

Praktikum

zu der Veranstaltung

Programmierung mit C++

Teil 1

Bachelor-Studiengang Informatik

PROF. DR.-ING. FRITZ MEHNER

Fachhochschule Südwestfalen

Fachbereich Informatik und Naturwissenschaften

E-Mail: mehner.fritz@fh-swf.de

© 2006-2013 Fritz Mehner

Version 2.0

Stand 19. November 2013

Inhaltsverzeichnis

Einführung	v
1. Summation unendlicher Reihen	1
1.1. Summenbildung bei endlichen Reihen	1
1.2. Summenbildung bei unendlicher Reihen	2
1.3. Reihensummen bekannter Funktionen	3
1.4. Summe der Reihe $1 \cdot 3 - 3 \cdot 5 + 5 \cdot 7 \dots$	3
2. Verwendung von Schleifen	5
2.1. ASCII-Tabelle	5
2.2. Rundungsfehler durch fortgesetzte Multiplikation und Division	6
2.3. Steuerung einer for -Schleife	6
3. Verwendung von Feldern	7
3.1. Feld mit Zufallszahlen belegen	7
3.2. Kennwerte einer Zahlenreihe aus einer Datei	7
3.3. Korrespondierende Felder	7
3.4. Erzeugung einer Graphik mit Hilfe von gnuplot	8
4. Umgang mit Texten	11
4.1. Buchstaben und Ziffern in einem Text abzählen	11
4.2. Wörter in einem Text abzählen	11
4.3. Wörter und Zahlen in einem Text abzählen	12
5. Codieren und Decodieren von Texten	13
5.1. Die Verschiebeciffre ROT13	13
5.2. Caesar-Verschlüsselung	14
5.3. Rotation durch Indexrechnung	15
6. Einfache statistische Kennwerte	17
6.1. Statistische Kennwerte	17
6.2. Sortieren durch Vertauschen	18
6.3. Berechnung des geometrischen Mittelwertes	19
7. Bildbearbeitung 1	21
7.1. Bilddatei lesen und speichern	21
7.2. Grauwertbild bearbeiten	22
7.3. Bilderzeugung	26
8. Bildbearbeitung 2 / Mengen	27
8.1. Bildbereich füllen	27
8.2. Darstellung von Mengen kleiner Zahlen	28
8.3. Beliebige große Mengen	28

8.4. Erzeugung von Permutationen	29
8.5. Sudoku-Rätsel lösen	30
A. Erstellen von Programmen	33
A.1. Editieren, Compilieren, Binden	33
A.2. Endlosschleife beenden	36
B. Programmdokumentation	37
B.1. Dateibeschreibung	37
B.2. Abschnittskommentare und Zeilenendkommentare	38
B.3. Einrückung und Schachtelungstiefe	39
C. Fehlersuche	41
C.1. Die häufigsten Anfängerfehler	41
C.2. Der Debugger DDD	42
Stichwortverzeichnis	45

Einführung

*Lehre bildet Geister,
doch Übung macht den Meister.*
Deutsches Sprichwort

Das vorliegende Dokument enthält die Praktikumsaufgaben zu der Veranstaltung „Programmierung mit *C++ 1*“ im Studiengang *Informatik* des Fachbereiches *Informatik und Naturwissenschaften*.

Die Durchführung der Praktikumsaufgaben gibt Gelegenheit, das in der Vorlesung Gehörte anzuwenden und zu vertiefen. Programmieren kann man nur lernen indem man programmiert. Wenn keine Vorkenntnisse vorhanden sind, können die Anfänge durchaus mühevoll sein. Die alleinige Teilnahme am Praktikum, das heißt Programmieren am Rechner im Umfang von zwei Semesterwochenstunden, reicht *keinesfalls* zum Erwerb gefestigter Grundkenntnisse aus.

Es ist zwingend erforderlich, auch außerhalb der Lehrveranstaltungen das in der Vorlesung erworbene Wissen praktisch nachzuvollziehen, anzuwenden und zu vertiefen.

Dazu sollte auf dem eigenen Rechner eine entsprechende Entwicklungsumgebung zur Verfügung stehen. Grundsätzlich ist jeder Compiler oder jede Entwicklungsumgebung geeignet, sofern die aktuellen Sprachnormen weitgehend erfüllt sind. Aus Kostengründen bietet sich freie Software an (**Linux** oder **FreeBSD**). Bei allen **Unix**-Betriebssystemen gehören **C-/C++**-Compiler zur Grundausstattung und werden durch weitere leistungsfähige Entwicklungswerkzeuge ergänzt. Selbstverständlich können stets auch die Einrichtungen der Fachhochschule benutzt werden. Für Windows-, Linux- und Mac-Benutzer steht der freie GNU-**C/C++**-Compiler zum Beispiel zusammen mit verschiedenen integrierten Entwicklungsumgebungen zur Verfügung.

Der zur Lösung der jeweiligen Aufgaben benötigte Stoff entspricht natürlich dem Stand, der in der Vorlesung bis zur Bearbeitungszeit erreicht wurde. Das heißt jedoch nicht, daß es für einzelne Aufgabenteile nicht elegantere Lösungen gibt, die aber Kenntnisse erfordern, die im Augenblick noch nicht vermittelt wurden.

Durchführung des Praktikums

Teilnahme und Testat Die Teilnahme am Praktikum und die selbständige Bearbeitung der Praktikumsaufgaben sind Pflicht.

Die *Anwesenheitspflicht* gilt als erfüllt, wenn für mindestens 80 Prozent der Praktikumstermine (siehe unten) die Anwesenheit durch den Betreuer bescheinigt wurde.

Die *Bearbeitungspflicht* gilt als erfüllt, wenn für mindestens 80 Prozent der Praktikumsaufgaben die selbständige und erfolgreiche Bearbeitung durch den Betreuer bescheinigt wurde.

Wenn beide Bedingungen erfüllt sind, dann wird am Ende des Semesters der Erwerb der Vorleistung bescheinigt. Nur diese Vorleistung berechtigt zur Teilnahme an der Klausur.

Terminplan Für die Testierung der Aufgaben gibt es einen Terminplan der auf der Internetseite zu der jeweiligen Veranstaltung eingesehen werden kann. Dieser Terminplan ist *verbindlich*. Eine Nachfrist für eine Aufgabe wird nur dann gewährt, wenn zum festgesetzten Termin die Lösung im wesentlichen vorliegt und deshalb nur Schwächen oder Fehler zu beheben sind.

Testattermin ist der Termin der Praktikumsgruppe (Doppelstunde), der der einzelne Teilnehmer zugeordnet ist. Die zu testierenden Aufgaben müssen im wesentlichen zu *Beginn dieses Praktikumsstermins* vorliegen, um den Betreuern die Durchsicht der Aufgaben aller Teilnehmer zu ermöglichen.

Vorbereitung und Durchführung Die erfolgreiche Durchführung des Praktikums setzt voraus, daß der zugrundeliegende *Stoff im wesentlichen bekannt* ist und daß die Aufgabenstellung vor Beginn des jeweiligen Praktikums *vollständig gelesen und verstanden* wurde.

Die Zeitdauer von zwei Wochenstunden pro Praktikumstermin wird in der Regel nicht zur vollständigen und richtigen Bearbeitung der geforderten Aufgaben ausreichen, so daß wesentliche Teile der Lösung auch außerhalb des Praktikums erarbeitet werden müssen.

Die Praktikumstermine dienen unter anderem zur Klärung der Aufgabenstellung und bieten Gelegenheit, Einzelfragen zum Stoff der Vorlesung mit dem Betreuer zu besprechen. Weiterhin ist die Diskussion der gewählten Lösungswege und die Festlegung von Verbesserungen und Berichtigungen ein wesentlicher Zweck der Veranstaltung.

Programmdokumentation Für die zu erstellenden Programme gilt ein *Mindeststandard* für die Programmdokumentation der in Anhang B erläutert ist. Nicht oder mangelhaft dokumentierte Programme werden nicht anerkannt.

Programmtests Grundsätzlich ist *jedes vorzulegende Programm* vorher vom Autor zu testen. Das geschieht in der Regel durch die Implementierung der jeweils geforderten Tests. Diese sind je nach Aufgabenstellung durch Nachrechnen oder durch stichprobenartige Überprüfung zu kontrollieren.

Wenn Testfälle vorgegeben sind (zum Beispiel Zahlenwerte für Gleichungskoeffizienten und ähnliches), dann müssen diese zur Vereinfachung der Kontrolle natürlich in der angegebenen Form verwendet werden.

Tests sind selbstverständlich auch dann durchzuführen, wenn Sie nicht ausdrücklich in der Aufgabenstellung gefordert werden.

Im Abschnitt C.1 sind einige typische Anfängerfehler aufgelistet. Sie sollten unbedingt lernen, diese Fehler zu vermeiden! Überprüfen Sie Ihre Lösungen vor der Abgabe auch mit Hilfe dieser Liste.

Rechnerplattform Die vorzulegenden Programme müssen auf den im Praktikum zur Verfügung stehenden **Linux**-Installationen ohne Fehler und Warnungen übersetzbar, bindbar und ausführbar sein.

Freiwillige Zusatzaufgaben Einzelne Übungen enthalten freiwillige Bestandteile die in der jeweiligen Überschrift mit der Kennzeichnung *freiwillig* ausgewiesen sind.

Die Bearbeitung dieser Teile wird empfohlen, ist aber nicht zur Erlangung der Vorleistung erforderlich. Die erarbeiteten Lösungen können natürlich trotzdem den Praktikumsbetreuern zur Beurteilung vorgelegt werden.

Bewertung der Aufgaben

Ziel der Bearbeitung einer Aufgabe ist natürlich die vollständige Bearbeitung und Lösung der Aufgabenstellung. Bei der Testierung können jedoch Teilaufgaben anerkannt werden, sodaß ein mangelhafter Anteil nicht das gesamte Testat in Frage stellt. Tabelle 1 listet die Punkte auf, die bei den einzelnen Teilaufgaben erzielt werden können. Die Numerierung der Teilaufgaben stimmt mit den Abschnittsnummern überein (Aufgabe 3, Teil 3 findet sich in Abschnitt 3.2). Einige wenige Abschnitte sind nicht aufgeführt, weil diese Erläuterungen zu den davorstehenden Aufgabenteilen enthalten und somit keine eigene Wertung besitzen.

Programmierung mit C++ 1

Aufgabe	Teil 1	Teil 2	Teil 3	Teil 4	Summe
1	7	3	-	-	10
2	4	2	4	-	10
3	2	2	3	3	10
4	3	3	4	-	10
5	4	6	-	-	10
6	5	5	-	-	10
7	4	6	-	-	10
8	4	6	-	-	10
					Σ 80

Tabelle 1.: Aufteilung der Punkte

Gesamtpunktzahl Insgesamt können 80 Punkte erreicht werden. Mit **64 Punkten** (80 Prozent) ist die Bearbeitungspflicht erfüllt.

Praxisbezug der Praktikumsaufgaben

Die Praktikumsaufgaben dienen dazu, den in der Vorlesung behandelten Stoff anzuwenden und zu vertiefen. Die Veranstaltungen **Programmierung mit C++ 1** und **Programmierung mit C++ 2** sind als Erstausbildung in einer höheren Programmiersprache angelegt. Bei der Auswahl der Beispiele muß deshalb entweder auf geeigneten Stoff aus gleichzeitig laufenden Veranstaltungen (zum Beispiel **Grundlagen der Informatik**) oder auf Beispiele aus der Mathematik zurückgegriffen werden, da hier mindestens der Kenntnisstand der Fachhochschulreife erwartet wird.

Bei Anwendungen, wie etwa einer Gerätesteuerung, einer Bildauswertungs-Software, einem Compiler oder einem Zeichenprogramm ist der Praxisbezug ganz offensichtlich gegeben. Die Erfahrung zeigt jedoch, daß die Behandlung solcher Beispiele, oder auch nur größerer Teile daraus, bereits erhebliche Kenntnisse der jeweiligen Anwendungsgebiete und fortgeschrittene Programmierkenntnisse voraussetzen. Der Anfänger müßte sich mit zwei Problemfeldern gleichzeitig auseinandersetzen. Das bedeutet jedoch keineswegs, daß die gewählten Praktikumsaufgaben nicht praxisbezogen sind – ganz im Gegenteil! Das Erlernen einer Programmiersprache beginnt mit dem Kennenlernen der Sprachbestandteile und deren Bedeutung. In weiteren Schritten müssen unter anderem einfache Vorgehensweisen und Trivialalgorithmen für immer wiederkehrende Anwendungssituationen erlernt werden. Hierzu zählen zum Beispiel wechselnde Vorzeichen in Zahlen- und Aufruffolgen, Wertetausch bei Variablen, die Bestimmung von Summen, Minimal- und Maximalwerten von Zahlenmengen und ähnliches. Der Umgang mit Zahlen, die im Gegensatz zur Mathematik endlich und nicht beliebig genau sind, muß ebenso geübt werden, wie etwa die Vermeidung häufig anzutreffender Fehler (siehe auch Abschnitt C.1) oder die Erzielung von logisch vollständigen Lösungen. Die saubere Formatierung und Kommentierung von Programmen, sowie die Durchführung von systematischen Tests und die Fehlersuche sind als nächste Schritte zu nennen. Diese und weitere Kenntnisse und Fähigkeiten sind unabdingbar für jeden, der später im Beruf verantwortlich Programme entwickeln möchte.

Aus den genannten Gründen sind die gerade genannten erworbenen Fähigkeiten praktisch in höchstem Maße wichtig, da man ohne sie nicht davon sprechen kann, eine Programmiersprache zu beherrschen.

Zur Darstellung

Programmcode, Programmausgaben, Programm- und Dateinamen, Schlüsselwörter von Programmiersprachen und Menüeinträge erscheinen in **Schreibmaschinenschrift mit fester Zeichenbreite**. In Codebeispielen und Listen werden für Code und Kommentar zur Verbesserung der Lesbarkeit unterschiedliche Schriftstile der Schriftart **luximono** verwendet: Schlüsselwörter sind halbfett gesetzt (zum Beispiel **double**), Kommentare sind kursiv gesetzt (zum Beispiel *//Schleifenzähler*), alle anderen Programmbestandteile sind im Normalschnitt gesetzt (zum Beispiel `erg = fkt3(y);`).

Dieses Dokument wurde in L^AT_EX 2_ε unter **Linux** erstellt.



1. Summation unendlicher Reihen

1.1. Summenbildung bei endlichen Reihen

7 Punkte

$$H = \sum_{i=1}^{\infty} \frac{1}{i} = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots \quad (1.1)$$

$$A = \sum_{i=1}^{\infty} \frac{(-1)^{i+1}}{i} = \frac{1}{1} - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \dots \quad (1.2)$$

$$L = \sum_{i=1}^{\infty} \frac{(-1)^{i+1}}{2i-1} = \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots \quad (1.3)$$

$$G = \sum_{i=0}^{\infty} \frac{1}{2^i} = \frac{1}{1} + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots \quad (1.4)$$

Gleichung	Name	Summe
1.1	harmonische Reihe	∞
1.2	alternierende harmonische Reihe	$\ln 2$
1.3	Leibnizsche Reihe	$\pi/4$
1.4	Geometrische Reihe	2

Tabelle 1.1.: Unendliche Reihen

Die Bildung einer Summe S aus n Summanden wird mit Hilfe einer Schleife durchgeführt. Bei jedem Schleifendurchlauf wird ein Summand s_i zu der sich aufbauende Summe S addiert:

$$S = s_1 + s_2 + s_3 + \dots + s_n$$

$$S = 0$$

$$S = S + s_1$$

$$S = S + s_2$$

$$\dots$$

$$S = S + s_n$$

1.1.1. Wechselnde Vorzeichen

Wechselnde Vorzeichen (Reihe 1.2 und 1.3) werden dadurch gebildet, daß man einer Hilfsvariablen v vor der Schleife den Wert $+1$ zuweist. Die Summanden werden in der Schleife mit der Vorzeichenva-

riablen v multipliziert. Danach wird mit der Zuweisung $v = -v$ die Umkehr des Vorzeichens für den nächsten Durchlauf erzwungen.

1.1.2. Nichtlinear wachsende oder fallende Summanden

Die Summanden der Geometrischen Reihe können ohne Potenzierung berechnet werden. Der jeweils nächste Summand ergibt sich durch Multiplikation mit dem Faktor $\frac{1}{2}$ aus dem vorhergehenden. Wenn die Variable `summand` den Wert des Summanden aus dem letzten Schleifendurchlauf besitzt, dann kann durch

```
summand = 0.5*summand;
```

der Wert für den aktuellen Schleifendurchlauf bestimmt werden.

Schreiben Sie ein `C++`-Programm, in welchem für jede dieser Reihen die Summe der ersten 1000 Summanden gebildet und mit fünf Nachkommastellen ausgegeben wird.

Reihe	Summe der ersten 1000 Summanden
harmonische Reihe	_____
alternierende harmonische Reihe	_____
Leibnizsche Reihe	_____
Geometrischen Reihe	_____

1.2. Summenbildung bei unendlicher Reihen

3 Punkte

Die Bildung von Summen unendlicher Reihen durch Aufaddieren ist auf einem Digitalrechner natürlich nicht möglich. Wegen der begrenzten Genauigkeit der Zahlendarstellung und der Tatsache, daß die Summanden einer konvergierenden Reihe immer kleinere Beträge annehmen, verändert die Addition weiterer Summanden eine bereits berechnete Summe ab einem gewissen i nicht mehr! Wenn die Summe aus dem letzten Rechenschritt als Vergleichswert zur Verfügung steht, kann dieser Umstand festgestellt und die Berechnung abgebrochen werden. Die bis dahin ermittelte Summe stellt eine Näherungslösung dar.

Erstellen Sie eine neue Version des Programmes aus dem ersten Teil, in welchem Sie die Geometrische Reihe solange aufsummieren, bis sich die Summe nicht mehr ändert.

Geben Sie die Anzahl der Summanden und die errechnete Summe (16 Nachkommastellen) aus und vergleichen Sie die Summe mit dem Grenzwert aus Tabelle 1.1.

Summe der Geometrischen Reihe: _____

Anzahl der Summanden: _____

1.3. Reihensummen bekannter Funktionen *freiwillig*

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} \dots \quad (1.5)$$

$$\cosh(x) = \sum_{k=0}^{\infty} \frac{x^{2k}}{(2k)!} = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} \dots \quad (1.6)$$

$$\cos(x) = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k}}{(2k)!} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} \dots \quad (1.7)$$

Viele mathematische Funktionen sind durch Reihensummen definiert. Dazu gehören auch die Exponentialfunktion (1.5), die Funktion Cosinus hyperbolicus (1.6) und die Kosinusfunktion (1.7).

Erstellen Sie ein Programm, in welchem die Reihensummen für den Wert $x = 1.2$ solange aufsummiert werden, bis sich die jeweilige Summe nicht mehr ändert. Dabei soll der Summand des aktuellen Schleifendurchlaufes aus dem Summanden des letzten Durchlaufes gebildet werden. Für den i -ten Summanden s_i der Exponentialfunktion gilt zum Beispiel

$$s_i = \frac{x^i}{i!} = \left(\frac{x}{i}\right) \cdot \frac{x^{i-1}}{(i-1)!} = \left(\frac{x}{i}\right) \cdot s_{i-1} \quad ; i > 0 \quad (1.8)$$

Der i -te Summand kann also für alle $i > 0$ durch Multiplikation mit dem Faktor x/i aus dem jeweils vorhergehenden gebildet werden. Dadurch wird die Berechnung der Fakultät und die Potenzierung vollständig vermieden. Ermitteln Sie für die beiden anderen Reihen den entsprechenden Zusammenhang.

$$\begin{array}{ll} e^{1.2} & \underline{\hspace{2cm}} \\ \cosh(1.2) & \underline{\hspace{2cm}} \\ \cos(1.2) & \underline{\hspace{2cm}} \end{array}$$

Kontrollieren Sie die Ergebnisse mit dem Taschenrechner.

Anmerkung. Die Berechnung der Reihensummen (1.5-1.7) stellt eine experimentelle Nachprüfung der Reihendarstellung dar und ist für praktische Berechnungen nicht gut geeignet. Leistungsfähige Prozessoren bieten eine Hardware-Berechnung dieser Funktionen, für andere Fälle gibt es geeignete Näherungsverfahren. In eigenen Programmen werden normalerweise die C-Bibliotheksfunktionen `exp()`, `cosh()` und `cos()` und so weiter verwendet (`include`-Datei `cmath`). Diese Funktionen können ebenfalls zur Kontrolle der Reihensummen verwendet werden.

1.4. Summe der Reihe $1 \cdot 3 - 3 \cdot 5 + 5 \cdot 7 \dots$ *freiwillig*

Erstellen Sie ein Programm, das die Summe der Reihe

$$S = \sum_{k=1}^n (-1)^{k+1} (2k-1) \cdot (2k+1) = 1 \cdot 3 - 3 \cdot 5 + 5 \cdot 7 \dots (-1)^{n+1} (2n-1) \cdot (2n+1) \quad (1.9)$$

für $n = 10$ berechnet. Versuchen Sie anschließend, für gerade Werte von n eine geschlossene Formel zu finden, sodaß keine Schleife benötigt wird.

2. Verwendung von Schleifen

2.1. ASCII-Tabelle

4 Punkte

Schreiben Sie ein *C++*-Programm, welches in 32 Zeilen eine vierspaltige ASCII-Tabelle ausgibt (Liste 2.1), die jeweils den oktalen, dezimalen und hexadezimalen Zahlenwert eines Zeichens (dreistellig mit führenden Nullen) und das zugehörige abdruckbare Zeichen enthält. Anstelle der nicht darstellbaren Steuerzeichen (0-31, 127) sollen drei Punkte ausgegeben werden.

Liste 2.1: Programmausgabe: ASCII-Tabelle

*** ASCII-Tabelle ***															
Okt	Dez	Hex	Zch	Okt	Dez	Hex	Zch	Okt	Dez	Hex	Zch	Okt	Dez	Hex	Zch
000	000	000	...	040	032	020		100	064	040	@	140	096	060	'
001	001	001	...	041	033	021	!	101	065	041	A	141	097	061	a
002	002	002	...	042	034	022	"	102	066	042	B	142	098	062	b
003	003	003	...	043	035	023	#	103	067	043	C	143	099	063	c
004	004	004	...	044	036	024	\$	104	068	044	D	144	100	064	d
005	005	005	...	045	037	025	%	105	069	045	E	145	101	065	e
006	006	006	...	046	038	026	&	106	070	046	F	146	102	066	f
...															
034	028	01C	...	074	060	03C	<	134	092	05C	\	174	124	07C	
035	029	01D	...	075	061	03D	=	135	093	05D]	175	125	07D	}
036	030	01E	...	076	062	03E	>	136	094	05E	^	176	126	07E	~
037	031	01F	...	077	063	03F	?	137	095	05F	_	177	127	07F	...

Lösen Sie zunächst die Grundaufgabe, die darin besteht, eine Ausgabe in Form der Liste 2.2 zu erzeugen. In den einzelnen Zeilen haben nebeneinanderstehende Werte die Differenz 32. Im zweiten Schritt werden die einzelnen Werte durch die oben genannten vier Werte ersetzt und der Tabellenkopf ergänzt.

Liste 2.2: Tabelle mit der Grundstruktur

000	032	064	096
001	033	065	097
002	034	066	098
003	035	067	099
...			
029	061	093	125
030	062	094	126
031	063	095	127

Über die ASCII-Codierung informiert unter anderem die zugehörige *Linux-/Unix*-Handbuchseite (`man ascii`).

2.2. Rundungsfehler durch fortgesetzte Multiplikation/Division 2 Punkte

Schreiben Sie ein Programm, in welchem die **float**-Zahl $x = 10^6$ in einer ersten Schleife 10-mal mit dem **float**-Faktor $k = 0.1693$ multipliziert wird. Das Ergebnis erhaltene wird danach in einer zweiten Schleife 10-mal durch den selben Faktor k dividiert. Theoretisch müßte danach x wieder den Ausgangswert 10^6 enthalten.

Geben Sie den Ausgangswert und den Endwert von x mit jeweils acht Nachkommastellen aus und vergleichen Sie die Werte.

Ausgangswert: _____

Endwert: _____

2.3. Steuerung einer for-Schleife

4 Punkte

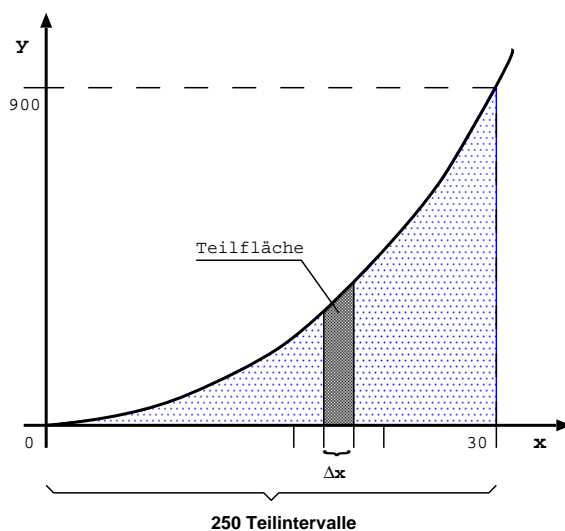


Abbildung 2.1.: Fläche unter einer Parabel

Für die Funktion $f(x) = x^2$ soll im Bereich von $x = 0 \dots 30$ die Fläche zwischen der Funktionskurve und der x-Achse durch eine Annäherung bestimmt werden (Abbildung 2.1). Dazu wird das x -Intervall $[0, 30]$ in 250 Teilintervalle der Länge $\Delta x = 30/250 = 0.12$ eingeteilt und die Funktion an den 251 Teilungspunkten x_0 bis x_{250} ausgewertet.

Die Gesamtfläche F wird dabei als Summe der 250 Teilflächen F_1 bis F_{250} berechnet. Eine Teilfläche F_i wird aus ihrem linken und ihrem rechten Funktionswert gemäß der Formel

$$F_i = \frac{1}{2} \cdot (f(x_{i-1}) + f(x_i)) \cdot \Delta x$$

bestimmt (Trapezfläche).

Die Teilungspunkte x_i und die dazugehörigen Funktionswerte $f(x_i)$ sollen in einer **for**-Schleife berechnet werden. Dazu wird ein ganzzahliger Schleifenzähler **i** verwendet. Das laufende x_i wird durch Multiplikation des Schleifenzählers mit der Intervalllänge ermittelt: $x_i = i * \Delta x$. dadurch hängt der aktuelle Wert von x_i nur von einer Multiplikation ab.

Die durch Integralrechnung ermittelte, exakte Fläche unter der Kurve beträgt 9000 Einheiten. Da es sich bei der hier verwendeten Berechnung um eine Näherung handelt, wird dieser Wert bis auf einen kleinen systematischen Fehler erreicht.

Schreiben Sie ein Programm für diese Flächenberechnung und geben Sie den ersten und den letzten verwendeten x -Wert und die errechnete Fläche mit jeweils zehn Stellen nach dem Komma aus. Verwenden Sie zur Berechnung die Funktion **f**:

```
double f ( double x )
{
    return x*x;
} // ----- end of function f -----
```

3. Verwendung von Feldern

3.1. Feld mit Zufallszahlen belegen – Kennwerte ermitteln 2 Punkte

Schreiben Sie ein Programm, in welchem ein **double**-Feld **a** (100 Elemente) mit Zufallszahlen zwischen 0 und 50 (jeweils einschließlich) belegt wird. Die Funktion **rand()**, **include**-Datei **cstdlib**, liefert Zufallszahlen im Bereich von 0 bis $2^{31} - 1$. Durch die Bildung des Divisionsrestes mit 51 (**C++**-Operator **%**, modulo-Operator) können daraus Werte zwischen 0 und 50 erzeugt werden:

```
a[i] = rand() % 51;
```

- Geben Sie das so belegte Feld mit 10 Werten pro Zeile aus.
- Bilden Sie das arithmetische Mittel aller Feldelemente und geben Sie diesen Wert aus.
- Ermitteln Sie den kleinsten Wert (Minimum) und den größten Wert (Maximum) der Feldelemente und geben Sie diese beiden Werte aus.

Vorgehensweise: die Variablen, die die beiden Extremwerte aufnehmen sollen, werden mit dem ersten Wert der zu untersuchenden Menge initialisiert. Dann erfolgt ein fortlaufender Vergleich mit allen anderen Werten der Menge. Wird ein größerer beziehungsweise kleinerer Wert gefunden, dann wird dieser als neuer Extremwert genommen.

Führen Sie die Berechnung in einem ersten Versuch zunächst mit drei oder vier Zahlen durch und kontrollieren Sie die Ergebnisse!

3.2. Kennwerte einer Zahlenreihe aus einer Datei 2 Punkte

Schreiben Sie auf der Basis des Programmes aus dem vorhergehenden Abschnitt 3.1 ein Programm, welches eine unbekannte Anzahl von Gleitkommazahlen aus der Datei **daten-3-2.txt** einliest (siehe Skript, Kapitel 1).

Die Anzahl der eingelesenen Werte (ein Wert pro Zeile) wird in der Einleseschleife ermittelt und ausgegeben. Geben Sie am Programmende den kleinsten Wert, den größten Wert und das arithmetische Mittel aller Werte aus.

3.3. Korrespondierende Felder 3 Punkte

Schreiben Sie ein Programm, in welchem die **double**-Felder **x**, **y1** und **y2** mit jeweils 1000 Elementen angelegt werden. Die Funktionen $y_1(x) = \sin(x)$ und $y_2(x) = \cos(x)$ sind im Bereich von $x = 0 \dots 6.4$ auszuwerten (**include**-Datei **cmath**). Dazu wird das Intervall $[0, 6.4]$ in 100 Teilintervalle der Länge $\Delta x = 6.4/100$ eingeteilt und die beiden Funktionen an den 101 Teilungspunkten ausgewertet (siehe Aufgabe 2.3). Die x -Werte werden fortlaufen im Feld **x**, die Funktionswerte entsprechend in den Feldern **y1** und **y2** abgelegt.

Nachdem die Felder belegt sind, sollen die Werte zeilenweise in der Form

```

+0.000000 +0.000000 +1.000000
+0.064000 +0.063956 +0.997953
+0.128000 +0.127651 +0.991819
. . .

```

in die Datei `kfeld.dat` Datei ausgegeben werden. Überprüfen Sie, ob alle Werte vorhanden und richtig berechnet sind!

3.4. Erzeugung einer Graphik mit Hilfe von **gnuplot**

3 Punkte

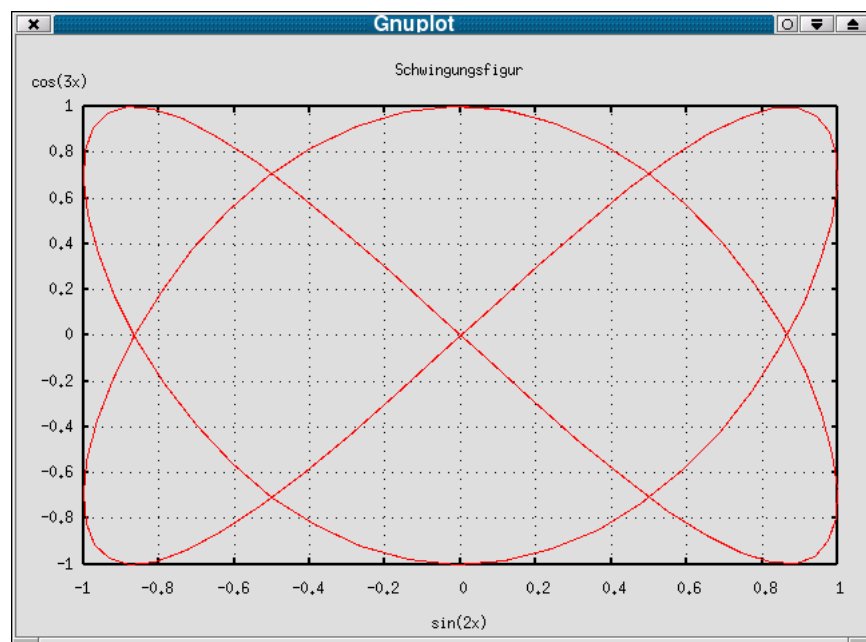


Abbildung 3.1.: Darstellung einer Schwingungsfigur durch das Programm **gnuplot** ($\cos(3x)$ über $\sin(2x)$)

Erstellen Sie nun mit dem Editor eine Datei mit dem Namen `kfeld.gpi` in der folgende Steueranweisungen für das Programm **gnuplot** enthalten sind:

Liste 3.1: Steueranweisungen für das Programm **gnuplot**

```

1 set title "Schwingungsfigur"
2 set grid
3 set nokey
4 set xlabel "sin(2x)"
5 set ylabel "cos(3x)"
6 plot "kfeld.dat" using 2:3 with lines
7 pause -1 "... weiter"

```


Zeile	Erklärung
1	Diagrammtitel festlegen
2	Gitter erzeugen
3	Legende nicht darstellen
4	Beschriftung der x-Achse
5	Beschriftung der y-Achse
6	Kurvenausgabe: aus Datei kfeld.dat Spalte 3 über Spalte 2 auftragen; Punkte durch Linien verbinden
7	Text "... weiter" ausgeben und auf beliebigen Tastenanschlag warten

Nachdem Sie **kfeld.gpi** abgespeichert haben, rufen Sie das Programm **gnuplot** mit folgender Kommandozeile in einem shell-Fenster auf:

```
student@rechner5:/home/student>gnuplot kfeld.gpi
```

Danach erscheint die graphische Ausgabe in einem neuen Fenster. Die Ausgabe wird beendet, wenn der Mauszeiger über dem shell-Fenster steht und die Eingabetaste gedrückt wird.

Berechnen Sie nun

$$y_1(x) = \sin(x) \qquad y_2(x) = \cos(x) \qquad (3.1)$$

$$y_1(x) = \sin(x) \qquad y_2(x) = \cos(3x) \qquad (3.2)$$

$$y_1(x) = \sin(2x) \qquad y_2(x) = \cos(3x) \qquad (3.3)$$

$$y_1(x) = x \cdot \sin(x) \qquad y_2(x) = x \cdot \cos(x) \qquad (3.4)$$

$$y_1(x) = x \cdot \sin(2x) \qquad y_2(x) = x \cdot \cos(3x) \qquad (3.5)$$

Das Skript **kfeld.gpi** erzeugt aus den Gleichungen 3.3 die Darstellung in Abbildung 3.1. Um die Funktion $y_1(x)$ über x aufzutragen muß die Zeile 6 in der Datei **kfeld.gpi** wie folgt aussehen:

```
plot "kfeld.dat" using 1:2 with lines
```

Für die Darstellung von $y_2(x)$ über x lautet diese Zeile

```
plot "kfeld.dat" using 1:3 with lines
```

Öffnen Sie nun die Datei **kfeld.dat** in Ihrem Programm (Vorlesungsskript, Abschnitt 1.9). Die Daten sollen nicht mehr auf die Konsole, sondern unmittelbar in diese Datei geschrieben werden. Nach der **close**-Anweisung, die die Ausgabedatei schließt, fügen Sie nun die folgenden Zeilen ein, die das Programm **gnuplot** direkt aufruft:

```
#include <cstdlib>
...
system("gnuplot kfeld.gpi");
```

gnuplot wird dann unmittelbar aus dem eigenen Programm gestartet; der Umweg über die Kommandozeile entfällt damit. Zur Abgabe der Aufgabe soll die Funktion in Abbildung 3.2 implementiert sein.

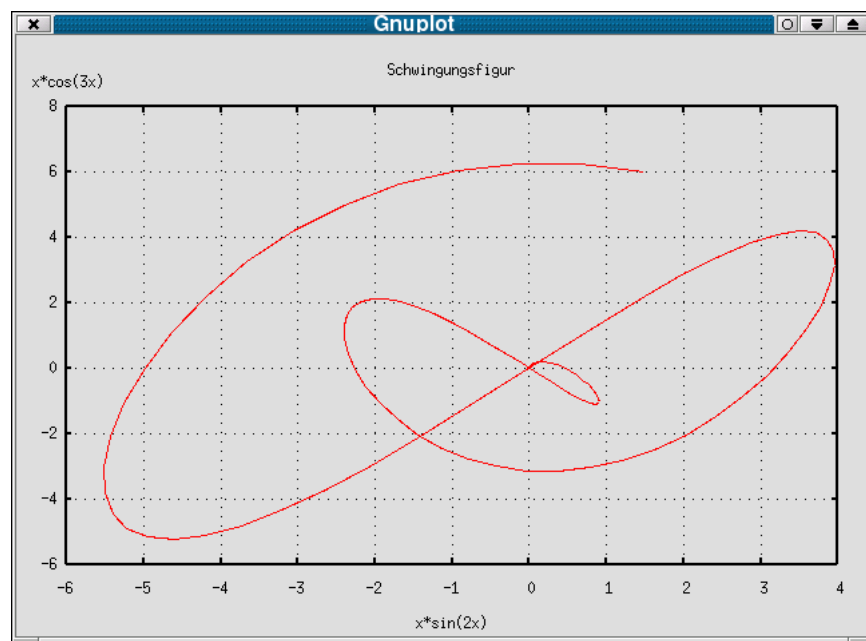
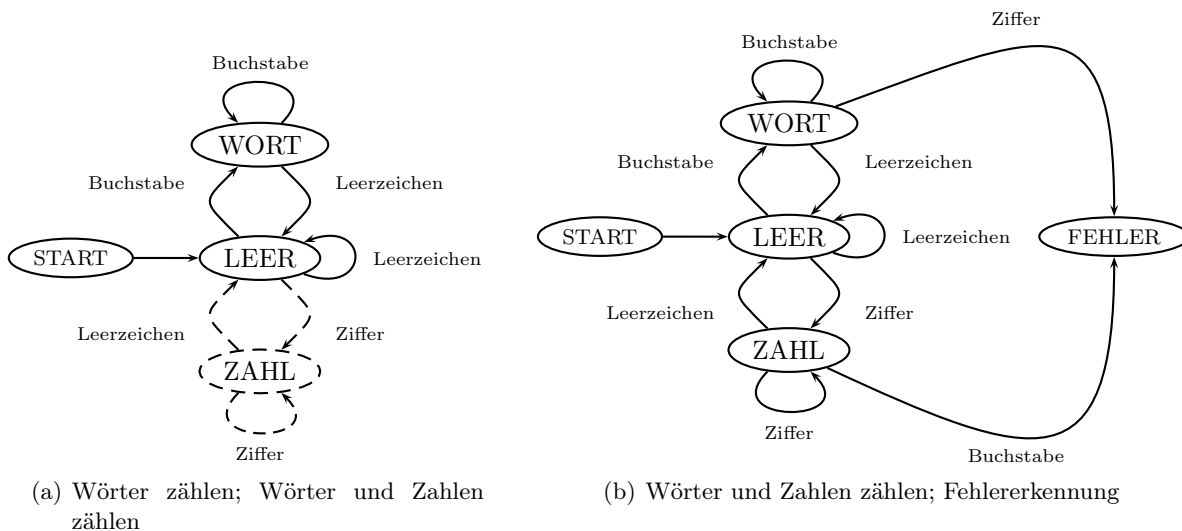


Abbildung 3.2.: Schwingungsfigur ($x \cdot \cos(3x)$ über $x \cdot \sin(2x)$)

4. Umgang mit Texten

4.1. Buchstaben und Ziffern in einem Text abzählen

3 Punkte



Abbildungung 4.1.: Zustandsdiagramme zur Wort- und Zahlenerkennung

- Geben Sie das Beispielprogramm 1.11.string.cc aus der Vorlesung ein, in welchem ein Text eingelesen, ausgegeben und die Anzahl der Klein- und Großbuchstaben ermittelt wird. Überprüfen Sie dessen Funktionsweise!
- Erweitern Sie das Programm so, daß auch Ziffern abgezählt werden können (Testfunktion: `isdigit()`).
- Ändern Sie das Programm nun so, daß die Unterscheidung zwischen Klein- und Großbuchstaben entfällt und nur Buchstaben (Testfunktion: `isalpha()`) und Ziffern abgezählt werden.

4.2. Wörter in einem Text abzählen

3 Punkte

Das unter 4.1 entwickelte Programm soll nun so abgeändert werden, daß die **Wörter** im Eingabetext abgezählt und deren Anzahl ausgegeben werden.

Die Analyse kann wie folgt durchgeführt werden: Das augenblicklich betrachtete Zeichen gehört entweder zu einem Wort oder zu einem Zwischenraum (Folge von Leerzeichen und/oder Tabulatoren). Jeder der beiden Zustände wird in einer Zustandsvariablen `zustand` codiert, zum Beispiel LEER: `zustand=0`, WORT: `zustand=1`. In jedem Schritt (das heißt beim jeweils nächsten zu betrachtenden Zeichen) wird überprüft, ob eine Zustandsänderung stattfindet (siehe Abbildung 4.1). Beim Zustandsübergang LEER → WORT wird der Wortzähler erhöht.

Es gibt zwar mehrere Zustände, die Implementierung benötigt jedoch nur eine einzige Zustandsvariable, da die Zustände numeriert sind.

Zur Klassifizierung der einzelnen Zeichen können die Klassifizierungsfunktionen verwendet werden, die in der header-Datei `cctype` definiert sind (siehe auch Abb 4.2).

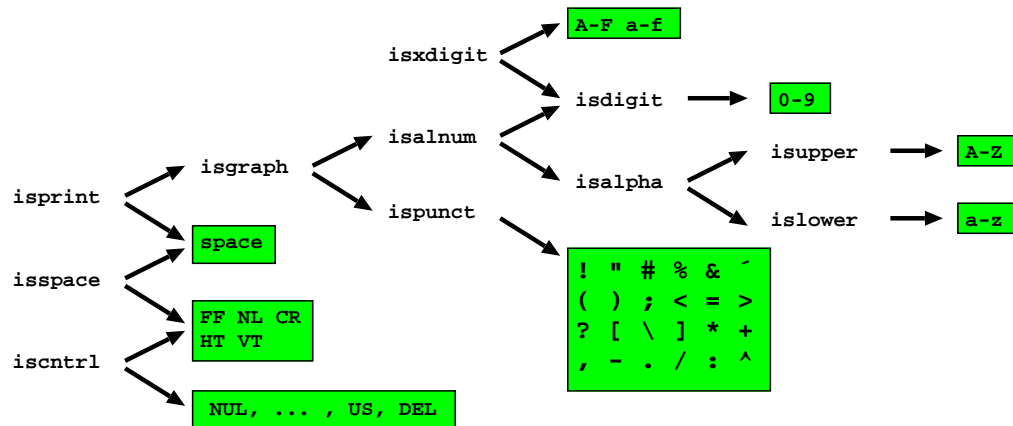


Abbildung 4.2.: Hierarchie der Zeichenklassen und der Klassifizierungsfunktionen in *C/C++* (header-Datei `cctype`)

4.3. Wörter und Zahlen in einem Text abzählen

4 Punkte

Das unter 4.2 entwickelte Programm soll nun so abgeändert werden, daß zusätzlich Zahlen im Eingabetext erkannt, abgezählt und auch deren Anzahl ausgegeben wird. Dazu ist gemäß Abbildung 4.1a ein weiterer Zustand ZAHN einzufügen und die zusätzlichen Übergänge zu implementieren.

Erweitern Sie das Programm indem Sie prüfen, ob in einer Buchstabenfolge eine Ziffer oder in einer Ziffernfolge ein Buchstabe auftritt (Abbildung 4.1b). In diesem Fall soll das Programm eine Fehlermeldung ausgeben und über den Funktionsaufruf `exit(1)` (`include`-Datei `cstdlib`) verlassen werden. Erstellen Sie nun in einem Editor eine mehrzeilige Textdatei `text.txt`, die eine größere Anzahl von Wörtern und Zahlen (evtl. aus einer anderen Textdatei kopiert). Überprüfen Sie das Ergebnis ihrer Auszählung!

Ersetzen Sie nun die Tastatureingabe durch das zeichenweise Einlesen des zu untersuchenden Textes aus der Datei `text.txt` (siehe letzter Abschnitt im 1. Kapitel des Vorlesungsskriptes).

```

i = 0;                                // Schleifenzähler
while ( i < maxstring-1 &&             // Pufferlänge überwachen
        ( text[i] = ifs.get() ) != EOF ) // Einlesen bis zum Dateende
    i = i+1;
text[i] = '\0';                       // Zeichenkette abschließen
  
```

Die `const int`-Variable `maxstring` enthält die Länge des Eingabepuffers `text`; `ifs` ist der Name des Eingabestromes. Prüfen Sie anschließend Ihr Ergebnis, indem Sie die Textdatei mit dem *Unix*-Systemprogramm `wc` (steht für word count) von der Kommandozeile aus auszählen:

```
.../home/student>wc -w text.txt
```

Der Kommandozeilenschalter `-w` sorgt dafür, daß nur Wörter gezählt werden. Die ausgegebene Zahl muß der Summe der Wörter und Zahlen in der Ausgabe des eigenen Programmes entsprechen.

5. Codieren und Decodieren von Texten

5.1. Die Verschiebechiffre ROT13

4 Punkte

Klartext	A	B	C	...	K	L	M	N	O	P	...	X	Y	Z
Verschlüsselung	N	O	P	...	X	Y	Z	A	B	C	...	K	L	M

Abbildung 5.1.: ROT13 ersetzt jeden Buchstaben des Klartextes durch den darunterstehenden Buchstaben. Das untere Alphabet ist gegenüber dem oberen um 13 Stellen zyklisch verschoben.

ROT13 („rotiere das Alphabet um 13 Stellen“) ist eine Verschiebechiffre, bei der die 26 Buchstaben des Alphabets um 13 Stellen zyklisch verschoben werden. Abbildung 5.1 zeigt das Prinzip: die Buchstaben des Klartextes (oben) werden jeweils durch Buchstaben des um 13 Plätze zyklisch verschoben Alphabets (unten) ersetzt. Aus dem Klartext „HALLO“ wird der verschlüsselte Text „UNYYB“. Sonderzeichen und Ziffern werden in dieser einfachen Form nicht berücksichtigt. Die Verschlüsselung ist offensichtlich so schwach, daß das Verfahren nur dazu verwendet werden kann, einen Text vorübergehend unleserlich zu machen. Wird ROT13 auf den verschlüsselten Text angewendet, dann wird der Klartext wieder hergestellt.

Erstellen Sie zunächst eine Datei `klartext.txt` mit folgendem Inhalt:

```
DIES IST EIN GEHEIMTEXT MIT 43 ZEICHEN!
QVRF VFG RVA TRURVZGRKG ZVG 43 MRVPURA!
```

Nun erstellen Sie ein `C++`-Programm, das den Text aus der Datei `klar.txt` einliest (Eingabestrom `ifs`) und in eine Ausgabedatei `geheim.txt` ausgibt (Ausgabestrom `ofs`). Das Einlesen kann mittels folgender Anweisung bewerkstelligt werden:

```
while ( ( c = ifs.get() ) != EOF ) {
    /*
        ...
    */
}
```

Die Ausgabe in die Datei `geheim.txt` erfolgt einfach durch Ausgabe der Variablen `c` in den Ausgabestrom `ofs` durch `ofs « c;`.

Zur Verschlüsselung wird das folgende Feld verwendet:

```
char rot13[] = {
    'N','O','P','Q','R','S','T','U','V','W','X','Y','Z',
    'A','B','C','D','E','F','G','H','I','J','K','L','M',
};
```

Die Codierung für A ist N und steht im Feldelement mit Index 0. Die Codierung für Z ist M und steht im Feldelement mit Index 25. Steht der zu codierende Buchstabe in der Variablen `c`, dann errechnet man durch `c - 'A'` den Index im Feld `rot13` für dessen Codierung.

5.2. Caesar-Verschlüsselung

6 Punkte

ROT13 ist ein Sonderfall der Caesar-Verschlüsselung, die auf Gaius Iulius Caesar zurückgeht. Die Caesar-Verschlüsselung verwendet eine beliebige Verschiebung des Alphabets (25 Möglichkeiten, eine Verschiebung um 26 Positionen verändert das Alphabet nicht). Um diesen allgemeineren Fall zu verwenden, muß das Feld, das zum Verschlüsseln verwendet wird um n Positionen (zum Beispiel um 5 Positionen) zyklisch verschoben werden. Das kann wie im Aufgabenteil 5.1 durch eine feste Feldbelegung erreicht werden. Eine weitere Möglichkeit benutzt ein Feld, das zunächst mit dem Alphabet in der üblichen Reihenfolge belegt und vor Beginn der Verschlüsselung um die gewünschten n Positionen verschoben wird. Hier soll die letzte Möglichkeit verwendet werden. Um den Inhalt eines Feldes um n Positionen zyklisch nach links zu rotieren, kann so vorgegangen werden, wie das Beispiel in Abbildung 5.2 für den Wert $n = 5$ und die Feldlänge 26 zeigt.

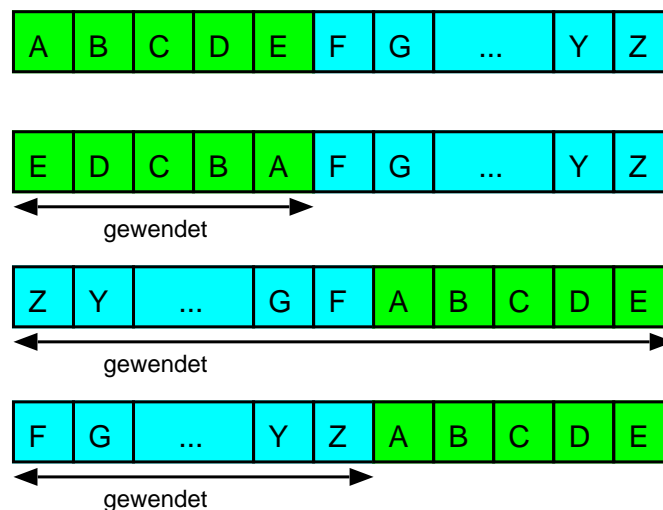


Abbildung 5.2.: Verschiebung eines Feldinhaltes um $n = 5$ Positionen nach links

Zunächst wird die Reihenfolge der ersten $n = 5$ Elemente im Feld umgekehrt (gewendet). Anschließend wird das gesamte Feld gewendet. Zum Abschluß werden die restlichen $26 - 5$ Elemente gewendet. Danach ist der Feldinhalt um $n = 5$ Plätze zyklisch nach links verschoben. Diese Vorgehensweise hat den Vorteil, daß die Teile des Feldes, die gewendet werden müssen, stets beim Index 0 beginnen. Das vereinfacht die Verwendung der Funktion `feld_wenden` (siehe unten).

Liste 5.1: Prototypen zu Aufgabe 5.2

```
void feld_wenden      ( char a[], unsigned int n );
void feld_links_rotieren ( char a[], unsigned int n, unsigned int shift );
void feld_rechts_rotieren ( char a[], unsigned int n, unsigned int shift );
```

1. Schreiben Sie zunächst eine **Funktion** `feld_wenden` (Prototyp in Liste 5.1), die ein `char`-Feld `a` der Länge `n` in sich wendet. Ein Hilfsfeld darf nicht verwendet werden.

2. Schreiben Sie dann eine **Funktion** `feld_links_rotieren` (Prototyp in Liste 5.1), die ein **char**-Feld `a` der Länge `n` um `shift` Plätze zyklisch nach links verschiebt. Diese Funktion soll durch drei Aufrufe der Funktion `feld_wenden` implementiert werden.
3. Schreiben Sie dann eine **Funktion** `feld_rechts_rotieren` (Prototyp in Liste 5.1), die ein **char**-Feld `a` der Länge `n` um `shift` Plätze zyklisch nach rechts verschiebt.

Für die beiden Funktionen `feld_links_rotieren` und `feld_rechts_rotieren` gilt: eine Verschiebung um 0 Plätze ist zulässig. Die Funktionen werden dann sofort wieder verlassen. Weiterhin muß die Verschiebung (Parameter `shift`) kleiner als die Feldlänge sein, um Zugriffe außerhalb des Feldes durch die Funktion `feld_wenden` zu vermeiden. Dazu müssen alle Vielfache der Feldlänge entfernt werden. Beispiel: bei einer Feldlänge von $n = 10$ entspricht eine zyklische Verschiebung um 33 Plätze einer tatsächlichen Verschiebung um 3 Plätze.

Diese Funktionen werden dann dazu verwendet, die gewünschte Anfangsbelegung des der Verschlüsselungsfeldes für eine Caesar-Verschlüsselung aus der Anfangsbelegung A bis Z herzustellen.

Ein Text, der mit einer Verschiebung n nach links verschlüsselt wurde, muß mit einer Feld mit einer Verschiebung $-n$ entschlüsselt werden (also mit einer Rechtsverschiebung um n Plätze).

5.3. Rotation durch Indexrechnung *freiwillig*

Eine weitere Lösung kommt ohne Hilfsfeld aus, wenn der Zielbuchstabe berechnet wird. Bei Beschränkung auf die bisher verwendeten 26 Großbuchstaben, kann der Zielbuchstabe einer Caesar-Verschlüsselung auch wie folgt berechnet werden:

```
cout << (char)( 'A' + ( c - 'A' + n )%26 );
```

Dabei ist `c` der Buchstabenwert aus dem Klartext und `n` die Verschiebung. Für $n = 13$ ergibt sich die ROT13-Verschlüsselung.

- Machen Sie sich zunächst klar, warum diese Berechnung das richtige Ergebnis liefert.
- Erweitern Sie ihr Programm so, daß Kleinbuchstaben und Ziffern ebenfalls verschlüsselt werden.

6. Einfache statistische Kennwerte

6.1. Statistische Kennwerte

5 Punkte

Folgende Kennwerte einer Zahlenfolge x_1, \dots, x_n sind zu berechnen:

Kennwert	Berechnungsvorschrift
Minimum	kleinster Wert der Zahlenfolge
Maximum	größter Wert der Zahlenfolge
Spanne	Maximum - Minimum
Arithmetischer Mittelwert	$am = \frac{1}{n} \sum_{i=1}^n x_i$
Geometrischer Mittelwert	$gm = \sqrt[n]{x_1 x_2 \cdots x_n}$ <p>Zur Berechnung siehe Abschnitt 6.3.</p>
Quadratischer Mittelwert	$qm = \sqrt{\frac{1}{n} \sum_{i=1}^n x_i^2}$
Mittlere Abweichung	$ma = \frac{1}{n} \sum_{i=1}^n \ x_i - am\ $
Median	<p>Der Median einer Menge von Zahlen, die <i>der Größe nach sortiert</i> sind, ist bei ungerader Anzahl der Wert in der Mitte, bei gerader Anzahl das arithmetische Mittel der beiden Werte in der Mitte.</p> <p>Zur Sortierung siehe Abschnitt 6.2.</p>

Bei der Programmerstellung ist folgendermaßen vorzugehen:

- Verwenden Sie zur Berechnung der Kennwerte jeweils eine eigene Funktion. Weiterhin wird das Füllen, Ausgeben und Sortieren eines des Feldes, sowie der Wertetausch zweier **double**-Variablen jeweils in einer eigenen Funktion dargestellt (siehe unten)
- Es sollen 200 **double**-Werte berücksichtigt werden, die mit Hilfe des Zufallsgenerators **rand()** (Prototyp in **cstdlib**) erzeugt werden können (Wertebereich 0 bis 100; eine Nachkommastelle, die ungleich Null sein kann).
- Zur Berechnung der Quadratwurzel wird die Funktion **sqrt()** verwendet (**cmath**).
- Zur Berechnung des Betrages einer **double**-Größe wird die Funktion **fabs()** verwendet (**cmath**).

- Die nachfolgenden Prototypen sind zu verwenden.

```

void    fuehle_feld          ( double a[], int n, int unten, int oben );
void    gib_feld_aus        ( double a[], int n );
void    tausche_double      ( double *a, double *b );
void    sortiere_feld_aufsteigend ( double a[], int n );
void    sortiere_feld_absteigend ( double a[], int n );
double  minimum             ( double a[], int n );
double  maximum             ( double a[], int n );
double  spanne              ( double a[], int n );
double  arithm_mittelwert   ( double a[], int n );
double  geom_mittelwert     ( double a[], int n );
double  quad_mittelwert     ( double a[], int n );
double  mittlere_abweichung ( double a[], int n );
double  median              ( double a[], int n );

```

- Die Funktion `fuehle_feld` füllt das übergebene Feld mit Zufallswerten die zwischen `unten` und `oben` liegen sollen.
- Die Funktion `gib_feld_aus` gibt das übergebene Feld in Zeilen zu 10 Werten aus.
- Die Funktion `tausche_double` tauscht die Werte der beiden übergebenen Variablen aus.
- Die beiden Sortierfunktionen `sortiere_feld_aufsteigend` und `sortiere_feld_absteigend` verwenden das Verfahren, welches in Abschnitt 6.2 dargestellt ist.
- Die Funktion `median` ruft zur Sortierung eine der beiden Sortierfunktionen auf. Die Daten werden dabei im Originalfeld sortiert, das heißt es soll kein Hilfsfeld angelegt werden.

6.2. Sortieren durch Vertauschen

5 Punkte

Vorgehensweise. Das erste und das zweite Element der vorgelegten Liste werden verglichen. Ist das erste größer als das zweite, so werden die beiden vertauscht. Der gleiche Vorgang wird auf das (nunmehr) zweite und dritte Element der Liste angewandt, dann auf das (nunmehr) dritte und vierte. Dies wird bis zum Vergleich des (n-1)-ten mit dem n-ten Element fortgesetzt. Nach (n-1) Vergleichen befindet sich das größte Element am Ende der Liste.

Nach dem gleichen Verfahren wird nun in der um eine Position verkürzten Liste das zweitgrößte Element auf die vorletzte Position gebracht. Dies wird fortgesetzt an einer immer kürzer werdenden Restliste bis diese schließlich nur noch aus einem Element besteht.

Bei der Implementierung soll die Funktion `tausche_double()` verwendet werden.

Aufwand. Bei einer Liste mit n Elementen sind (n-1) Durchläufe erforderlich. Beim ersten Durchlauf werden (n-1) Vergleiche durchgeführt, beim letzten nur noch ein Vergleich. Der Sortieraufwand A ergibt sich durch das Aufsummieren der anfallenden Vergleichsschritte:

$$A = \sum_{i=1}^{n-1} (n-i) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \approx \frac{n^2}{2}$$

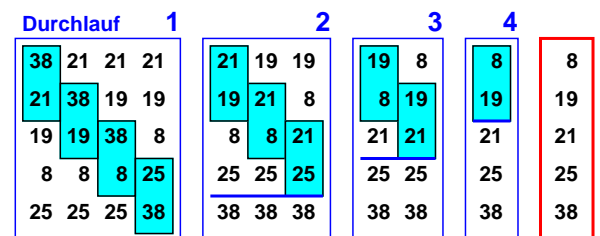


Abbildung 6.1.: Sortieren durch Vertauschen

Vorteil. Das Verfahren hat den Vorteil, daß es keinerlei zusätzlichen Speicher benötigt. Der Sortiervorgang läuft innerhalb der vorgelegten Originalliste ab. Das Verfahren ist außerdem sehr einfach zu implementieren.

Nachteil. Ein schwerwiegender Nachteil ist die Zunahme des Aufwandes mit dem Quadrat der Listenlänge n . Das Verfahren ist daher nur für kurze, selten zu sortierende Listen geeignet.

6.3. Berechnung des geometrischen Mittelwertes

Durch Logarithmieren der Gleichung

$$gm = \sqrt[n]{x_1 x_2 \cdots x_n}$$

mit dem natürlichen Logarithmus \ln (Basis e) erhält man

$$\begin{aligned} GM = \ln gm &= \ln \sqrt[n]{x_1 x_2 \cdots x_n} \\ &= \frac{1}{n} (\ln x_1 + \ln x_2 + \cdots + \ln x_n) \\ &= \frac{1}{n} \sum_{i=1}^n \ln x_i \end{aligned}$$

das heißt man berechnet zunächst den Logarithmus von gm als arithmetisches Mittel der Summe der Logarithmen der Größen x_i . Man erhält nun das gesuchte Ergebnis gm durch Potenzierung:

$$gm = e^{GM} = e^{\ln gm}$$

Natürliche Logarithmus, Exponentialfunktion: `log()`, `exp()`, Prototypen in `cmath`.

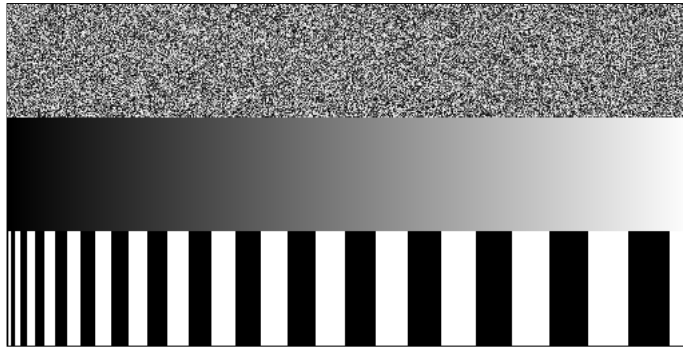
Ergebnisse

Kennwert	Ergebnis (2 Nachkommastellen)
Minimum	
Maximum	
Spanne	
Arithmetischer Mittelwert	
Geometrischer Mittelwert	
Quadratischer Mittelwert	
Mittlere Abweichung	
Median	

7. Bildbearbeitung 1

7.1. Bilddatei lesen und speichern

4 Punkte

Abbildung 7.1.: Grauwertbild in der Datei `dreifach.pgm`

In der Datei `dreifach.pgm` ist ein Grauwertbild (Abbildung 7.1) der Größe 512×256 Bildpunkten gespeichert. Der Dateianfang ist in Liste 7.1 dargestellt (siehe auch Tabelle 7.1). Eine Bildzeile (512 Grauwerte) ist dabei in 32 aufeinanderfolgenden Dateizeilen gespeichert.

Liste 7.1: Beginn der pgm-Datei `dreifach.pgm`

```

1 P2
2 512 256
3 255
4 103 198 105 115 81 255 74 236 41 205 186 171 242 251 227 70
5 124 194 84 248 27 232 231 141 118 90 46 99 51 159 201 154
6 102 50 13 183 49 88 163 90 37 93 5 23 88 233 94 212
7 171 178 205 198 155 180 84 17 14 130 116 65 33 61 220 135
8 112 233 62 161 65 225 252 103 62 1 126 151 234 220 107 150
9 143 56 92 42 236 176 59 251 50 175 60 84 236 24 219 92
10 2 26 254 67 251 250 170 58 251 41 209 230 5 60 124 148
11 117 216 190 97 137 249 92 187 168 153 15 149 177 235 241 179
12 5 239 247 0 233 161 58 229 202 11 203 208 72 71 100 189
13 31 35 30 168 28 123 100 197 20 115 90 197 94 75 121 99
14 59 112 100 36 17 158 9 220 170 212 172 242 27 16 175 59

```

Schreiben Sie ein Programm in welches die Datei `dreifach.pgm` eingelesen wird. Die Bildinformationen (Zeile 1-3) sollen zur Kontrolle ausgegeben werden. Das Bild wird in ein hinreichend großes Feld vom Typ `unsigned char` (bild-)zeilenweise eingelesen (siehe Liste 7.2). Beim Einlesen ist zu überprüfen, ob die Codierung des Dateityps (hier P2) richtig ist und ob das Bild in die bereitgestellte Matrix paßt. Im Fehlerfall ist das Programm mit einer entsprechenden Meldung abzubrechen. Anschließend ist das Bild im selben Dateiformat in eine Datei `dreifach.out.pgm` zu speichern. Hierbei soll überprüft werden, ob das Schreiben in die Ausgabedatei erfolgreich war. Im Fehlerfall ist das

Zeile	Inhalt	Bedeutung
1	P2	Codierung des Dateityps, hier pgm
2	512 256	Bildbreite und Bildhöhe in Pixel
3	255	maximaler Grauwert (0 = schwarz , 255 = weiß)
4 bis Dateiende		Grauwerte der einzelnen Bildpunkte; zeilenweise fortlaufend abgelegt

Tabelle 7.1.: Aufbau der pgm-Datei

Programm mit einer entsprechenden Meldung abzuberechnen.

Das abgespeicherte Bild kann mit den Programmen **xv**, **Gwenviiew** oder ähnlichen Bildbetrachtern dargestellt werden (Sichtvergleich; verwenden Sie die Lupenfunktion, um charakteristische Bildbereiche zur Kontrolle zu vergrößern):

```
xv dreifach.pgm
```

Programm **xv**: Bei Betätigung der rechten Maustaste erscheint ein Dialogfenster. Wenn der Mauszeiger über dem Bild steht, reicht die Eingabe des Buchstabens **q** (für **quit**) um das Programm zu beenden.

Liste 7.2: Prototypen der Bildbearbeitungsfunktionen

```

1 // #####  VARIABLES  -  LOCAL TO THIS SOURCE FILE  #####
2 const int N = 1000;                                // max. Bildhöhe und Bildbreite
3
4 // #####  TYPE DEFINITIONS  #####
5 typedef unsigned char Pixel;                       // Datentyp Pixel
6
7 // #####  LOCAL PROTOTYPES  #####
8 void glaetten ( Pixel bild1[N][N], Pixel bild2[N][N], int nz, int ns, int graumax );
9 void invertieren ( Pixel bild1[N][N], Pixel bild2[N][N], int nz, int ns, int graumax );
10 void kantenbildung( Pixel bild1[N][N], Pixel bild2[N][N], int nz, int ns, int graumax );
11 void schwellwert ( Pixel bild1[N][N], Pixel bild2[N][N], int nz, int ns, int graumax,
12                  Pixel schwellwert );

```

7.2. Grauwertbild bearbeiten

6 Punkte

Im zweiten Schritt soll durch die Anwendung von Bearbeitungsschritten aus dem Originalbild ein Ergebnisbild erzeugt werden. Erstellen Sie eine weitere Version des Programmes, in welcher die Grauwertbilder bearbeitet werden. Alle Bildverarbeitungsfunktionen werden als eigene Funktion dargestellt (Prototypen in Liste 7.2). Die Parameter **nz** und **ns** geben die Zeilen- und Spaltenanzahl der Bilder in den festen Pixel-Feldern an.

Legen Sie ein weiteres Feld zur Aufnahme des bearbeiteten Bildes an. Die unten beschriebenen Bearbeitungsfunktionen sind immer so anzuwenden, daß die zu verarbeitende Pixel-Information aus dem eingelesenen und unveränderten Originalbild genommen wird, die daraus erzeugten Pixel im Ergebnisbild abgelegt werden.

Geben Sie zur Steuerung Ihres Programmes ein kleines Menü aus (Abbildung 7.2) und lesen Sie einen Buchstaben zur Auswahl der Bearbeitungsfunktion ein. Mit diesem Buchstaben wird eine **switch**-Anweisung gesteuert, in welcher die Bildoperationen mittels Funktionsaufruf durchgeführt werden.

Invertierung. Bei der Invertierung wird der neue Grauwert b_{ij} aus dem ursprünglichen Grauwert a_{ij} wie folgt berechnet:

$$b_{ij} = \text{max.Grauwert} - a_{ij}$$

Es entsteht ein Negativbild: Schwarz wird zu Weiß und umgekehrt. Kontrollieren Sie das Ergebnis mit einem Bildbetrachter.

Schwellwertoperation. Alle Bildpunkte, die im Originalbild einen Grauwert kleiner 150 besitzen, werden im Ergebnisbild zu Null gesetzt. Alle anderen Bildpunkte werden auf den maximalen Grauwert (weiß) gesetzt.

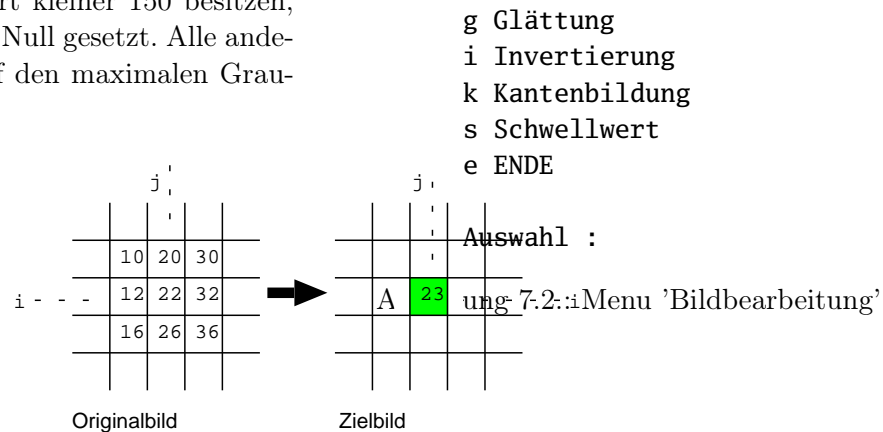


Abbildung 7.3.: Glättung durch Mittelwertbildung über 9 Bildpunkte

Glättung. Bei der Glättung entsteht das Zielpixel durch Mittelwertbildung über das Originalpixel und dessen 8 Umgebungspixel (Abbildung 7.3). Durch die Glättung werden harte Kontraste gemildert.

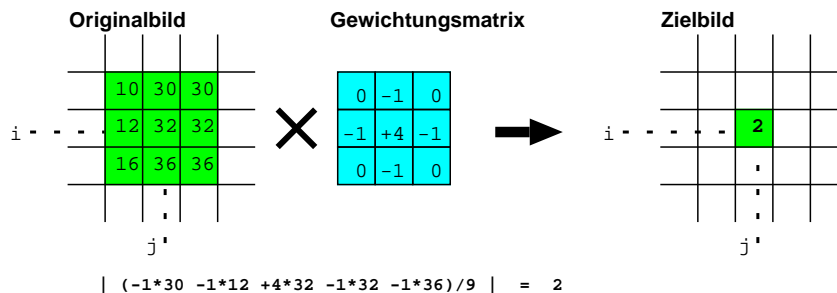


Abbildung 7.4.: Kantenerkennung durch Verknüpfung einer 3x3-Umgebung des Originalbildes mit einer Gewichtungsmatrix

Kantenerkennung. Objektkanten sind dort zu vermuten, wo sprungartige Helligkeitsübergänge vorhanden sind. Um Kanten rechnerisch zu ermitteln wird über das Bild ein sogenanntes Operatorfenster geschoben. Der aktuelle Bildpunkt und die 8 Nachbarpunkte werden mit den Gewichtungen in Abbildung 7.4 multipliziert, addiert und zur Normierung durch 9 geteilt. Der **Betrag** dieses Ergebnisses ist der Wert des Zielbildpunktes. Wie bei der Glättung kann diese Operation nur auf die inneren Punkte des Originals angewendet werden. Zur Bildung des Betrages steht die Bibliotheksfunktion **abs** zur Verfügung.

Um bei der Glättung und der Kantenerkennung im Zielbild die ursprüngliche Bildgröße beizubehalten, werden bei der Glättung die Randpunkte des Originalbildes in das Zielbild übernommen, bei der Kantenerkennung wird der Rand schwarz gesetzt.

Ergebnisse der Bildbearbeitung

Überprüfen Sie zum Abschluß die richtige Funktionsweise der Programme mit Hilfe der vorgegebenen Bilder. In Abbildung 7.5 sind die Ergebnisse der vier geforderten Bearbeitungen dargestellt.

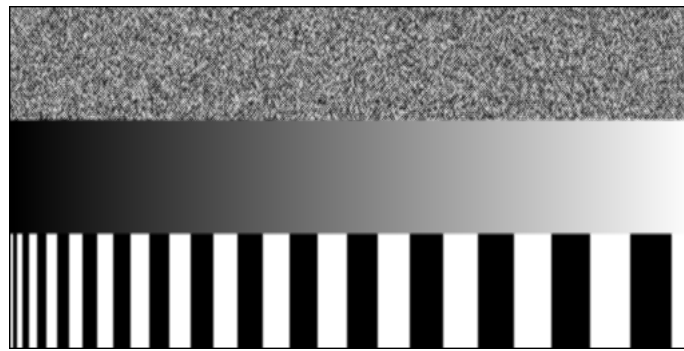
Abbildung 7.5a zeigt das Ergebnis der Glättung. Die harten Kontraste (große Grauwertübergänge) sind abgemildert. Das bearbeitete Bild wirkt, besonders im oberen Drittel, gegenüber dem Original etwas verwaschen. Im unteren Drittel haben die Schwarz-Weiß-Übergänge zwei Graustufen.

In Abbildung 7.5c ist das Ergebnis der Kantenbildung dargestellt (dieses Ergebnisbild ist aus drucktechnischen Gründen invertiert). Kanten werden nur bei den großen Grauwertänderungen (Übergänge von Weiß nach Schwarz) erkannt. Sie erscheinen im Ergebnisbild als dünne graue Linien. Im oberen Drittel ist die Kantenbildung offensichtlich sinnlos. Im mittleren Drittel werden keine Kanten gefunden, weil der Grauwertübergang gleichmäßig von dunkel nach hell verläuft. Im unteren Drittel werden die Schwarz-Weiß-Übergänge als Kanten erkannt.

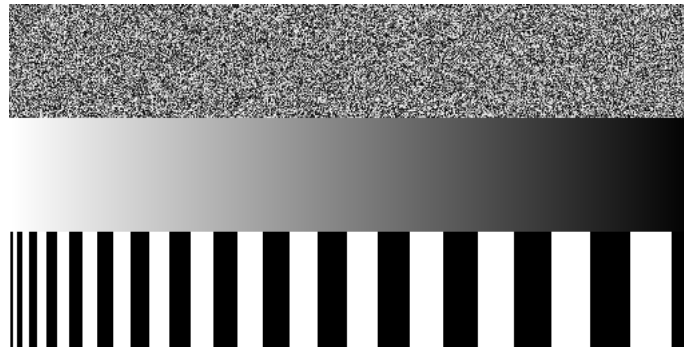
Bei der Schwellwertoperation in Abbildung 7.5d werden alle Bildpunkte mit Grauwerten kleiner als 150 schwarz dargestellt.

Die Abbildung 7.5 kann zum Sichtvergleich der eigenen Ergebnisse herangezogen werden. Die Verwendung von Bildbetrachtungsprogrammen ermöglicht unter anderem die Untersuchung an vergrößerten Darstellungen. Ein Sichtvergleich reicht jedoch keinesfalls zur Überprüfung!

Die richtige Behandlung der Bildränder, die vollständige Bearbeitung aller Punkte (Schleifen!) und die richtige Arbeitsweise der einzelnen Operationen muß stichprobenartig mit Hilfe der Zahlenwerte einzelner Bildpunkte überprüft werden. Die Ergebnisbilder können hierzu in den Editor geladen und untersucht werden.



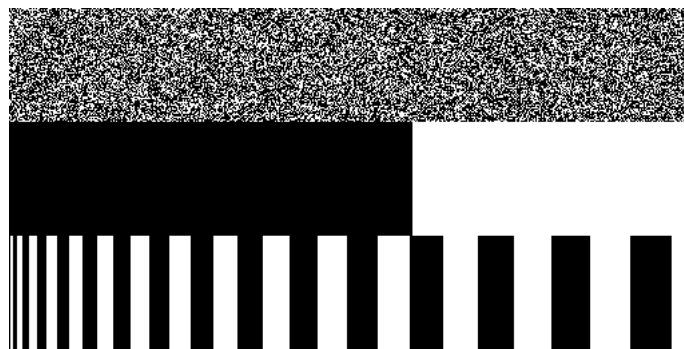
(a) Glättung



(b) Invertierung



(c) Kantenbildung (invertierte Darstellung)



(d) Schwellwert

Abbildung 7.5.: Ergebnisse der Bildbearbeitung (Original: `dreifach.pgm`)

7.3. Bilderzeugung *freiwillig*

Da der bisher verwendete Bildtyp als Zahlenmatrix dargestellt wird, können ohne weiteres beliebige Bilder in einem Programm als Matrix erzeugt werden. Abbildung 7.6 zeigt ein Bild der Größe 256×512 , bei dem sich schwarze und weiße Streifen der Breite 8 Pixel abwechseln. In Abbildung 7.7 ist die erste Zeile der Bildmatrix wiedergegeben.

Die Berechnung der einzelnen Bildpunkte kann sehr einfach erfolgen, wenn folgender Zusammenhang ausgenutzt wird. Wenn ein Spaltenindex ganzzahlig durch die Streifenbreite geteilt wird, entsteht die Streifennummer. Die Zählung der Streifennummern beginnt mit 0. Index 15 ganzzahlig geteilt durch 8 ergibt die Streifennummer 1, Index 17 ganzzahlig geteilt durch 8 ergibt die Streifennummer 2 und so weiter. Alle Bildpunkte mit geradzahligem Streifennummer erhalten den Grauwert schwarz, alle Bildpunkte mit ungerader Streifennummer erhalten den Grauwert weiß.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25		
0	0	0	0	0	0	0	0	0	x	x	x	x	x	x	x	0	0	0	0	0	0	0	0	x	x	...	0

Abbildung 7.7.: Indexrechnung bei der Bilderzeugung ('0' schwarz, 'x' weiß)

Erweitern Sie die oben dargestellte Vorgehensweise so, daß ein Schachbrettmuster (Abbildung 7.8) erzeugt wird. Die Felder sollen die Größe 8×8 besitzen. Hierzu folgender Hinweis: bei diesem Muster ist die Summe der jeweiligen Nummern des senkrechten und des waagerechten Streifens für die schwarzen Felder immer gerade, für die weißen immer ungerade.

Erweitern Sie die letzte Lösung so, daß ein Rechteckmuster erzeugt wird. Die Rechtecke sollen beliebige Seitenlängen größer Null haben können.

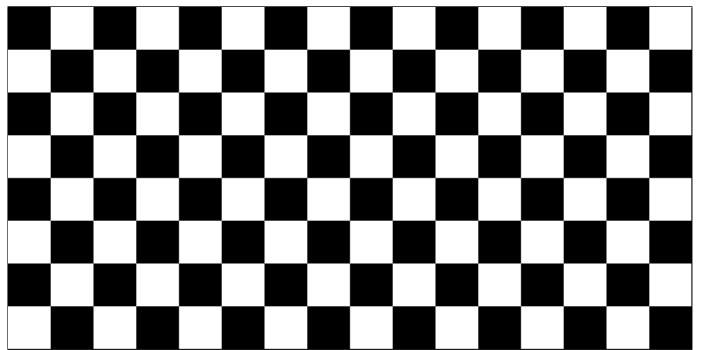


Abbildung 7.8.: Schachbrett

8. Bildbearbeitung 2 / Mengen

8.1. Bildbereich füllen

4 Punkte

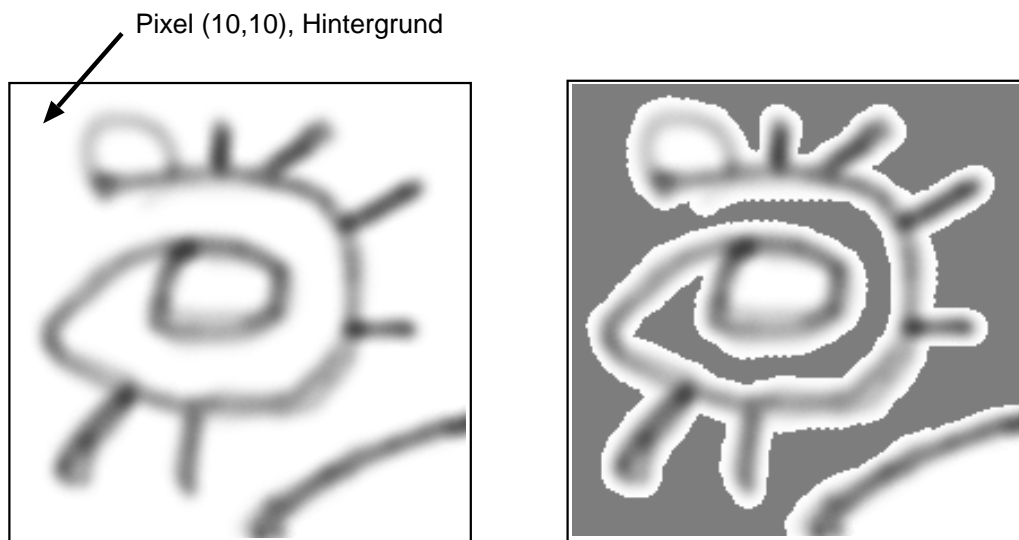


Abbildung 8.1.: Rekursives Füllen eines Bildbereiches

Bei Grauwertbildern sollen zusammenhängende Bereiche mit einem neuen Grauwert versehen werden. Dazu wird ein Impfpixel auf einen neuen Wert gesetzt und rekursiv alle Nachbarpixel untersucht. Wenn ein Nachbarpixel denselben Originalgrauwert besitzt, dann erhält es ebenfalls den neuen Wert und seine Nachbarn müssen auch untersucht werden. Beachten Sie, daß jeder innere Bildpunkt 4 echte Nachbarn besitzt (Randpunkte entsprechend weniger!).

In Abbildung 8.1, links, wird ein Hintergrundpixel mit einem neuen Grauwert „geimpft“. Im rechten Bild haben alle Hintergrundpixel rekursiv den neuen Grauwert erhalten. Als Bilddatei soll **kunst.pgm** verwendet werden. Erweitern Sie die Lösung aus Aufgabe 7 um eine Funktion **fill**. Die zu erstellende Funktion **fill** besitzt die in Liste 8.1 gezeigte Schnittstelle.

Liste 8.1: Schnittstelle der Funktion **fill**

```

1 void fill ( Pixel bild[N][N],           // Originalbild
2             int nz, int ns,             // Bildhöhe, Bildbreite
3             int inz,                    // Zeilenindex Impfpixel
4             int ins,                    // Spaltenindex Impfpixel
5             Pixel oldval,                // alter Farbwert
6             Pixel newval )              // neuer Farbwert

```

Der alte Grauwert an der Stelle (10,10) ist 255, als neuer Grauwert wird 127 verwendet.

8.2. Darstellung von Mengen kleiner Zahlen

6 Punkte

Um Mengen positiver ganzer Zahlen darzustellen können die Bits geeigneter Datenstrukturen verwendet werden. Mit einem Byte (Datentyp `char`) können die acht Werte von 0 bis 7 codiert werden, indem die Bits der entsprechenden Stellenwertigkeit gesetzt werden. Abbildung 8.2 zeigt ein Beispiel. Zur Manipulation der Bits werden die Bit-Operationen von `C++` verwendet.

7	6	5	4	3	2	1	0
0	0	1	0	1	0	1	0

Abbildung 8.2.: Darstellung einer Menge von maximal 8 Zahlen aus dem Bereich 0 bis 7 durch die 8 Bits eines Bytes. Hier ist die Menge {2, 4, 6} dargestellt.

Zur leichteren Handhabung werden typische Mengenoperationen als Funktionen implementiert. Folgende Funktionen (Prototypen in Liste 8.2) sind zu implementieren und mit geeigneten Beispielen zu testen:

set_count Zählt die gesetzten Bits und gibt deren Anzahl zurück. Das entspricht dem Umfang der Menge.

set_first Gibt die Nummer des ersten Elementes zurück (im Beispiel oben die 1). Wenn kein Element vorhanden ist wird der Wert 8 zurückgegeben.

set_clear Ein bestimmtes Bit (Element der Menge) über dessen Nummer löschen.

set_complement Erzeugt die Komplementmenge.

set_empty Löscht alle Bits (leere Menge erzeugen).

set_print Einfache Ausgabe als Folge von 8 Nullen und Einsen. Der zweite Parameter übernimmt einen Namen, der vor der Bitfolge ausgegeben wird. Zum Beispiel wie folgt, wobei `s1` der übergebene Name ist:

Menge `s1` : 11010101

set_set Ein bestimmtes Bit (Element der Menge) über dessen Nummer setzen.

Liste 8.2: Prototypen der Funktionen zur Mengendarstellung

```

1 // ##### TYPE DEFINITIONS - LOCAL TO THIS SOURCE FILE #####
2 typedef char      SmallSet;
3 typedef unsigned int  uint;
4
5 // ##### PROTOTYPES - LOCAL TO THIS SOURCE FILE #####
6 uint set_count    ( SmallSet *s );
7 uint set_first    ( SmallSet *s );
8 void set_clear    ( SmallSet *s, uint n );
9 void set_complement ( SmallSet *s );
10 void set_empty    ( SmallSet *s );
11 void set_print    ( SmallSet *s, string name );
12 void set_set      ( SmallSet *s, uint n );

```

8.3. Beliebige große Mengen^{freiwillig}

Erweitern Sie die Lösung so, daß mit Hilfe von `char`-Feldern Mengen positiver ganzer Zahler von beliebigem Umfang verwendet werden können.

8.4. Erzeugung von Permutationen^{freiwillig}

Eine Permutation entsteht durch die Umordnung einer Objekt- oder Zahlenreihe. Eine Aufgabenstellung aus der Kombinatorik ist die systematische Erzeugung aller $n!$ Anordnungen einer Menge von n Zahlen. Die Menge $(1, 2, 3)$ besitzt $3! = 6$ Permutationen (siehe Tabelle 8.1).

Eine Möglichkeit zur Erzeugung von Permutationen kann rekursiv programmiert werden. Ein Feld `a[]` ist anfänglich zum Beispiel fortlaufend mit den Werten $0, 1, 2, \dots, 7$ belegt und wird wie folgt permutiert:

Die Permutation der Stufe 0 entsteht durch Eintauschen des Wertes des 1. Feldplatzes mit allen anderen Feldplätzen (8 Möglichkeiten). Die 1. Stufe berücksichtigt nur noch die Feldplätze 2 bis n (Abbildung 8.3a), und so weiter.

Die Ordnung n der Permutationen wird in einer globalen Variablen `n` abgelegt. Die Funktion `void perm (int k)` kann so geschrieben werden, daß bei einem Startaufruf mit `perm (0)` gemäß Ablauf in Abbildung 8.3b die Permutationen für $0, 1, 2, 3, \dots$ der Reihe nach erzeugt werden. Erst bei Erreichen der Stufe $n - 1$ wird die jeweilige Permutation (Inhalt des Feldes `a[]`) ausgegeben.

1	2	3
1	3	2
2	1	3
2	3	1
3	2	1
3	1	2

Tabelle 8.1.: Permutationen von 3 Zahlen

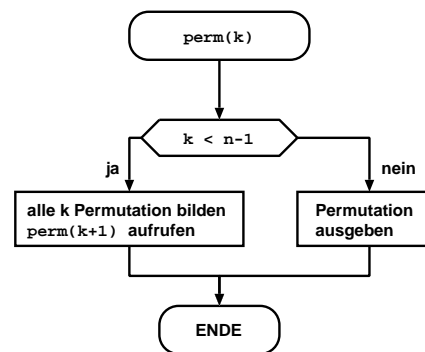
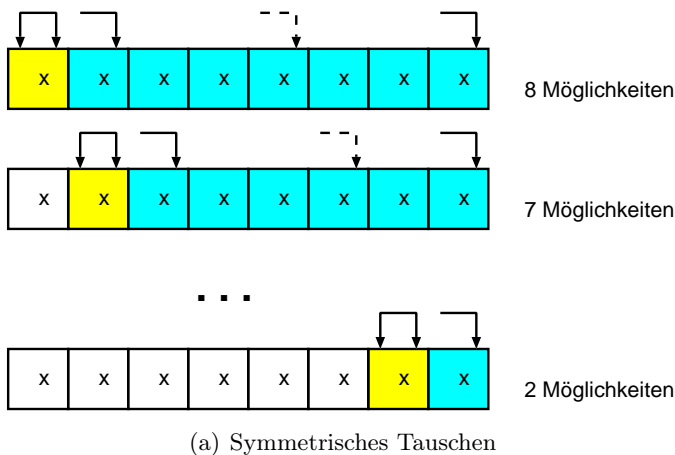


Abbildung 8.3.: Erzeugung von Permutationen

Die Korrektheit der Ergebnisse kann nur für kleine n (zum Beispiel $n = 2, 3, \dots$) direkt überprüft werden. Für größere n kann die Programmausgabe in eine Datei geschrieben werden. Ändern Sie Ihr Programm so ab, daß die Ausgabe in die Datei `perm8.dat` geschrieben wird und verwenden Sie für n auf den Wert 8.

Wenn eine Zeile pro Permutation erzeugt wurde (hier $8! = 40320$ Zeilen), dann kann mit dem Kommando

```
wc -l perm8.dat
```

zunächst die Anzahl der erzeugten Permutationen festgestellt werden, die hier der Anzahl der Zeilen der Datei `perm8.dat` entspricht. `wc` ist ein **Unix**-Kommando zur Wort- und Zeilenzählung, mit der Option `-l` wird jedoch nur die Zeilenzahl ausgegeben.

Nun ist festzustellen, ob alle Permutationen voneinander verschieden sind. Dazu wird der Dateiinhalt sortiert und evtl. vorhandene doppelte Einträge (sogenannte Dupletten) entfernt. Das Ergebnis wird wieder mit `wc` ausgezählt. Das wird durch den folgenden Befehlsstapel von der Kommandozeile aus erledigt:

```
cat perm8.dat | sort -u | wc -l
```

Die Zeilenzahlen der unsortierten und der sortierten Ausgaben müssen übereinstimmen

Das Kommando `cat` gibt die Datei `perm8.dat` auf die Kommandozeile aus. Durch die Umleitungen `|` (shell pipe) wird die Ausgabe jedoch unmittelbar an das nachfolgende Kommando `sort` übergeben und danach sofort an `wc`. `sort` ist das *Unix*-Sortierkommando, die Option `-u` erzwingt, daß mehrfach vorhandene Datensätze nach der Sortierung nur noch einmal vorkommen.

Weicht die Satzanzahl der beiden Zählungen voneinander ab, dann ist die Erzeugung der Permutationen noch fehlerhaft.

8.5. Sudoku-Rätsel lösen *freiwillig*

Die Erzeugung aller Lösungen eines Sudoku-Rätsel ist ein Paradebeispiel für die Verwendung der Rekursion. Abbildung 8.4 zeigt ein Beispiel für ein 9×9 -Sudoku. Das Schema muß bekanntlich so ergänzt werden, daß die Zahlen 1 bis 9 in jeder Zeile, in jeder Spalte und in jedem 3×3 -Teilfeld genau einmal vorkommen.

	0	1	2	3	4	5	6	7	8
0	3		4				6		2
1	9			6	2	7			4
2	6			1		4			7
3	2	4	9				7	3	1
4	1	6						8	5
5		8	3				4	6	
6	7			8		5			3
7				2	6	3			
8	8		5				9		6

Abbildung 8.4.: Ein Sudoku-Rätsel mit 2 Lösungen

Algorithmus:

1. Suche, beginnend von der Position $[0,0]$ (linke obere Ecke), die erste unbesetzte Position.
Wenn keine unbesetzte Position mehr vorhanden ist, ist eine Lösung gefunden.
2. Ermittle für die gefundene unbesetzte Position durch Untersuchung der zugehörigen Zeile und Spalte sowie des zugehörigen Teilfeldes alle Zahlenwerte, die in diesem Feld stehen dürfen.
Wenn diese Menge leer ist, dann ist das Rätsel nicht vollständig lösbar; die Suche wird abgebrochen.
3. Setze einen Wert aus der in Schritt 2 gefundenen Menge in das freie Feld ein und gehe zu Schritt 1.

Das erste frei Feld im obigen Beispiel ist das Feld $[0, 1]$. Die Menge der dort einsetzbaren Zahlen lautet $\{1, 5, 7\}$. Der Reihe nach eingesetzt, entstehen 3 Sudokus mit einem freien Feld weniger. Deren erstes freies Feld hat den Index $[0, 3]$.

Der angegebene Algorithmus arbeitet auch, wenn mehrere Lösungen vorhanden sind. Ist das Ausgangsfeld leer, dann werden alle möglichen Lösungen (etwa $6,67 \cdot 10^{21}$) gefunden! Die Rekursion sollte also nach einer vorgegeben Anzahl gefundener Lösungen beendet werden.

A. Erstellen von Programmen

A.1. Editieren, Compilieren, Binden

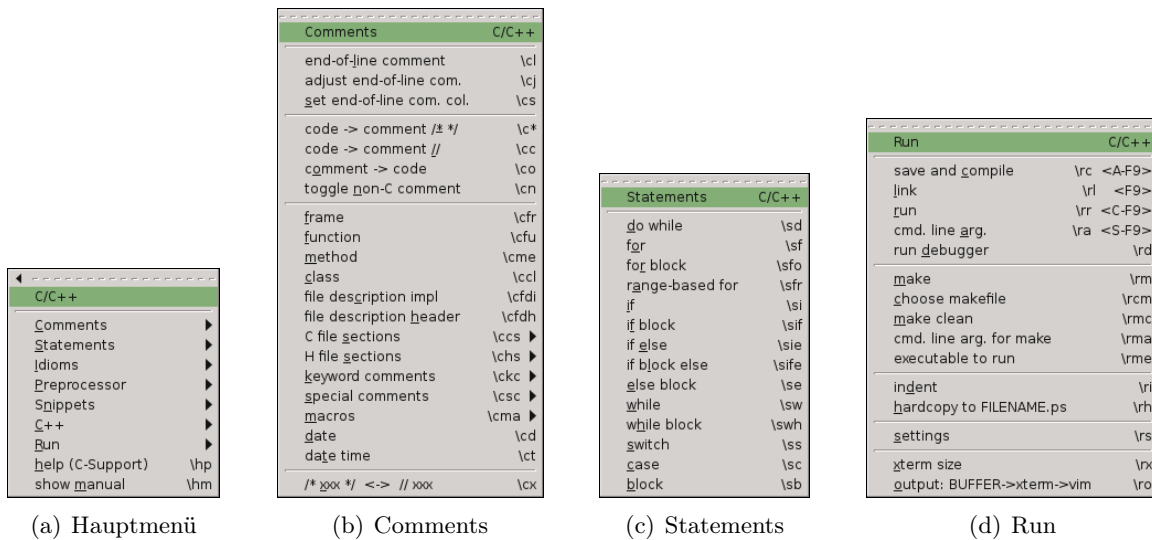


Abbildung A.1.: gVim - Menüs der *C/C++*-Unterstützung (Auswahl)

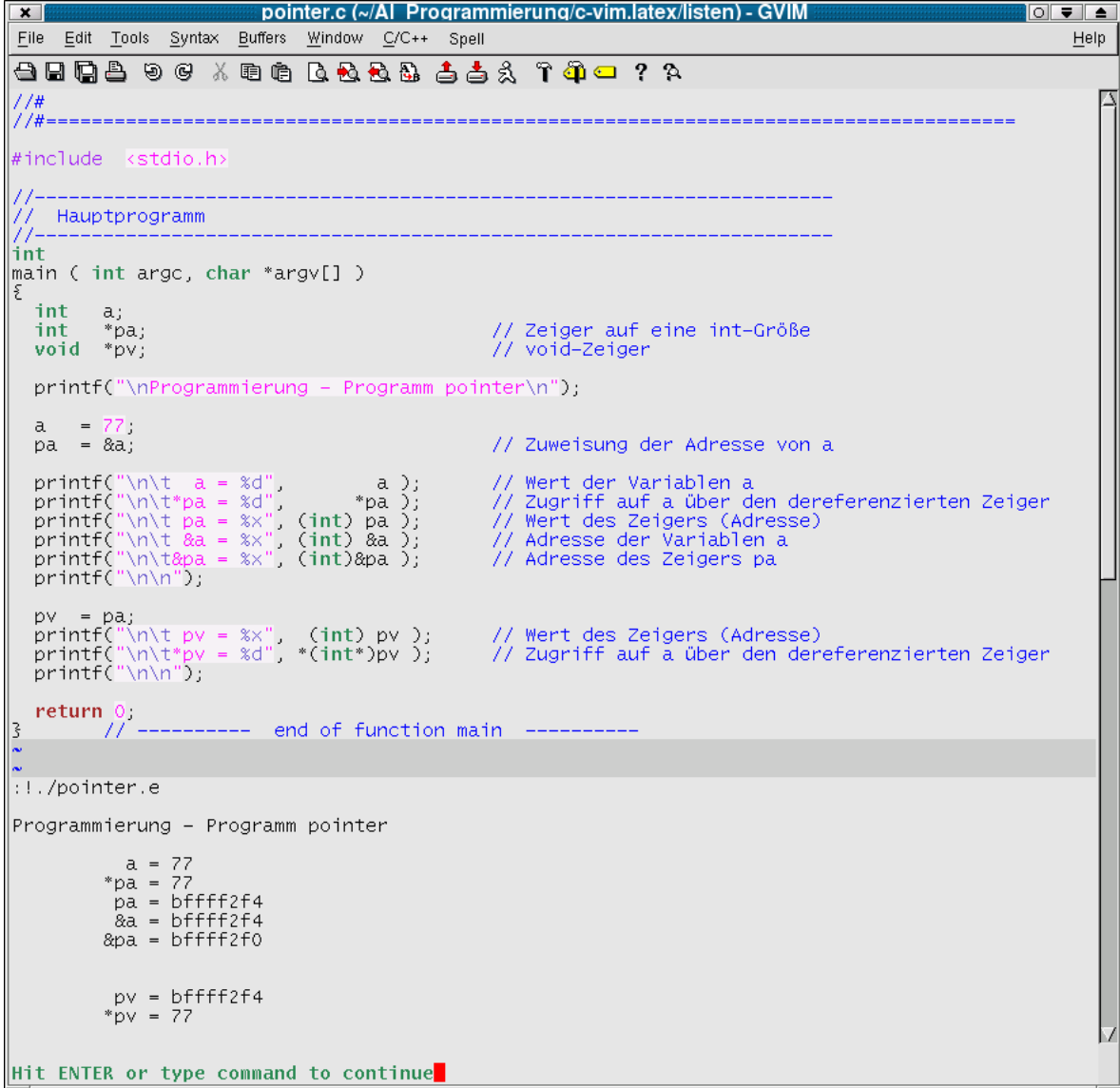
Zur Programmierung wird der sehr leistungsfähige Editor **gVim** verwendet. Dieser Editor ist eine Weiterentwicklung des klassischen *Unix*-Editors **vi**. Die nichtgraphische Version des **gVim** heißt **Vim**. Die Handhabung des Editors ist unter anderem in den Büchern und Anleitungen beschrieben, die am Ende dieses Anhangs aufgeführt sind.



gVim kann als integrierte Entwicklungsumgebung (IDE) eingesetzt werden. Der Editor kann beliebig viele Dateien zum Editieren öffnen. Jede offene Datei wird in einem Editierpuffer gehalten (Das Menü **Buffers** ermöglicht die Navigation). Die Fenster können längs und quer geteilt werden.

Das Menü **Run** (Abbildung A.1) ermöglicht die Compilierung des aktiven Puffers (Eintrag **save and compile**) mit Standardeinstellungen für den Compiler. Der Pufferinhalt wird vor der Übersetzung abgespeichert. Beim Auftreten von Übersetzungsfehlern wird ein Fehlerfenster geöffnet. Die fehlerhaften Code-Zeilen im Programmfenster können durch Auswahl der Fehlereinträge (Maus oder Richtungstasten) im Fehlerfenster angesprungen und berichtigt werden. Für Programme, die nur aus einer einzigen Datei bestehen, besteht die Möglichkeit, diese Datei nach dem Übersetzen zu binden (Menüeintrag **link**) und die Ausführung zu starten (Menüeintrag **run**). Dadurch kann die Erstellung eines **make**-Skriptes vermieden werden. Die Programmausgaben erscheinen in einer subshell im Editorfenster (Abbildung A.2 unten).

Bei Anwahl des Menüpunktes **Run**→**link** wird der Pufferinhalt vor dem Binden übersetzt, wenn die evtl. vorhandene Object-Datei älter ist, als die vor dem Binden gerade gesicherte Quelldatei.



```

pointer.c (~/.AI Programmierung/c-vim.latex/listen) - GVIM
File Edit Tools Syntax Buffers Window C/C++ Spell Help
// #
// =====
#include <stdio.h>
// -----
// Hauptprogramm
// -----
int
main ( int argc, char *argv[] )
{
    int    a;
    int    *pa;           // Zeiger auf eine int-Größe
    void    *pv;          // void-Zeiger

    printf("\nProgrammierung - Programm pointer\n");

    a    = 77;
    pa    = &a;           // Zuweisung der Adresse von a

    printf("\n\t a = %d",      a );      // Wert der Variablen a
    printf("\n\t *pa = %d",    *pa );      // Zugriff auf a über den dereferenzierten Zeiger
    printf("\n\t pa = %x",    (int) pa );  // Wert des Zeigers (Adresse)
    printf("\n\t &a = %x",    (int) &a );  // Adresse der Variablen a
    printf("\n\t &pa = %x",   (int) &pa );  // Adresse des Zeigers pa
    printf("\n\n");

    pv    = pa;
    printf("\n\t pv = %x",    (int) pv );  // Wert des Zeigers (Adresse)
    printf("\n\t *pv = %d",   *(int*)pv ); // Zugriff auf a über den dereferenzierten Zeiger
    printf("\n\n");

    return 0;
} // ----- end of function main -----
~
~
:!. ./pointer.e
Programmierung - Programm pointer

    a = 77
    *pa = 77
    pa = bffff2f4
    &a = bffff2f4
    &pa = bffff2f0

    pv = bffff2f4
    *pv = 77

Hit ENTER or type command to continue

```

Abbildung A.2.: gVim - Programmausgabe in einer subshell im Editorfenster

Bei Anwahl des Menüpunktes **Run**→**run** wird der Pufferinhalt vor dem Starten des Programmes übersetzt und gebunden, wenn die evtl. vorhandene Object-Datei oder die ausführbare Datei älter ist, als die vor dem Starten gerade gesicherte Quelldatei.

Für größere Projekte ist die Verwendung von **make**-Skripten oder ähnlicher Werkzeuge erforderlich. Das **make**-Programm kann dann ebenfalls über das **Run**-Menü gestartet werden.

Für Programme mit längeren Ausgaben ist der Aufruf mit Weiterleitung der Ausgaben durch einen sogenannte pager (Programm zum Blättern in Dateien oder Konsolausgaben) vorgesehen.

Die Hauptmenüeinträge **Comments**, **Statements** und so weiter erlauben das Einsetzen von vorbereiteten Kommentaren und von **C**- beziehungsweise **C++**-Anweisungen (Abbildung A.1).

Für einige der bei der Programmerzeugung häufig auszuführenden Tätigkeiten stehen Tastenkürzel zur Verfügung (siehe Tabelle A.1)

Tastenkombination	Wirkung
F2	Inhalt des aktiven Puffers sichern.
F3	Dialog zum Öffnen von Dateien aufrufen.
Shift-F3	Neue Dateien anlegen.
F5	Fehlerfenster öffnen.
F6	Fehlerfenster schließen.
F7	Zur Position des vorherigen Fehlers springen.
F8	Zur Position des nächsten Fehlers springen.
Alt-F9	Pufferinhalt abspeichern und danach den Pufferinhalt compilieren.
F9	Programm binden, gegebenenfalls vorher compilieren.
Strg-F9	Programm ausführen, gegebenenfalls vorher compilieren und binden.
Shift-F9	Kommandozeilenargumente für das Programm festlegen.

Tabelle A.1.: gVim - Die Funktionstastenkombinationen der **C/C++**-Erweiterung

Der Editor **Vim/gVim** ist ein leistungsstarker Programmiereditor für den Profi. Die Bedien- und Anwendungsmöglichkeiten sind entsprechend umfangreich. Neben der online-Hilfe und der Originaldokumentation stehen folgende Bücher und Anleitungen zur Verfügung:

Mehner, Fritz gVim-Kurzanleitung

Einführung in die Bedienung und einige Zusätze (**C**-Unterstützung, Quellcodeformatierung, Navigation im Quellcode).

Als PDF-Datei auf den Web-Seiten zu den Veranstaltungen *Programmierung mit C++ 1* und *Programmierung mit C++ 2*.

Qualine, Steve Vi IMproved - Vim¹

New Riders Publishing, 2001, ISBN: 0735710015

oder unter <http://vim.sourceforge.net/docs.php> als PDF-Datei

Robbins, Arnold vi und Vim – kurz & gut ¹

O'Reilly, 2011, ISBN 978-3-89721-321-0

(Kurzanleitung, 96 Seiten, 1. Auflage)

Lamb, Linda / Robbins, Arnold Textbearbeitung mit dem vi-Editor ¹

O'Reilly, 1999, ISBN 3-89721-126-2

(deutsche Übersetzung der 6. amerikanischen Auflage)

¹mehrere Exemplare in der Bibliothek der FH SWF vorhanden

<http://vim.sourceforge.net> The Vim (Vi IMproved) Home Page

Homepage des Vim-Projektes; viele Verweise auf andere Seiten und auf Dokumentation.

Vim-Skripte zur Erweiterung des Editors, Tips, Neuigkeiten, ...

A.2. Endlosschleife beenden

Gelegentlich kommt es vor, daß sich ein Programm nach dem Aufruf nicht mehr selbst beendet. Die häufigste Ursache ist eine sogenannte Endlosschleife: auf Grund eines Programmierfehlers wird eine Schleifensteuerung falsch oder gar nicht beeinflusst. Dadurch wird die Schleife nicht beendet.

Kommandozeilenprogramme können unter *Linux/Unix* mit der Tastenkombination **Strg-C** beendet werden. Gegebenenfalls muß der Befehl öfters wiederholt werden.

Bei Programmen, die Konsolenausgaben in der Endlosschleife durchführen (zum Beispiel mit **printf**), ist **Strg-C** meist wirkungslos. In diesem Fall muß die Prozeßnummer des Programmes ermittelt und der Prozeß beendet werden. Unter einer graphischen Oberfläche, wie zum Beispiel KDE unter *Linux*, verwendet man am einfachsten einen Prozeßmanager, der alle oder ausgewählte Prozesse anzeigen und im Dialog beeinflussen kann.

Abbildung A.3 zeigt den Prozeßmanager **ksysguard**. Im Suchfenster wurde als Suchbegriff bereits der Name des abzubrechenden Programmes eingegeben (hier: **endlos**). Im Hauptfenster ist der zugehörige Prozeß angezeigt. Dieser Eintrag kann mit der Maus ausgewählt werden. Mit Hilfe der rechten Maustaste wird ein menü geöffnet, das das Zusenden des Signals **KILL** ermöglicht.

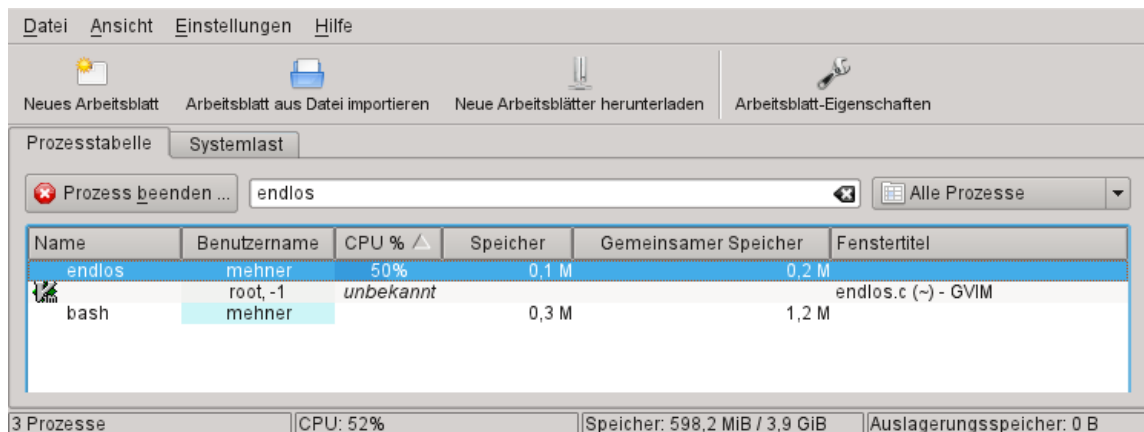


Abbildung A.3.: Einen Prozeß mit Hilfe des Prozeßmanagers **ksysguard** beenden

Wenn das fehlerhafte Programm aus dem Editor **gVim** heraus aufgerufen wurde, dann bemerkt dieser den Abbruch. Der Editor kann sofort wieder benutzt werden um den Fehler zu beheben. Der Abbruch des Editors ist in keinem Fall erforderlich!

B. Programmdokumentation

Programme sollen grundsätzlich richtig, zweckmäßig, vollständig, verständlich und leicht wartbar sein. Diese Ziele werden unter anderem durch eine Reihe von Merkmalen und Eigenschaften sichergestellt, die Bestandteile des Programmierstils sind. Dazu gehören:

- die Form der Kommentierung
- die Art der Formatierung
- die Art der Benennung von Bezeichnern
- der Aufbau eines Moduls

Die praktische Umsetzung kann, besonders bei größeren Projekten, nicht allein dem persönlichen Geschmack des einzelnen Programmierers überlassen werden. Es ist üblich, die notwendigen Vorgaben in firmen- oder projektbezogenen Programmierrichtlinien niederzulegen und alle Beteiligten auf die Einhaltung dieser Richtlinien zu verpflichten.

Die durchgängige Einhaltung eines festgelegten Programmierstils erlaubt dann das automatische Herausziehen von Kommentaren zur Erzeugung von Projektdokumentationen, sowie das automatische Durchsuchen großer Quellcode-Bäumen. Dazu sind eine Reihe gebräuchlicher Werkzeuge vorhanden. Auch im Rahmen dieses Praktikums muß für jedes Programm ein Mindestumfang an Programmdokumentation vorhanden sein. Dieser wird in den nachfolgenden Abschnitten beschrieben. Grundsätzlich gilt folgendes:

- Kommentierungssprache ist in der Regel **Deutsch**. Die **Rechtschreibung** ist zu beachten! Wer sich nicht die Mühe macht, treffend und unter Verwendung der Fachsprache zu kommentieren, bei dem wird man vermuten, daß die Implementierung und die Programmzuverlässigkeit wegen fehlender Sorgfalt oder Unfähigkeit ebenfalls mit Mängeln behaftet sind.
- **Personenangaben** sind vollständig aufzuführen (Vor- und Nachname, gegebenenfalls Matrikelnummer). Die Verwendung von Spitznamen, Phantasiebezeichnungen, Schimpfwörtern und ähnliches ist zu unterlassen.

B.1. Dateibeschreibung

Liste B.1: Kopfkomentar als Dateibeschreibung

```

1 // =====
2 //
3 //      Filename:  p-1-1.cc
4 //
5 //      Description:  Programmierung mit C++ 1, Blatt 1, Aufgabe 1.1
6 //                   Summenbildung bei endlichen Reihen
7 //
8 //      Version:    1.0
9 //      Created:    18.09.2013 09:19:13
10 //      Revision:   none
11 //      Compiler:   g++

```

```

12 //
13 //      Author:  Dr. Fritz Mehner (fgm), mehner.fritz@fh-swf.de
14 //      Organization:  FH Südwestfalen, Iserlohn
15 //
16 // =====

```

Jede Datei muß an ihrem Anfang eine Dateibeschreibung in Form eines Kommentares enthalten. Wenn die Datei in einem Editor geöffnet wird ist dieser Kommentar sofort sichtbar. Ein Kommentar, wie in Liste B.1 dargestellt, sollte als Schablone in einer Datei vorliegen, die bei Bedarf einkopiert wird. Damit wird sichergestellt, daß die Dateibeschreibungen stets die gleiche Form besitzen.

B.2. Abschnittskommentare und Zeilenendkommentare

Kurze Funktionen und kurze, logisch zusammenhängende Programmabschnitte werden zur optischen Gliederung und zur Erläuterung des Inhaltes mit Abschnittskommentaren versehen (Liste B.2, Zeilen 1-3, 17-19).

Liste B.2: Abschnittskommentare und Zeilenendkommentare

```

1  //-----
2  //  Hauptprogramm
3  //-----
4  int
5  main ( )
6  {
7      int      n          = 1000;          // Anzahl der Summanden
8      int      width      = 10;           // Ausgabebreite summe
9      int      precision  = 5;           // Nachkommastellen summe
10     double   summe;           // Reihensumme (Ergebnis)
11     double   summand;        // Hilfsvariable
12     int      i, vz;          // Hilfsvariablen
13
14     cout << "Praktikum 'Programmierung mit C++ 1' / Blatt 1 / Aufgabe 1.1\n";
15     cout << "Summenbildung bei endlichen Reihen\n\n";
16
17     //-----
18     //  Summation der harmonischen Reihe
19     //-----
20     summe = 0.0;
21     for ( i=1; i<=n; i+=1 )
22         summe = summe + 1.0/i;
23
24     cout << "          Summe der harmonischen Reihe von 1..." << n;
25     cout << " : " << fixed << setw(width) << setprecision(precision) << summe << "\n";
26     return 0;
27 } // ----- end of function main -----

```

Die Abschnittskommentare (Blockkommentare) richten sich in ihrer Einrücktiefe stets nach der Schachtelungstiefe des zu kommentierenden Codes. Der Kopf des Hauptprogrammes in Liste B.2 hat die Schachtelungstiefe 0, der Kopfkomentar (Zeilen 1-3) ist dementsprechend nicht eingerückt. Der Abschnittskommentar über der Summation (Zeilen 17-19) hat die Schachtelungstiefe 1 und ist dementsprechend eine Tabulatorweite eingerückt (hier 2 Zeichenpositionen).

Kurze Erläuterungen werden als Zeilenendkommentare an die Codezeile angehängt (Liste B.2, Zeile 7-12). Wenn möglich, sollten Zeilenendkommentare (abschnittsweise) jeweils in der selben Spalte beginnen. Zu kommentieren sind

- **Variablen** (Bedeutung und Verwendung; der Kommentar steht bei der Vereinbarung).
- **Funktionen** (mindestens: Zweck, Parameter und Rückgabewert; bei sehr kurzen oder einfachen Funktionen reicht ein Kopfkomentar).
- **Klassen** (mindestens: Zweck der Klasse, kurze Beschreibung der Datenkomponenten und Methoden; die Implementierungen der Methoden sind wie Funktionen zu behandeln).

Kommentare sollen kurz, treffend und aussagekräftig sein. Der Kommentartext **Schleifenzähler** stellt eine sinnvolle Aussage dar:

```
int i, j;                // Schleifenzähler
```

Der Kommentartext **Hilfsvariable** ist überflüssig. Jede derart vereinbarte Variable ist eine Hilfsvariable:

```
int i, j;                // Hilfsvariable
```

Die Gesamtlänge einer Zeile (mit oder ohne Kommentar) sollte 90 Zeichen nicht überschreiten. Damit kann das Suchen in Querrichtung vermieden werden und die Datei bleibt bei üblichen Papierbreiten ohne abgeschnittene Zeilenenden abdruckbar.

B.3. Einrückung und Schachtelungstiefe

Die Einrückung von Programmkonstrukten muß in der Regel mit der Schachtelungstiefe dieser Konstrukte übereinstimmen.

Liste B.3: Einrückung innerhalb einer Funktion

```

1 // == FUNCTION =====
2 //      Name:  init_matrix
3 // Description: quadratische Matrix initialisieren
4 // =====
5 void init_matrix ( double matrix[N][N], int n, double init_val )
6 {
7     for ( i=0; i<n; i+=1 ) {
8         for( j=0; j<n; j+=1 ) {
9             matrix[i][j] = 0.0;
10        }
11    }
12    return;
13 } // ----- end of function init_matrix -----
```

- Der Kommentar zur Funktion, der Funktionskopf und die Rumpfkammern (Liste B.3, Zeilen 1-6, 13) besitzen die Schachtelungstiefe 0.
- Die erste **for**-Schleife (Liste B.3, Zeile 7) besitzt die Schachtelungstiefe 1.
- Die zweite **for**-Schleife in Zeile 8 gehört zum Rumpf der umgebenden **for**-Schleife und besitzt deshalb die Schachtelungstiefe 2.
- Die Zuweisung in Zeile 9 gehört zum Rumpf der zweiten **for**-Schleife in Zeile 8 und besitzt deshalb die Schachtelungstiefe 3.

Einrückung ist die wichtigste Maßnahme zur Erzeugung lesbarer Programme und ist daher *zwingend* anzuwenden! Unter anderem werden Strukturfehler die sich in der Schachtelung widerspiegeln dadurch sofort erkennbar.

Gute Programmiereditoren unterstützen die Einrückung beim Einfügen. Zur Formatierung vorhandener Quellen gibt eine ganze Reihe von Formatierungsprogrammen (engl. *beautifier*), die eine Vielzahl von Darstellungsoptionen unterstützen. Ein bekanntes Beispiel ist das Programm **indent**.

C. Fehlersuche

C.1. Die häufigsten Anfängerfehler

Die folgende Liste enthält Programmierfehler, die erfahrungsgemäß häufig von Programmieranfängern gemacht werden. Es wird dringend empfohlen, jedes Programm auf diese Fehler hin durchzusehen. Das sollte auch dann geschehen, wenn das Programm anscheinend die richtigen Ergebnisse liefert.

Nr.	Fehler	Beschreibung
1	Variable besitzt keinen Anfangswert	Variablen vom Speichertyp auto werden nicht mit Null initialisiert und müssen immer einen Anfangswert erhalten, wenn ihr erster Gebrauch auf der rechten Seite einer Zuweisung oder in einer Parameterliste stattfindet.
2	Ganzzahlige Division	Verwendung der ganzzahligen Division anstatt der reellen Division, also zum Beispiel laenge *= 1/2; anstatt laenge *= 0.5; . Der Faktor 1/2 im ersten Beispiel wird auf Grund der Regeln für die ganzzahlige Division zu Null berechnet!
3	Schleife läuft zu kurz / zu weit	Eine Schleife hat einen Durchlauf zuviel oder zu wenig, weil die Abbruchbedingung nicht richtig ist (zum Beispiel i<=n anstatt i<n) oder weil die Initialisierung der Steuervariablen falsch ist (zum Beispiel i=1 anstatt i=0).
4	Zugriff über die Feldgrenzen hinaus	Zugriff auf einen Feldindex, der nicht vorhanden ist.
5	Feld zu klein	Die Größenangabe eines Feldes in einer Vereinbarung bezeichnet die Elementanzahl, nicht den letzten Index! Bei Zeichenpuffern (char -Felder) muß meist ein zusätzliches Element für das Abschlußzeichen (die binäre Null) vorgesehen werden.
6	Zuweisung statt Vergleich	Der Operator zur Feststellung der Gleichheit zweier Werte besteht aus zwei Gleichheitszeichen (==). Die Bedingung in if (n = n_alt) stellt eine Zuweisung dar: der Wert der Variablen n wird überschrieben und anschließend als Wahrheitswert verwendet!
7	Strichpunkt hinter dem Kopf einer Schleife oder Bedingung	Ein Strichpunkt hinter einem Schleifenkopf oder hinter dem Kopf einer Verzweigung (if , switch) trennt den nachfolgenden Rumpf ab; der Kopf wird dadurch zu einer selbständigen, abgeschlossenen Anweisung. Wenn die Bedingung wahr ist wird die Schleife dadurch zur Endlosschleife: <pre>while (n <= n_max); { ... }</pre>

C.2. Der Debugger DDD

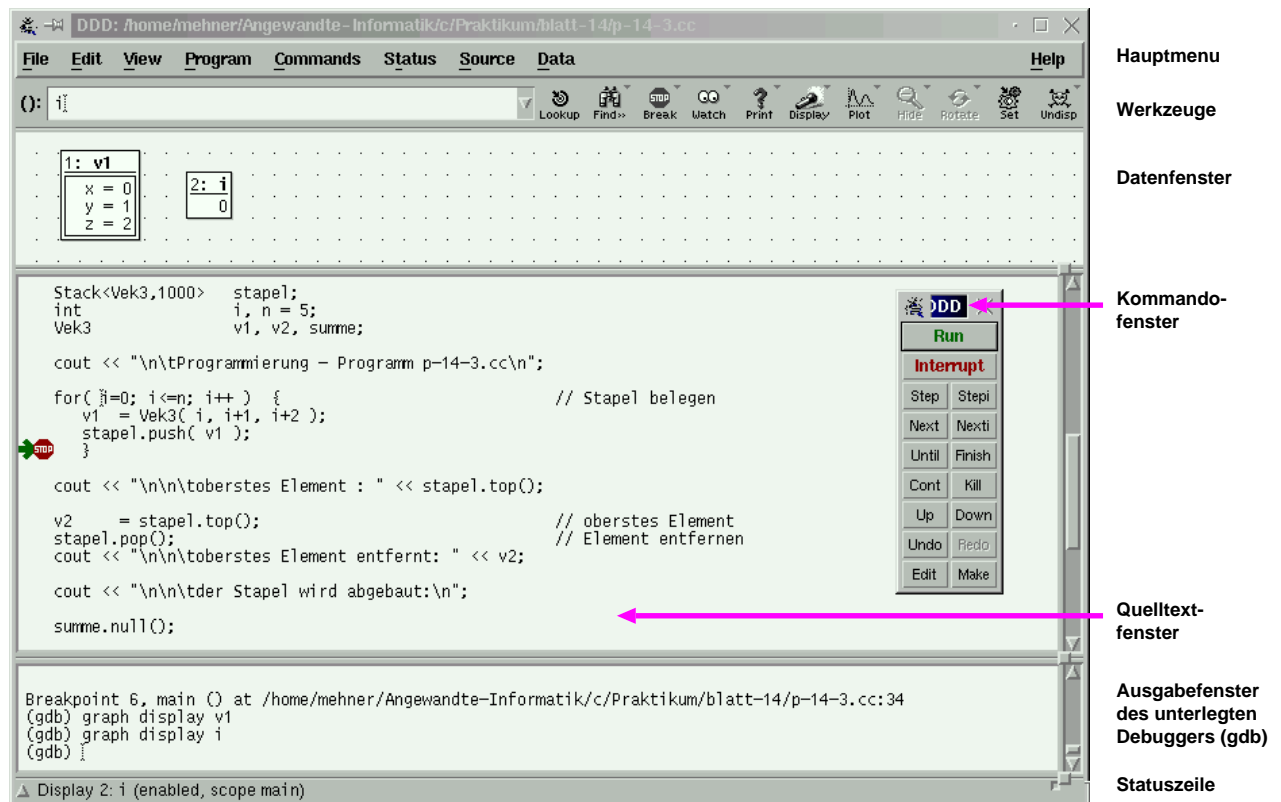
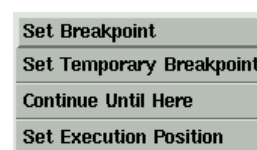


Abbildung C.1.: Der Debugger DDD

Zur Fehlersuche in Programmen kann der Debugger DDD verwendet werden. Das zu testende Programm muß dazu als ausführbare Datei vorliegen, das heißt es muß übersetzt und gebunden sein und enthält damit keine syntaktischen Fehler mehr. Der DDD kann über das entsprechende Symbol auf der Arbeitsfläche gestartet werden. Im Dialog wird das zu testende und ausführbare Programm geöffnet. Die zugehörige Quelldatei erscheint nun im Quelltextfenster (Abbildung C.1). Außerdem erscheint im Quelltextfenster das frei verschiebbare Kommandofenster.



(a) Haltepunktmenü



(b) Kommandomenü

Haltepunkte (breakpoints). Setzt man die Schreibmarke an den Anfang einer ausführbaren Quellcodezeile und betätigen die rechte Maustaste, dann erscheint das Haltepunktmenü (Abbildung C.2a). Die erste Auswahl (**Set Breakpoint**) setzt einen festen Haltepunkt. Der zweite Eintrag (**Set Temporary Breakpoint**) setzt einen Haltepunkt, der nach dem Erreichen wieder gelöscht wird. Der dritte Eintrag (**Continue Until Here**) ermöglicht die Programmausführung oder -weiterführung bis zu der Zeile vor der Schreibmarke.

Abbildung C.2.: Haltepunkt- und Kommandomenü

Programm schrittweise testen. Durch die Anwahl des Eintrages **Run** im Kommandomenü (Abbildung C.2b) wird das Programm gestartet und bis zur letzten ausführbaren Programmanweisung *vor* dem Haltepunkt ausgeführt. Die erreichte Stelle wird durch einen roten Pfeil gekennzeichnet (Abbildung C.1). Mit Hilfe der anderen Möglichkeiten des Kommandomenüs kann das Programm Zeile für Zeile ausgeführt werden (Tabelle C.1). Nach jedem Schritt können dann zum Beispiel Variablenwerte angesehen oder verändert werden.

Run	Starte das zu testende Programm
Interrupt	Unterbreche das zu testende Programm
Step	eine Zeile weiter (bei Unterprogrammaufrufen wird in das Unterprogramm gesprungen)
Next	eine Zeile weiter (bei Unterprogrammaufrufen wird der Aufruf übersprungen)
Until	Programm bis zur nächsten ausführbaren Zeile fortsetzen (zeilenweise Ausführung)
Cont	Programm nach einem Haltepunkt fortsetzen
Kill	Programm abbrechen

Tabelle C.1.: DDD - die Befehle des Kommandomenüs

Der Debugger **DDD** ist äußerst leistungsfähig und steht für mehrere Rechnerplattformen zur Verfügung. Er ist nicht auf **C/C++** beschränkt. Eine umfassendere Darstellung würde den Rahmen an dieser Stelle sprengen. Alles weitere muß der Originaldokumentation entnommen werden.

Haltepunkt setzen	<ul style="list-style-type: none"> - Cursor am Anfang der gewünschten Zeile positionieren - rechte Maustaste niederhalten (Haltepunktmenü) - Set Breakpoint oder Delete Breakpoint wählen - am Zeilenanfang erscheint ein Stop-Schild
Haltepunkt löschen	<ul style="list-style-type: none"> - Cursor über dem Stop-Schild positionieren - rechte Maustaste niederhalten (Haltepunktmenü) - Delete Breakpoint wählen
Programm bis zum Haltepunkt ausführen	<ul style="list-style-type: none"> - Menüeintrag Run wählen
Wert einer Variable ansehen	<ul style="list-style-type: none"> - Mauszeiger im Quelltextfenster über dem Variablennamen positionieren
Datenelement im Datenfenster anzeigen	<ul style="list-style-type: none"> - Mauszeiger im Quelltextfenster über dem Namen positionieren - rechte Maustaste niederhalten (Display-Menü erscheint) - Display wählen - im Datenfenster erscheint die entsprechende Anzeige
Datenelement im Datenfenster erweitern	<p>Bei Strukturen, Feldern, komplexen Objekten und so weiter kann die Darstellung erweitert oder zusammengefaßt werden:</p> <ul style="list-style-type: none"> - Mauszeiger im Datenfenster über dem Datenelement positionieren - rechte Maustaste niederhalten (Menü) - Show All oder Hide All wählen

Tabelle C.2.: DDD - grundlegende Bedienung

Index

- Abschnittskommentar, 38
- Abweichung, 17
- ASCII-Tabelle, 5
- beautifier, 40
- Bildbearbeitung, 21, 27
- Bildbereich füllen, 27
- Binden, 33
- breakpoint, 42
- Buchstaben
 - abzählen, 11
- Caesar-Verschlüsselung, 14
- cat**, 30
- chiffrieren, 13
- codieren, 13
- Compilieren, 33
- Cosinus hyperbolicus, 3
- Dateibeschreibung, 37
- DDD**, 42
- Debugger, 42
- Debugging, 41
- decodieren, 13
- Editieren, 33
- Editor, 33
- Einrückung, 39
- Endlosschleife, 36
- Exponentialfunktion, 3
- Fehlersuche, 42
- Felder, 7
 - korrespondierende, 7
- Glättung, 23
- gnuplot**, 8
- Grauwertbild, 21
- gVim**, 33
- Gwenview**, 22
- Haltepunkt, 42
- indent**, 40
- Invertierung, 23
- Kantenerkennung, 23
- Kennwerte
 - statistische, 17
- Kosinusfunktion, 3
- ksysguard**, 36
- Logarithmus
 - natürlicher, 19
- make**, 35
- Maximum, 7, 17
- Median, 17
- Mengen, 27
- Minimum, 7, 17
- Mittelwert
 - arithmetischer, 7, 17
 - geometrischer, 17, 19
 - quadratischer, 17
- Permutation, 29
- Programmdokumentation, 37
- Programmierichtlinien, 37
- Programmierstil, 37
- Prozessmanager, 36
- Reihe
 - Cosinus hyperbolicus, 3
 - Exponentialfunktion, 3
 - geometrische, 1
 - harmonische, 1
 - alternierend, 1
 - Kosinusfunktion, 3
 - leibnizsche, 1
 - unendliche, 1
- Rekursion, 27, 29, 30
- ROT13-Verschlüsselung, 13
- Rundungsfehler, 6
- Schachtelung, 39
- Schachtelungstiefe, 38, 39

- Schleifen, 5
- Schwellwertoperation, 23
- sort**, 30
- Sortieraufwand, 18
- Sortieren durch Vertauschen, 18
- Spanne, 17
- Sudoku, 30
- Summation, 1

- Testen, 41
- Text
 - codieren, decodieren, 13

- UNIX
 - cat**, 30
 - indent**, 40
 - make**, 35
 - sort**, 30
 - wc**, 12, 30

- Verschiebeciffre, 13
- Verschiebung
 - zyklisch, 13
- vi**, 33
- Vim**, 33
- Vorzeichen
 - wechselndes, 1

- wc**, 12, 30

- xv**, 22

- Zahlenmengen, 28
- Zeichenklassen, 12
- Zeilenendkommentar, 38
- Ziffern
 - abzählen, 11
- Zustandsdiagramm, 11
- Zustandsvariablen, 11