausführlichers« ersetzen durch »Ein etwas ausführlicheres«

Kap.5.2, S.95 in PDF: »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.)« ersetzen durch »Mit ihr soll der dritten Komponente von a die Summe des aktuellen Wertes von i und des aktuellen Wertes der (i+1)-ten Komponente von b zugewiesen werden. (Genau genommen ist es der Wert der (Wert-von-(i+1))-ten Komponente von b.)«

Kap.5.2, S.97 in PDF:

```
int main () {
   const int n = 10;
   int f[n], // f[i] soll i! enthalten
       i; f[0] = 1; i = 0;
   while (i<n) {
      i = i+1;
      f[i] = f[i-1]*i;
   }
}
```

damit nicht mehr auf f[n] zugegriffen wird, ersetzen durch

```
while (i<n-1) {
      i = i+1;
      f[i] = f[i-1]*i;
   }
```

Kap. 5.6, S.104 in PDF:

- Im Programm zum Punkt »Das Dreieck als Inhalt einer Matrix«:
  - die abschließende »}« zu »int main () {« ergänzen
  - `for (int i= 1; i< 100; ++i)` ersetzen durch `for (int i= 1; i< 100; ++i)`
  - »Vorg"angern« ersetzen durch »Vorgängern«.
- Im Programm zum Punkt »Das Dreieck zeilenweise als Feldinhalt«:
  - `for (int l=0; l<=L; ++l) a[l] = 0;` ersetzen durch
    `for (int l=0; l<L; ++l) a[l] = 0;`

Kap. 5.6, S.105 in PDF: ersetze

```
for (int k=j; k>0; --k) {
   //INV : a[k+1..j] ist neu, a[0..k] ist alt
   a[k] = a[k-1] + a[k];
}
```

durch

```
for (int k=j-1; k>0; --k) {
   //INV : a[k+1..j] ist neu, a[0..k] ist alt
   a[k] = a[k-1] + a[k];
}
```

denn der Index j hat maximal den Wert L, also hat im ersten Fall auch k maximal den Wert L; auf a[L] darf aber nicht zugegriffen werden, sondern nur auf a[L-1] (nach der Deklaration als int a[L] auf S. 104 in PDF).

Kap. 5.8, S.110 in PDF: ersetze »Platz f"ur maximal 10 Personen« durch »Platz fuer maximal 10 Personen«

Kap. 5.9.1, Punkt 1, S.113 in PDF: »Erläutern Sie weiterhin allgemein die Bedingungen unter denen $L = R$; und $L = R; L = R$; äquivalent sind!« ersetzen durch »Erläutern Sie weiterhin allgemein die Bedingungen unter denen $L = R$; und $L = R; L = R$; nicht äquivalent sind!«, denn dazu wird in den Lösungshinweisen die Antwort gegeben.

Kap. 5.9.1, Punkt 3, S.113 in PDF: »ein l-Wert« ersetzen durch »einen l-Wert«.

Kap. 5.9.4, S.115 in PDF: »Sie in Programm« ersetzten durch »Sie ein Programm«.

Kap. 5.9.5, S.115 in PDF: `b[i].s_i = a[b[i].s_i].s_i;` ersetzen durch `b[i].s_i = a[i].s_i;`, denn sonst wird mit einem zufällig gewählten Index (b[i].s_i wurde nicht initialisiert!) auf das Feld a zugegriffen, d.h. es erfolgt ein Zugriff über die Feldgrenzen hinaus und ggf. ein Laufzeitfehler »Speicherzugriffsfehler«.

Kap. 6.1, S.117 in PDF: »for (int i=0; i<y; ++i)« ersetzen durch »for (int i=0; i<x; ++i)«.

Kap. 6.1, S.118 in PDF: »Die reihenfolge der Abarbeitung« ersetzen durch »Die Reihenfolge der Abarbeitung«.

Kap. 6.2, S.121 in PDF: im Punkt »Funktionsaufruf« »cout << Min (c,d) = << min (c,d) << endl;« ersetzen durch »cout << "Min (c,d) =" << min (c,d) << endl;«

Kap. 6.3, S.124 in PDF: »// Strecke d, sag wie lang du bist!« ersetzen durch »// Strecke str, sag wie lang du bist!«

Kap. 6.6.1, Punkt 3, S. 131 in PDF: In den Lösungen wird nur eine Funktion für zwei Int-Parameter angegeben!

Kap. 6.6.1, Punkt 7, S. 131 in PDF: »der Angestellten mit der zweiten Personalnummer« ersetzen durch »des Angestellten mit der zweiten Personalnummer«. Denn eine Personalnummer wird eindeutig vergeben?

Kap. 6.6.2: Im Skript wurde bisher nicht erklärt, was Referenzparameter sind.

Kap, 6.6.2: »1. Bruch erweitere (Bruch, int);« ersetzen durch »1. Bruch erweitere (Bruch, unsigned int);«, denn dazu passt die Aufgabenlösung B.6.2 und die parallele Deklaration »2. void erweitere (Bruch &, unsigned int);« 4 Zeilen darunter.

Kap. 7.7, S.148 in PDF: »`cout << rp << endl; // Wertparameter`« ersetzen durch »`cout << rp << endl; // Referenzparameter`«

Kap. 7.7, S.149 in PDF:

```
struct S {
   int x;
   int f (int, int &);
};
```

ersetzen durch

```
struct S {
   int x;
   int f (int &);
};
```

Kap. 7.9, S.150 in PDF: »mit Hilfe Bereichsauflösungsoperators« ersetzen durch »mit Hilfe des Bereichsauflösungsoperators«.

Kap. 7.9, S.151 in PDF: »der unqualifizierte Namen« ersetzen durch »der unqualifizierte Name«

Kap. 7.9, S.153 in PDF: »die Schachtel Namenraum« ersetzen durch »die Schachtel Namensraum«

Kap. 8.1, S.159 in PDF:

»- Von dieser wird fak(3) aufgerufen. Damit wird eine zweite Instanz von fak erzeugt (ohne die erste zu beenden!).

- In fak(2) (genauer: in der Instanz von fak, die zur Berechnung von fak(2) erzeugt wurde) wird fak(1) aufgerufen und damit die dritte Instanz erzeugt.«

ersetzen durch

»- 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.«

Kap. 8.2, S.160 in PDF (Abb.15): »zweite Instanz von fak (für fax(4)) wartet auf f(3) um dann zu multiplizieren« ersetzen durch »zweite Instanz von fak (für fak(4)) wartet auf fak(3) um dann zu multiplizieren«.

Kap. 8.2, S.162 in PDF: »den Variablen x, i und tt a ab.« ersetzen durch »den Variablen x, i und altem a ab.«

Kap. 8.3, S.165 in PDF: »return zahlwert (s.substr(0, s.length()-1)) * 10 + zifferwert (s.at(s.length()-1));« ersetzen durch »return zahlwert (s.substr(0, s.length()-2)) * 10 + zifferwert (s.at(s.length()-1));«

Kap. 8.9.1, S.179 in PDF: »Aufgabe 2« ersetzen durch »Aufgabe 1«.

Kap. 8.9.1, S.179 in PDF: »n über k mit $0 < n < k$« ersetzen durch »n über k mit $0 \leq k \leq n$«.

Kap. B.2.1, Punkt 7, S.191 in PDF: »Durch Verwendung von m2 kann man m2 einsparen« ersetzen durch »Durch Verwendung von m2 kann man m3 einsparen«

Kap. B.2.4, S.192 in PDF: »und 2*b ist der alte Wert von b« ersetzen durch »und 2*b ist der alte Wert von a«.

Kap. B.3.8, S.198 in PDF: »z ist die aktuelle Zeilen- und s die aktuelle Spaltennummer« ersetzen durch »z und s sind die Zeilen- bzw. Spaltennummer der letzten beendeten Zeile« bzw. Formeln ändern.

Kap. B.4.1, Punkt 14, S.200 in PDF: Das Skript sagt (S.81; bzw. S.83 in PDF), dass Konversionen von int nach enum implizit ausgeführt werden, aber mit einer Warnung bedacht werden. Hier steht, dass keine implizite Konversion möglich ist.

Kap. B.5.5, Punkt 2, S.207 in PDF: »(in C++ in ähnlichen Sprachen)« ersetzen durch »(in C++ und ähnlichen Sprachen)«

Kap. B.5.6, Punkt 3(l), S.207 in PDF: »Typ, Variable:« ersetzen durch »2 Typen:«.

Kap. B.6.3, S.210 in PDF: »(gekuerze) Brueche« ersetzen durch »(gekuerzte) Brueche«

Kap. B.8.2, Punkt 3, S.214 in PDF: »Endlos-Rekursion« ersetzen durch »Mit dem Argument $x < 0$ liefert `fak(x)` aufgrund eines Wertüberlaufs $-32768 - 1 \rightarrow 0$:

$$\prod_{i=\text{INT\_MIN}}^{x} i = (-32768) \cdot (-32767) \cdot \ldots \cdot x$$

«

Kap. B.8.2, Punkt 4, S.214 in PDF: »11 mal eine int-Variable« ersetzen durch »12 mal eine int-Variable«; »von 11 Instanzen von f« ersetzen durch »von 12 Instanzen von f«.

Kap. B.8.2, Punkt 5, S.214 in PDF: »Beide haben als Vorbedingung, dass der erste Parameter 0 oder positiv sein muss. Bei f2 muss der zweite Parameter ebenfalls 0 oder positiv sein.« ersetzen durch »Vorbedingungen: Bei f1 muss der erste, bei f2 der zweite Parameter 0 oder positiv sein, damit die Rekursion endet und ein korrektes Ergebnis liefert.«

Kap. B.8.2, Punkt 7, S.215 in PDF: »Jede Schleife der Form `while (P(x)) do A(x);` kann in die äquivalente rekursive Funktion `void wF (T x) { if (P(x)) { A(x); wF(x) }` umgesetzt werden.« ersetzen durch:

```
do {
   A (x);
} while (P(x));
```

wird zu

```
void f1(T x) {
   A(x);
   if (P(x)) f1(x);
}
```

D Frequently Asked Questions

D.1 Warum entstehen beim Kompilieren "`undefined reference to cout`"?

Wenn `#include <iostream>` eingebunden wurde, so liegt das einfach daran, dass der falsche Compiler benutzt wird: `gcc` ist ein C-Compiler, `g++` ein C++-Compiler, trotz dass beide miteinander in einem Paket integriert sind. Der Aufruf müsste also z.B. lauten:
`g++ -Wall -pedantic Aufg.HelloWorld.cc`

D.2 Warum bekomme ich `int`-Werte bei Division, trotz dass der Ergebnistyp `float` ist?

In einer Anweisung wie `pi = pi + 1/folgeglied;` (bei einem Anfangswert $pi_0 = 1$; `float pi; int folgeglied;`) erfolgt eine ganzzahlige Division, keine float-Division. Warum? Weil beide Argumente der Division Integer sind. Ist eines der beiden Elemente ein float, so erfolgt implizite Konversion des anderen Arguments von Integer zu float. Beispiel: `pi = pi + 1.0/folgeglied;`. Die Zahldarstellung im Programm (`1` oder `1.0`) zeigt also an ob diese Zahl ein Integer oder ein float sein soll.

D.3 Wie bekomme ich den Betrag einer `float`-Zahl?

Man benutze die Funktion `fabs(float)` aus der Bibliothek `cmath`.

D.4 Wie kann ich die Ausgabe von `cout` formatieren?

Die Verwendung des Attributs `cout.width(x)` sorgt dafür, dass alle Zahlen bei der nächsten (und nur bei der nächsten) Ausgabe `x` Stellen einnehmen. Weitere Möglichkeiten zur Ausgabeformatierung sind dokumentiert im Skript [1, Kap. A.2 »Ausgabeformate«].

D.5 Woran liegt die Fehlermeldung »`parse error before '{'; label '<value>' not in switch statement;`«?

Eine Möglichkeit ist, dass die Klammerung um den Ausdruck des switch-Statements unvollständig ist, wie z.B. in:

```
switch (P == ((Q == P) && R) {
   case true: cout << "P= " << endl; break;
   case false: cout << "Q= " << endl; break;
}
```

D.6 Worauf weist die Compilermeldung »`WARNING: statement with no effect`« hin?

Diese Meldung kann bei `for`-Schleifen anzeigen, dass die Anweisung zur Veränderung der Laufvariablen diese nicht verändert und deshalb eine Endlosschleife auftreten kann (siehe Beispiel). Die Meldung wird dabei irritierenderweise für die letzte Zeile des Blocks der `for`-Schleife ausgegeben, denn hier wird die betreffende Anweisung erst ausgeführt, nicht schon zu Beginn der `for`-Schleife.

```
for (unsigned int z=1; z <= hbitval && z!=0; z<<1) {
  // ...
}
```

richtig wäre:

```
for (unsigned int z=1; z <= hbitval && z!=0; z=z<<1) {
  // ...
}
```

D.7 Wie sieht die Operatorenrangfolge inkl. der bitweisen Operatoren aus?

Auf jeden Fall bindet `==` stärker als `&`, so dass `(i&z==z)` eigentlich meint `(i&(z==z))`, äquivalent `(i&1)`, was einen Wahrheitswert entsprechend des letzten Bites von `i` ergibt und keinen abhängig von dem in `z` gesetzten Bit.

D.8 Woran liegt der Compilerfehler »`case label '"s"' does not reduce to integer constant`«?

Der Compiler meint: der Wert des für die switch-Anweisung ausgewerteten Ausdrucks ist eine Integer-Konstante (z.B. ist der Rückgabewert von wort.at(pos) ein char, also eine Integer-Konstante, und kein String mit einem Zeichen!). Die notierten `case`-Labels sind jedoch keine Integer-Konstanten und lassen sich auch nicht auf solche reduzieren (in diesem Fall sind es `string`'s). Eine solche oder ähnliche Fehlermeldung ist also immer ein Hinweis auf Typinkompatibilität zwischen Ergebnis des `switch`-Ausdrucks und den `case` labels.

D.9 Was bedeutet »Speicherzugriffsfehler« während der Laufzeit eines Programms?

Ein Programm hat versucht, Speicherzellen auszulesen, die außerhalb des für dieses Programm reservierten Bereichs liegen. Das deutet auf einen Programmierfehler hin, z.B. auf die Verwendung falscher Indizes für Felder, so dass der Zugriff über die Feldgrenzen hinaus erfolgt.

D.10 Wie kann man sich im Debugger gdb den Rückgabewert einer Funktion ausgeben lassen, wenn dieser ein Ausdruck aus mehr als einer Variablen ist? Wie geht das bei rekursiven Funktionen?

E Dateiliste der beiliegenden C++ Dateien

Es sind alle in einem Programm zu realisierenden Lösungen zu den Übungsaufgaben aus [1] (Dateiname beginnt mit `Aufg.*`; Verweis auf den jeweils richtigen Dateinamen von den Lösungshinweisen () aus.). Zusätzlich sind ein paar Programme, die in [1] vorgestellt wurden, zu Testzwecken enthalten (Dateiname beginnt mit `Test.*`).

- `Aufg.ASCII-Code.cc`
- `Aufg.AfolgtB.cc`
- `Aufg.AusdrAnalyse.cc`
- `Aufg.AusdrAnalyseBruch.cc`
- `Aufg.AusgabeTest.cc`
- `Aufg.BimPam.cc`
- `Aufg.BitmusterInt.cc`
- `Aufg.Bruchrechnung.cc`
- `Aufg.EnumSelektor.cc`
- `Aufg.FakRekursiv.cc`
- `Aufg.FeldOhneDuplikate.cc`
- `Aufg.FeldVergleich.cc`
- `Aufg.Fibonacci.cc`
- `Aufg.FolgeFx.cc`
- `Aufg.FunkParameter.cc`
- `Aufg.GaussSchema.cc`
- `Aufg.GeradeZahl.cc`
- `Aufg.MatheMatrix.cc`
- `Aufg.Matrix3x4.cc`
- `Aufg.MaxFunktion.cc`
- `Aufg.MinAus4.Hilfsvar0.cc`
- `Aufg.MinAus4.Hilfsvar1.cc`
- `Aufg.MinAus4.Hilfsvar2.cc`
- `Aufg.MinAus4.HilfsvarN.cc`
- `Aufg.MinMaxAus4.cc`
- `Aufg.N_ueber_K.cc`
- `Aufg.NaeherungEX.cc`
- `Aufg.NaeherungPi.cc`
- `Aufg.ProduktIterativ.cc`

- `Aufg.ProduktSumme.cc`
- `Aufg.PruefProgramm.cc`
- `Aufg.QuadratRekursiv.cc`
- `Aufg.StringOp.cc`
- `Aufg.Summe.nZahlen.cc`
- `Aufg.Summe2i.cc`
- `Aufg.Summe2iInduktion.cc`
- `Aufg.SummeUndMittelwert.cc`
- `Aufg.SwapSort.cc`
- `Aufg.TabelleUngeradeZahlen.cc`
- `Aufg.TeilerProbe.cc`
- `Aufg.Wahrheitstafel.cc`
- `Aufg.WieOft.cc`
- `Aufg.WurzelHeron.cc`
- `Aufg.Zahlsytemwandlung.cc`
- `Aufg.ggT-Funktion.cc`
- `Aufg.kGroesstes.cc`
- `Aufg.kgV-Funktion.cc`
- `Aufg.nkRekursiv.cc`
- `TestFak.cc`
- `TestInt.cc`
- `TestNamespace.cc`
- `TestPascalDreieck.cc`

Literatur

[1] »Programmierung 1 C++«; Thomas Letschert; FH Gießen-Friedberg (Version vom 10. September 2001). Dieses Skript ist die Arbeitsgrundlage für das vorliegende Dokument und so gut, dass es keinesfalls ersetzt werden muss. Deshalb beschränkt sich das vorliegende Dokument auf einige Ergänzungen, Lösungsvorschläge und Verbesserungen und stellt keinesfalls ein vollständiges Dokument zur Vorlesung Programmieren 1 dar. Bezugsquellen für das Skript von Prof. Letschert: zu Semesterbeginn in der Fachschaft Informatik der FH Gießen-Friedberg für ca. 6 EUR als gedruckte Version. Oder als PDF-Datei von der Homepage von Prof. Letschert: und kostet 12DM. Es ist auch im Internet erhältlich: gelinkt auf der Hompage von Prof. Letschert http://homepages.fh-giessen.de/~hg51/ nach Prog1-Index http://velociraptor.mni.fh-giessen.de/Programmierung/prog-1.html und von dort zu Dokumenten in PDF http://velociraptor.mni.fh-giessen.de/Programmierung/progI.pdf, PostScript und HTML.

[2] Von Professor Lauwerth im internen FH-Netz zur Verfügung gestellte Dateien, darunter die Übungsblätter. Vom Explorer auf einem NT-Rechner in einem Labor aus: »`Netzwerkumgebung | Gesamtes Netzwerk | Microsoft [...] | Omnibus | Alabama | Public | Lauwerth | PGI`«, dann verbinden als `guest113`. Verfahren des Zugriffs auf den Server Alabama von den Sun-Unix-Workstations aus, entsprechend nach einem ssh-Login von jedem beliebigen anderen Rechner aus: `smbclient //ALABAMA/PUBLIC -U student`, es muss kein Passwort angegeben werden.

[3] Stroustrup, Bjarne: »The C++ Programming Language«; 92 DM. Es eignet sich für das gesamte Studium und später beim Arbeiten zum Nachschlagen, jedoch nicht für Anfänger zum Lernen. Es gibt auch eine deutsche Fassung, die jedoch nicht empfehlenswert ist, weil man als Informatiker ohnehin Englischkenntnisse benötigt.

[4] Breymann, Ulrich: »C++ - Einführung und professionelle Programmierung«; 80DM. Mit einer CD, die u.a. den Compiler `gcc` und den Editor `PFE` (Programmer's File Editor) enthält. Dieses Werk ist als eine Einführung gedacht.

[5] »C/C++ Unterlagen Weltweit«; Linksammlung als Ergänzung zu einführenden Lehrveranstaltungen in C und C++. http://nestroy.wi-inf.uni-essen.de/Lv/cpp/weltweit.html. Enthält Verweise auf FAQs usw.

[6] »Programmierung in C++ :: Elemente von C++-Programmen (2. EInheit C++)«. Folien zu einer Veranstaltung der Universität Essen. Quelle: http://kom.wi-inf.uni-essen.de/download/c%2B%2B/C%2B%2B%20Skript02neu.pdf. Eine knapp gefasste referenzatige Darstellung der Sprache C++ von Anfang an. Daher trotz der enthaltenen Beispiele nur bei Vorkenntnissen zu empfehlen.