Einführung in die Programmierung mit C

(Entwurf)

WS 06/07

Sven Eric Panitz

FH Wiesbaden

Version 15. März 2007

Die vorliegende Fassung des Skriptes ist ein Entwurf für die Vorlesung des Wintersemesters und wird im Laufe der Vorlesung erst seine endgültige Form finden. Kapitel können dabei umgestellt, vollkommen revidiert werden oder gar ganz wegfallen.

Dieses Skript entsteht begleitend zur Vorlesung des WS 06/07.

Naturgemäß ist es in Aufbau und Qualität nicht mit einem Buch vergleichbar. Flüchtigkeits- und Tippfehler werden sich im Eifer des Gefechtes nicht vermeiden lassen und wahrscheinlich in nicht geringem Maße auftreten. Ich bin natürlich stets dankbar, wenn ich auf solche aufmerksam gemacht werde und diese korrigieren kann.

Der Quelltext dieses Skripts ist eine XML-Datei, die durch eine XQuery in eine LaTeX-Datei transformiert und für die schließlich eine pdf-Datei und eine postscript-Datei erzeugt wird. Beispielprogramme werden direkt aus dem Skriptquelltext extrahiert und sind auf der Webseite herunterzuladen.

Inhaltsverzeichnis

1 Einführung **1-1**

1.1 Erste Programme 1-1

2 Grundkonzepte **2-1**

2.1 Operatoren 2-1

2.1.1 arithmetische Grundrechenarten 2-1

2.1.2 Vergleichsoperatoren 2-3

2.1.3 Logische Operatoren 2-4

2.1.4 Bedingungsoperator 2-5

2.2 Funktionen 2-6

2.2.1 Seiteneffekte 2-8

2.2.2 Prozeduren 2-11

2.2.3 rekursive Funktionen 2-12

2.3 Variablen 2-16

2.3.1 Zuweisung 2-16

2.3.2 lokale und globale Variablen 2-18

2.3.3 statische Variablen 2-20

2.3.4 Auch Parameter sind Variablen 2-22

2.3.5 Fragen des Stils 2-23

2.4 Kontrollstrukturen 2-24

2.4.1 if Verzweigungen 2-24

2.4.2 Schleifen 2-30

2.4.3 switch Verzweigungen 2-41

2.5 Zuweisung und Operatoren 2-45

2.5.1 Zuweisungsoperatoren 2-45

2.5.2 Inkrement und Dekrement 2-46

3 Softwarentwicklung **3-1**

3.1 Programmieren . . . 3-1

3.1.1 Disziplinen der Programmierung . . . 3-1

3.1.2 Was ist ein Programm . . . 3-3

3.1.3 Klassifizierung von Programmiersprachen . . . 3-4

3.2 Präprozessor und Linker: hilfreiche Freunde des Compilers . . . 3-5

3.2.1 Funktionsprototypen . . . 3-5

3.2.2 Header-Dateien . . . 3-7

3.2.3 Objektdateien und Linker . . . 3-8

3.2.4 Mehrfach inkludieren . . . 3-9

3.2.5 Konstantendeklaration . . . 3-11

3.3 Kompilierung: Mach mal! . . . 3-11

3.4 Dokumentation . . . 3-13

3.4.1 Kommentare . . . 3-13

3.4.2 Benutzung von Doxygen . . . 3-14

3.4.3 Dokumentation des Quelltextes . . . 3-15

3.5 Debuggen . . . 3-16

3.5.1 Debuggen auf der Kommandozeile . . . 3-17

3.5.2 Graphische Debugging Werkzeuge . . . 3-20

3.6 Programmierumgebung: Arbeiten mit Eclipse . . . 3-21

4 Daten **4-1**

4.1 Basistypen . . . 4-1

4.1.1 Zahlenmengen in der Mathematik . . . 4-1

4.1.2 Zahlenmengen im Rechner . . . 4-2

4.1.3 natürliche Zahlen . . . 4-2

4.1.4 ganze Zahlen . . . 4-3

4.1.5 Kommazahlen . . . 4-5

4.1.6 Zahlentypen in C . . . 4-7

4.2 Neue Namen für Typen einführen . . . 4-16

4.3 Strukturen . . . 4-16

4.3.1 Definition von Strukturen . . . 4-16

4.3.2 Erzeugen von Strukturobjekten . . . 4-17

4.3.3 Zugriff auf Strukturkomponenten . . . 4-18

4.3.4 Strukturen als Funktionsparameter . . . 4-18

4.3.5 Einwortnamen für Strukturen . . . 4-20

4.4 Aufzählungen 4-23
4.4.1 Definition von Aufzählungen 4-23
4.4.2 Fallunterscheidung für Aufzählungen 4-24
4.4.3 Einwortnamen für Aufzählungen 4-25
4.4.4 Bool'sche Werte als Aufzählung 4-25
4.5 Funktionen als Daten 4-26
4.5.1 typedef und Funktionstypen 4-28
4.6 Dynamische Daten 4-29
4.6.1 Adresstypen 4-29
4.6.2 der void-Zeiger 4-36
4.6.3 Rechnen mit Zeigern 4-37
4.6.4 Speicherverwaltung 4-38
4.7 Reihungen (Arrays) 4-48
4.7.1 Die Größe einer Reihung 4-50
4.7.2 Reihungen sind nur Zeiger 4-51
4.7.3 Reihungen als Rückgabewert 4-52
4.7.4 Funktionen höherer Ordnung für Reihungen 4-54
4.7.5 Sortieren von Reihungen mit bubblesort 4-56
4.8 Arbeiten mit Texten 4-65
4.9 Typsumme mit Unions 4-68
4.10 Rekursive Daten 4-72
4.10.1 Listen 4-72
4.11 Binäre Suchbäume 4-93
4.12 Praktikumsaufgabe 4-96

5 Programmiersprachen **5-1**
5.1 Syntax und Semantik 5-1
5.1.1 Syntax 5-1
5.1.2 kontextfreie Grammatik 5-2
5.1.3 Erweiterte Backus-Naur-Form 5-9
5.1.4 Parser 5-11
5.1.5 Parsstrategien 5-11
5.1.6 Semantik 5-15
5.2 Geschichte von Programmiersprachen 5-15

6 Bibliotheken **6-1**

6.1 Eigene Bibliotheken bauen 6-1

6.1.1 dynamische Bibliotheken 6-1

6.2 Kleines Beispiel einer GUI-Bibliothek 6-2

7 Schlußkapitel **7-1**

7.1 weitere Tricks des Präprozessors 7-1

7.1.1 Auskommentierung 7-1

7.1.2 Parameterisierte Makros 7-2

7.2 register, extern, auto Variablen 7-2

7.3 der Kommaoperator 7-2

7.4 Labels und Goto 7-2

7.4.1 Tailcall-Optimierung 7-3

7.4.2 Ergebnis akkumulieren 7-3

A C Operatoren **A-1**

B primitive Typen **B-1**

B.1 4.1 Integer Data Types B-1

B.2 4.2 Floating Point Data Types B-2

Begriffsindex B-3

Programmverzeichnis B-4

Kapitel 1

Einführung

1.1 Erste Programme

Im Zuge dieses Kurse wollen wir zunächst einmal alle Überlegungen theoretischer Natur auf später verschieben und direkt mit dem ersten Programm anfangen. Hierzu betrachten wir zunächst das leere Programm, das erst einmal nichts tut:

DasLeereProgramm.c

```
1 int main(){
2   return 0;
3 }
```

Programme der Programmiersprache C sind in einer Textdatei zu speichern, deren Dateiendung `.c` ist. Obiges Programm ist also in einer Textdatei namens `DasLeereProgramm.c` zu speichern. Um Textdateien zu schreiben und zu speichern benötigt man ein weiteres Programm, einen Editor. Es gibt viele Editoren, mit denen man dies bewerkstelligen kann. Es reichen hierzu primitivste Editorprogramme. Unter dem Windows-Betriebssysteme findet sich in der Regel das Editorprogramm `notepad`. Ein schwer zu bedienender jedoch auch sehr mächtiger und auf allen Plattformen verfügbarer Editor ist das Programm `vi`. Auf Linux-Betriebssystemen findet sich zumeist das Editorprogramm `kate`. Ich persönlich benutze seit Jahrzenten das Editorprogramm `emacs`. Wählen Sie auf Ihrer Platform einen einfachen Texteditor und schreiben und speichern mit ihm das obige Programm: `DasLeereProgramm.c`.

Obwohl das Programm nichts tut, besteht es schon immerhin aus drei Zeilen. Man hätte das Programm auch in einer Zeile schreiben können als:

AuchDasLeereProgramm.c

```
1 int main(){return 0;}
```

Ob und wo man Zeilenumbrüche macht, ist im Prinzip für die Programmiersprache C egal[1]. Trotzdem hat man sich auf Konventionen verständigt, wann man eine neue Zeile beginnt und wann man eine Zeile einrückt. Diese Konventionen erlauben es einen menschlichen Leser Programme besser zu verstehen. Deshalb ist das Beachten solcher Konventionen in den meisten

[1] Es gibt tatsächlich Programmiersprachen, in denen der Beginn einer neuen Zeile eine Bedeutung hat, in C aber nicht.

Firmen Pflicht; und auch in unserem Programmierkurs werden wir ein paar wenige Konventionen verbindlich machen. Die erste Konvention lautet:

> **Wann immer die Programmiersprache C verlangt, das Code zwischen einer öffnenden und einer schließenden geschweifte Klammer zu stehen hat, ist der Code zwischen den Klammern ein Stückchen weiter einzurücken. Nach der öffnenden Klammer ist eine neue Zeile zu beginnen. Die schließende Klammer steht auf einer neuen Zeile.**
>
> **In wieweit eingerückt wird, ist einheitlich zu handhaben.**

Die Programme dieses Skriptes halten sich weitgehendst an diese Regel. Es wird dabei jeweils um zwei Leerzeichen eingerückt.[2]

Das Programm `AuchDasLeereProgramm.c` verletzt diese Konvention.[3]

Auch wenn die beiden obigen Programme noch nichts tun, wollen wir sie einmal ausführen. Um ein Programm auf dem Computer ausführen zu können, muss es in eine Form übersetzt werden, die der Computer versteht. Der Programmtext, wie er mit dem Texteditor geschrieben wurde, kann nicht direkt vom Computer ausgeführt werden. Er ist erst in eine Folge von Nullen und Einsen zu übersetzen, die vom Betriebssystem aus gestartet und als Prozessorbefehle direkt den Prozessor des Computers zum Ausführen gegeben werden können. Das Übersetzen eines Programmtextes in ein ausführbares Programm unternimmt ein weiteres Programm, der Übersetzer für eine bestimmte Programmiersprache. Dieser wird auf Englisch *compiler* bezeichnet. Auf Deutsch wird eher von einem *Übersetzer* als von einem *Kompilierer* gesprochen.[4] Meist benutzt man auf Deutsch auch das englische Wort *compiler*.

Wir müssen also unseren obigen Programmtext, den wir mit einem Texteditor geschrieben haben, einen Kompilierer für die Programmiersprache C geben, damit wir ein ausführbares Programm erhalten. Den Programmtext den wir mit dem Editor geschrieben haben, bezeichnen wir als Quelltext.

Für die Programmiersprache C gibt es eine ganze Reihe von Kompilierern. Im Laufe dieses Kurses werden wir den Kompilierer `gcc` von `gnu` benutzen. Sein Vorteil ist, dass er für unterschiedliche Rechnertypen und Betriebssysteme zur Verfügung steht. Wir können somit unseren Quelltext mit der `gcc`-Version für Windows in ein ausführbares Programm übersetzen lassen, das unter dem Windows-Betriebssystemen läuft und den gleichen Quelltext mit der `gcc`-Version für Linux in ein ausführbares Programm übersetzen lassen, das auf dem Linux-Betriebssystem läuft. Allerdings lassen sich die ausführbaren Programme nicht austauschen. Das Windowsprogramm kann nicht auf Linux gestartet werden, und umgekehrt.

Um jetzt ein C-Programm mit dem Kompilierer `gcc` zu übersetzen, öffnen wir eine Kommandozeile, gehen mit dem `cd`-Befehl des Betriebssystems zum Wechseln in ein anderes Verzeichnis, in das Verzeichnis, in dem die Quelltextdatei gespeichert wurde. Dann rufen wir dort den Kompilierer `gcc` auf, geben mit `-o` gefolgt von einen beliebigen Namen den Namen an, den das ausführbare Programm haben soll, und geben schließlich noch an, welche Quelltextdatei übersetzt werden soll.

Anschließen gibt es das erzeugte ausführbare Programm, das ausgeführt werden kann.

Alles in allem erhalten wir folgende kleine Sitzung auf der Kommandozeile:

---

[2] Es wird bei den Beispielprogrammen bei Einzeilern davon abgewichen.

[3] Und würde bei einer Benotung deshalb Abzüge bekommen.

[4] Während sich im Französischen das Wort *compilateur* durchgesetzt hat.

```
sep@pc305-3:~> cd fh/c/student/src
sep@pc305-3:~/fh/c/student/src> gcc -o dasLeereProgramm DasLeereProgramm.c
sep@pc305-3:~/fh/c/student/src> ./dasLeereProgramm
sep@pc305-3:~/fh/c/student/src>
```

Man sieht nicht sehr viel bei dieser Ausführung. Der Aufruf des Kompilierers mit `gcc -o dasLeereProgramm DasLeereProgramm.c` führt zu keiner weiteren Ausgabe auf dem Bildschirm. Das ist auch gut so, denn wenn der Aufruf des Kompilerers zu Ausgaben auf der Kommandozeile führt, bedeutet das zumeist, dass etwas faul ist, nämlich der Quelltext Fehler enthält, aufgrund derer der Kompilierer kein ausführbares Programm erzeugen kann. Es ist gut sich mit solchen Fehlermeldungen frühzeitig auseinander zu setzen, denn in der Praxis werden wir sehr oft Fehler machen und manchmal verzweifelt nach einer Lösung suchen.

Schauen wir uns einmal an, was alles passiert, wenn wir in unser erstes leeres Programm Fehler einbauen.

Zunächst sei die schließende Klammer am Ende des Programms vergessen:

DasFalscheLeereProgramm1.c

```
1 int main(){
2   return 0;
```

Versuchen wir dieses zu übersetzen, so beschwert sich der Kompilierer über einen syntaktischen Fehler:

```
sep@pc305-3:~/fh/c/student/src> gcc -o leeresProg DasFalscheLeereProgramm1.c
DasFalscheLeereProgramm1.c: In function ‘int main()’:
DasFalscheLeereProgramm1.c:2: error: syntax error at end of input
sep@pc305-3:~/fh/c/student/src>
```

Ihm fehlt etwas am Ende des Programms.

Als nächstes sei einmal die Zeile `return 0;` die `0` weggelassen:

DasFalscheLeereProgramm2.c

```
1 int main(){
2   return;
3 }
```

```
sep@pc305-3:~/fh/c/student/src> gcc DasFalscheLeereProgramm2.c
DasFalscheLeereProgramm2.c: In function ‘int main()’:
DasFalscheLeereProgramm2.c:2: error: return-statement with no value, in
   function declared with a non-void return type
sep@pc305-3:~/fh/c/student/src>
```

Auch dieses führt zu einen Fehlermeldung. Diese ist für uns noch schwer zu verstehen. Eine Funktion, die nicht *void* sei, soll einen Wert mit dem `return` liefern. Später werden wir sehen, was ein *void*-Funktion ist.

Ein weiterer simpler Fehler ist, das Semikolon am Ende des `return`-Befehls zu vergessen

DasFalscheLeereProgramm3.c

```
1 int main(){
2   return 0
3 }
```

Hier meldet uns der Compiler, dass ihm irgendetwas vor der schließenden Klammer zu fehlen scheint.

```
sep@pc305-3:~/fh/c/student/src> gcc -Wall  DasFalscheLeereProgramm3.c
DasFalscheLeereProgramm3.c: In function ‘int main()’:
DasFalscheLeereProgramm3.c:3: error: syntax error before ‘}’ token
sep@pc305-3:~/fh/c/student/src>
```

Leider kommt der Compiler nicht auf die Idee, uns auf das fehlenden Semikolon explizit hinzuweisen.

Eine weitere interessante Fehlermeldung erhalten wir, wenn wir mit etwas zu viel norddeutschen Blut in den Adern, die Funktion `main` zu `moin` umbenennen:

DasFalscheLeereProgramm4.c

```
1 int moin(){
2   return 0;
3 }
```

Nun werden wir in einer Fehlermeldung auf das Nichtvorhandensein der Funktion `main` aufmerksam gemacht.

```
sep@pc305-3:~/fh/c/student/src> gcc -o leeresProg DasFalscheLeereProgramm4.c
/usr/lib/gcc-lib/i586-suse-linux/3.3.3/../../../crt1.o(.text+0x18): In function ‘_start’:
../sysdeps/i386/elf/start.S:98: undefined reference to ‘main’
collect2: ld returned 1 exit status
sep@pc305-3:~/fh/c/student/src>
```

Das Interessante an dieser Fehlermeldung ist, dass sie nicht auf einen syntaktischen Fehler hinweist, sondern auf das Fehlen einer Fuktionalität. Wie wir später sehen werden, kommt dieser Fehler auch nicht direkt vom Compiler, sondern einem kleinen Freund des Compilers, dem Linker.

Es ist schließlich an der Zeit auch Programme zu schreiben, die den ersten sichtbaren Effekt erzielen. Hierzu wollen wir einen Text auf der Kommandozeile ausgeben. Beliebige Texte können in C in Anführungszeichen eingeschlossen geschrieben werden. Es gibt eine eingebaute Funktion namens `printf`, der solche Texte übergeben werden können, damit sie die Texte auf der Kommandozeile ausgibt. Um diese Standardfunktion benutzen zu können, muß eine entsprechende Bibliothek für standard Ein- und Ausgabe, in das Programm mit eingebunden werden.

Das einfachste Programm, welches eine Ausgabe auf den Bildschirm tätigt, sieht damit wie folgt aus:

HalloFreunde.c

```
1 #include <stdio.h>
2 int main(){
3   printf("Hallo Freunde!");
4   return 0;
5 }
```

Die anfängliche Zeile `#include <stdio.h>` bindet die Standardbibliothek zur Ein- und Ausgabe ein. In der Funktion `main` wird vor dem Befehl `return` ein weiterer Befehl ausgeführt: der

Aufruf der Funktion `printf`, mit dem Text `Hallo Freunde!` als Argument. Argumente werden Funktionen in runden Klammern übergeben.

Texte werden in C als Zeichenketten in Anführungszeichen eingeschlossen. Diese Zeichenketten werden in Programmiersprachen als `String` bezeichnet.[5]

In der Regel dürfen Zeichenketten keinen Zeilenumbruch beinhalten. Das folgende Programm führt zu einem syntaktischen Fehler.

HalloIllja1.c

```
1 #include <stdio.h>
2 int main(){
3   printf("Hallo Freunde!
4           Hallo Illja!");
5   return 0;
6 }
```

Der Compiler beschwert sich mit einer ganzen Reihe von Fehlern

```
HalloIllja1.c:3:10: missing terminating " character
HalloIllja1.c: In function ‘main’:
HalloIllja1.c:4: error: parse error before "Hallo"
HalloIllja1.c:4:23: missing terminating " character
HalloIllja1.c:3:10: missing terminating " character
HalloIllja1.c: In function ‘main’:
HalloIllja1.c:4: error: parse error before "Hallo"
HalloIllja1.c:4:23: missing terminating " character
```

Will man trotzdem einen String aus Layout- oder Platzgründen auf mehrere Zeilen verteilen, so ist am Ende der Zeile ein rückwärtiger Schrägstrich anzubringen:

HalloIllja2.c

```
1 #include <stdio.h>
2 int main(){
3   printf("Hallo Freunde!\
4           Hallo Illja!");
5   return 0;
6 }
```

Dieses Programm übersetzt, hat aber bei der Ausgabe keinen Zeilenumbruch:

```
sep@pc305-3:~/fh/c/student> bin/HalloIllja2
Hallo Freunde!          Hallo Illja!sep@pc305-3:~/fh/c/student>
```

Möchte man, das in einer Zeichenkette ein Zeilenumbruch auftritt, so ist dieses mit einer sogenannten Fluchtsequenz zu markieren. Fluchtsequenzen bestehen in einem String aus einem rückwärtigen Schrägstrich und einem Buchstaben. Für einen Zeilenumbruch ist es die Sequenz `\n`. Um eine Ausgabe mit Zeilenumbrüchen zu erzielen, ist also in einem String die entsprechende Fluchtsequenz einzufügen:

[5] Nicht zu verwechseln mit einem knappen Teil Unterwäsche für junge Damen, wie einmal ein Student von mir bemerkte.

HalloIllja3.c

```
1 #include <stdio.h>
2 int main(){
3   printf("Hallo Freunde!\nHallo Illja!\n");
4   return 0;
5 }
```

Dieses Programm gibt dann die folgende Ausgabe:

```
sep@pc305-3:~/fh/c/student> bin/HalloIllja3
Hallo Freunde!
Hallo Illja!
sep@pc305-3:~/fh/c/student>
```

Wir haben bisher die Funktion `printf` immer genau einmal aufgerufen. In einem C Programm können mehrere Befehle nacheinander folgen und werden dann von oben nach unten abgearbeitet.

LichtAusSpotAn.c

```
1 #include <stdio.h>
2 int main(){
3   printf("Hallo Freunde!\n");
4   printf("Hallo Illja!\n");
5   printf("Licht aus!\n");
6   printf("Spot an!\n");
7   return 0;
8 }
```

Bisher haben wir nur Zeichenketten in Form eines Textes auf der Konsole ausgegeben. Die Funktion `printf` in C ist ein ziemlich vertrackter und mächtiger Befehl. Um eine ganze Zahl mit ihr ausgeben zu können, übergebe man der Funktion zwei Argumente: zunächst eine Zeichenkette, in der mit den Folge `%i` markiert wird, wo die ganze Zahl stehen soll, und dann nach einem Komma, die Zahl die auszugeben ist.

ErsteAntwort.c

```
1 #include <stdio.h>
2 int main(){
3   printf("Die Antwort lautet: ");
4   printf("%i",42);
5   return 0;
6 }
```

Dieses Spielchen läßt sich beliebig weitertreiben und so in einer Zeile mehrere Zahlen innerhalb eines Textes integriert ausgeben.

MehrAntworten.c

```
1 #include <stdio.h>
2 int main(){
3   printf("Die Antwort: %i wenn man %i und %i verdoppelt.\n",42,17,4);
4   return 0;
5 }
```

Nachdem wir nun einen Fuß in der Tür haben, werden wir in unserer Vorlesung in Kapitel 2 die Grundbausteine zur Strukturierung von Programmen kennenlernen: Ausdrücke, Funktionen und zusammengesetzte Befehle. Anschließend in Kapitel 3 wird es dann Zeit, sich einige Gedanken über die Softwareentwicklung zu machen. Kapitel 4 ist dann vollständig Daten gewidmet. In weiteren Kapiteln werden dann die gelernten Konzepte an Beispielprogrammen und durch Benutzung von Bibliotheken vertieft.

Aufgabe 1 Stellen Sie sich den anderen Kommilitonen und dem Dozenten Ihrer Praktikumsstunde vor. Nennen Sie dabei grob Ihre Vorkenntnisse und äußern Sie Ihre Erwartungen an die Veranstaltung.

Aufgabe 2 Beantworten Sie den Fragebogen über Vorkenntnisse und geben diesen dem Dozenten des Praktikums.

Aufgabe 3 Starten Sie den C-Compiler mit dem Befehl `gcc --version` auf der Kommandozeile.

Aufgabe 4 Compilieren und starten Sie die Programme aus Kapitel 1.

Aufgabe 5 Schreiben Sie ein erstes eigenes C Programm, das Ihren Namen auf der Kommandozeile ausgibt. Übersetzen Sie das Programm auf der Kommandozeile und führen Sie es aus.

Kapitel 2

Grundkonzepte

Im letzten Kapitel wurde ein Anfang gemacht. Wir können nun Programme schreiben, die Ausgaben auf der Kommandozeile machen. Nun wollen wir aber auch endlich Programme schreiben, die Rechnungen durchführen und das Ergebnis dieser Rechnungen ausgeben.

2.1 Operatoren

In der Programmiersprache C gibt es die Rechenoperatoren, wie wir sie aus der Grundschule bereits gewohnt sind.

2.1.1 arithmetische Grundrechenarten

Die einfachsten Operatoren sind Grundrechenarten. Ebenso wie wir es gewohnt sind, werden die Operatoren in C zwischen die beiden Operanden geschrieben. Man spricht von Infixoperatoren. Wie wohl kaum anders zu erwarten, kennt C die vier Grundrechenarten: `+, -, *, /`.

EinsPlusEins.c

```
1 #include <stdio.h>
2 int main(){
3   printf("%i\n",1+1);
4   return 0;
5 }
```

Seit dem Mittelalter ist es üblich eine manchmal etwas verwirrende Konvention zu berücksichtigen, in der die Operatoren der Punktrechnung stärker binden, als die Operatoren der Strichrechnung. Auch C folgt dieser Konvention, so dass das folgende Programm die Zahl 25 als Ergebnis ausgibt.

PunktVorStrich.c

```
1 #include <stdio.h>
2 int main(){
3   printf("%i\n",17+4*2);
4   return 0;
5 }
```

Wünscht man diese Konvention zu durchbrechen, so lassen sich Unterausdrücke in C explizit klammern. Das folgende Programm gibt endlich die ersehnten Antwort `42` aus:

Geklammert.c

```
1 #include <stdio.h>
2 int main(){
3   printf("%i\n",(17+4)*2);
4   return 0;
5 }
```

Die Division ist mit Sicherheit der problematischste Operator. Bisher rechnen wir nur mit ganzen Zahlen. Was ist die Hälfte einer ganzen Zahl? Desweiteren ist die Division nicht definiert für den Teiler `0`. Was passiert, wenn wir trotzdem versuchen, durch `0` zu teilen?

Division.c

```
1 #include <stdio.h>
2 int main(){
3   printf("%i\n",84/2);
4   printf("%i\n",85/2);
5   printf("%i\n",-4/2);
6   printf("%i\n",1/0);
7   return 0;
8 }
```

Die Division im ganzzahligen Bereich führt zu einen ganzzahligen Ergebnis. Tatsächlich bricht das obige Programm schließlich ab, wenn es dazu kommt, durch die Zahl `0` zu teilen.

```
sep@pc305-3:~/fh/c> ./student/bin/Division
42
42
-2
Gleitkomma-Ausnahme
sep@pc305-3:~/fh/c>
```

C kennt noch einen weiteren Operator, der sich mit der Division beschäftigt. Dies ist der sogenannte Modulo-Operator. Er wird durch das Symbol `%` bezeichnet. Er hat als Ergebnis den ganzzahligen Rest, der beim Teilen des ersten Operanden durch den zweiten Operanden übrig bleibt.

Modulo.c

```
1 #include <stdio.h>
2 int main(){
3   printf("%i\n",84%2);
4   printf("%i\n",85%2);
5   printf("%i\n",-4%2);
6   printf("%i\n",1%0);
7   return 0;
8 }
```

Auch der Modulo-Operator führt bei `0` als zweiten Operanden zu einen Fehler. Es wäre ja der Rest, der bleibt, wenn durch `0` geteilt wird.

2.1.2 Vergleichsoperatoren

C stellt weitere Operatoren zur Verfügung, die uns aus der Schule sicher bekannt sind. Dabei geht es darum, zwei Zahlen zu vergleichen. Hierzu kennt C die Operatoren `<, >, ==, !=, <=, >=` mit den naheliegenden Bedeutungen.

Die Frage ist allerdings, was ist das Ergebnis einer Vergleichsoperation. Der Vergleich ist entweder wahr oder falsch. Schön wäre es, wenn C Werte für wahr oder falsch kennen würde, so wie über 90% aller Programmiersprachen. Das erstaunliche ist, dass C solche Werte nicht kennt, und eine wahre Aussage mit der Zahl `1` und eine falsche Aussage mit der Zahl `0` ausdrückt.

GleichUndUngleich.c

```
1  #include <stdio.h>
2  int main(){
3    printf("(17+4)*2==42: %i\n",(17+4)*2==42);
4    printf("(17+4)*2==25: %i\n",(17+4)*2==25);
5    printf("(17+4)*2!=42: %i\n",(17+4)*2!=42);
6    printf("(17+4)*2!=25: %i\n",(17+4)*2!=25);
7    printf("(17+4)*2>17+4*2: %i\n",(17+4)*2>17+4*2);
8    printf("(17+4)*2<17+4*2: %i\n",(17+4)*2<17+4*2);
9    printf("(17+4)*2>=17+4*2: %i\n",(17+4)*2>=17+4*2);
10   printf("(17+4)*2<=17+4*2: %i\n",(17+4)*2<=17+4*2);
11   return 0;
12 }
```

Das Programm führt zur entsprechenden Ausgabe:

```
sep@pc305-3:~/fh/c> gcc student/src/GleichUndUngleich.c
sep@pc305-3:~/fh/c> ./a.out
(17+4)*2==42: 1
(17+4)*2==25: 0
(17+4)*2!=42: 0
(17+4)*2!=25: 1
(17+4)*2>17+4*2: 1
(17+4)*2<17+4*2: 0
(17+4)*2>=17+4*2: 1
(17+4)*2<=17+4*2: 0
sep@pc305-3:~/fh/c>
```

Fast alle anderen Programmiersprachen sind sich des fundamentalen Nutzen von Wahrheitswerten bewusst und kennen dafür einen eigenen Typen mit Namen `bool` oder `boolean`. Dieser Typ hat stets die zwei Werte `true` und `false`. In C müssen wir, wenn wir einen expliziten Typ für Wahrheitswerte benutzen wollen, diesen selbst definieren, was wir später im Kurs auch tun werden.

Auch wenn es Wahrheitswerte in C explizit nicht gibt, so gibt es in C eine ganze Reihe von Programmkonstrukten, die so tun, als wäre ein Typ `boolean` mit zwei Wahrheitswerten vorhanden. Hierzu gehören die Konstrukte, um Verzweigungen und Wiederholungen im Programmfluss auszudrücken. Zusätzlich gibt es aber auch eine Reihe von Operatoren, die nur von zwei Zuständen für `true` und `false` ausgehen.

Auch wenn Wahrheitswerte in C, nichts anderes sind als die Zahlen `0` und `1`, sollte man sich bemühen, die Wahrheitswerte nicht als Zahlen zu benutzen und verwirrende Programme schreiben, wie z.B. das Folgende, in dem auf das Ergebnis einer Vergleichsoperation eine arithmetische Operation angewendet wird:

George Boole (* 2. November 1815 in Lincoln, England; † 8. Dezember 1864 in Ballintemple, Irland) war ein englischer Mathematiker (Autodidakt) und Philosoph.
Ursprünglich als Lehrer tätig, wurde er auf Grund seiner wissenschaftlichen Arbeiten 1848 Mathematikprofessor am Queens College in Cork (Irland).
Boole entwickelte 1847 den ersten algebraischen Logikkalkül für Klassen und Aussagen im modernen Sinn. Damit begründete er die moderne mathematische Logik. Booles Kalkül wird gelegentlich noch im Rahmen der traditionellen Begriffslogik interpretiert, obwohl Boole bereits in The Mathematical Analysis of Logic ausschließlich von Klassen spricht.
Boole arbeitete in einem kommutativen Ring mit Multiplikation und Addition, die den logischen Verknüpfungen *und* und *entweder...oder* entsprechen; in diesem booleschen Ring gilt zusätzlich zu den bekannten Rechenregeln die Regel aa=a, in Worten (a und a)=a.
In der Boole-Schule wurde die zum booleschen Ring gleichwertige boolesche Algebra entwickelt, die mit *und* und *oder* arbeitet. Sie ist heute in der Aussagenlogik und Mengenlehre bekannter und verbreiteter als der originale, rechnerisch elegantere boolesche Ring.
George Boole ist der Vater von Ethel Lilian Voynich.
Auf George Boole sind die booleschen Variablen zurückzuführen, die zu den Grundlagen des Programmierens in der Informatik zählen.

Abbildung 2.1: **Wikipediaeintrag** (30. August 2006): George Boole

WahrPlusEins.c

```
1 #include <stdio.h>
2 int main(){
3   printf("(1==1)+1 =  %i\n",(1==1)+1);
4   return 0;
5 }
```

2.1.3 Logische Operatoren

Nun wollen wir aber auch mit Wahrheitswerten Rechnen. Als Grundoperationen stehen hierzu ein logisches *und* im Zeichen `&&` und ein logisches *oder* im Zeichen `||` zur Verfügung. Sie sind die in der formalen Logik mit dem Zeichen $\wedge$ für *und* und $\vee$ für *oder* bezeichnete Operatoren

Es gibt die beiden Operatoren auch in einer einfachen statt der doppelten Version. Angewendet auf reine Zahlen, die einen Wahrheitswert repräsentieren, haben beide Versionen, die doppelte und die einfache, das gleiche Endergebnis. Die beiden Versionen unterscheiden sich allerdings fundamental in der Art der technischen Ausführung. Vorerst werden wir nur die doppelte Version benutzen.

LogikOPs.c

```
1 #include <stdio.h>
2 int main(){
3   printf("1==1 && 1==2: %i\n",1==1 && 1==2);
4   printf("1==1 || 1==2: %i\n",1==1 || 1==2);
5   printf("1==1 & 1==2: %i\n",(1==1) & (1==2));
6   printf("1==1 | 1==2: %i\n",(1==1) | (1==2));
7   return 0;
8 }
```

Ein weiterer Operator für Wahrheitswerte in C ist die Negation, ein klassisches *nicht*. Es dreht einen Wahrheitswert um, macht aus *wahr* ein *falsch* und umgekehrt. Dieser Operator wird in C durch ein Ausrufezeichen ausgedrückt. In der formalen Logik wird das Zeichen ¬ für die Negation benutzt.

Negation.c

```
1 #include <stdio.h>
2 int main(){
3   printf("!(1==1): %i\n",!(1==1));
4   printf("!(1==2): %i\n",!(1==2));
5   return 0;
6 }
```

Im Gegensatz zu den bisherigen Operatoren handelt es sich bei diesem Operator um einen einstelligen. Er hat nur einen Operanden, wohingegen die übrigen Operatoren bisher zweistellig waren. Sie hatten jeweils zwei Operanden.

2.1.4 Bedingungsoperator

C kennt auch einen Operator mit drei Operanden. Er besteht aus zwei einzelnen Zeichen, die als Trenner zwischen den Operanden stehen. Zunächst kommt der erste Operand, dann das Zeichen `?`, dann der zweite Operand, gefolgt vom Zeichen `:`, dem der dritte Operand folgt. Schematisch sehen die Ausdrücke dieses Operators wie folgt aus:

$cond\ ?\ alt_1\ :\ alt_2$

Das Ergebnis dieses Operators wird wie folgt berechnet: Der erste Operand wird zu einem Wahrheitswert ausgewertet. Wenn dieser *wahr* ist, so wird der zweite Operand als Ergebnis ausgewertet, wenn er *falsch* ist, wird der dritte Operand als Ergebnis ausgewertet.

SoOderSo.c

```
1 #include <stdio.h>
2 int main(){
3   printf((17+4)*2==42?"tatsaechlich gleich\n":"unterschiedlich\n");
4   return 0;
5 }
```

Der Bedingungsoperator ist unser erstes Konstrukt, um Verzweigungen auszudrücken. Da der Bedingungsoperator auch einen Wert errechnet, können wir diesen benutzen, um mit ihm weiterzurechenen. Der Bedingungsoperator kann also tatsächlich ineinandergesteckt werden:

Signum1.c

```
1 #include <stdio.h>
2 int main(){
3   printf("signum(42) = %i\n",(42>0)?1:((42<0)?-1:0));
4   return 0;
5 }
```

Hier wird zunächst geschaut, ob die Zahl 42 größer als 0 ist. Ist dieses der Fall wird die Zahl 1 ausgegeben, ansonsten wird weitergeschaut, ob die Zahl 42 kleiner 1 ist. Hier wird im Erfolgsfall die Zahl -1 ausgegeben. Wenn beides nicht der Fall war, wird die Zahl 0 ausgegeben.

Zugegebener Maßen ist dieser Ausdruck schon schwer zu lesen. Wir werden später bessere Konstrukte kennenlernen, um verschiedene Fälle zu unterscheiden.

2.2 Funktionen

Im letzten Abschnitt haben wir die wichtigsten Operatoren in C kennengelernt. Mit diesen lassen sich im Prinzip schon komplexe Berechnungen durchführen. Allerdings können uns die theoretischen Informatiker beweisen, dass wir mit den Operatorausdrücken allein noch nicht mächtig genug sind, um alle erdenklichen berechenbaren mathematischen Funktionen berechnen zu können. Hierzu Bedarf es noch dem Konzept der Funktionen. Betrachten wir hierzu eine Funktion im klassischen mathematischen Sinn: eine Funktion hat eine Reihe von Argumenten und berechnet für die Argumente ein Ergebnis. Wichtig ist, dass eine Funktion immer für die gleichen Argument ein und dasselbe Ergebnis auf deterministische Weise errechnet. Die Anzahl der Argumente einer Funktion wird als ihre Stelligkeit bezeichnet.

Im folgenden Beispiel wird die Funktion `quadrat` definiert. Sie hat ein Argument. Als Ergebnis gibt sie das Quadrat dieser Zahl zurück. Nachdem diese Funktion definiert wurde, benutzen wir sie in der Funktion `main` mehrfach, jeweils mit anderen Argumenten.

Quadrat.c

```
1  #include <stdio.h>
2  int quadrat(int x){
3    return x*x;
4  }
5
6  int main(){
7    printf("%i\n",quadrat(5));
8    printf("%i\n",quadrat(10));
9    printf("%i\n",quadrat(0));
10   printf("%i\n",quadrat(-12));
11   return 0;
12 }
```

Wichtig bei der Definition einer Funktion ist, der `return`-Befehl. Eine Funktion berechnet ein Ergebnis. Dieses Ergebnis ist der Ausdruck, der hinter dem Befehlswort `return` steht.

C wertet jetzt jedesmal wenn eine Funktion benutzt wird, diesen Aufruf aus und ersetzt ihn quasi durch sein Ergebnis, nämlich den Ausdruck, der hinter dem Befehlswort `return` steht.

In Programmiersprachen werden die Argumente einer Funktion als Parameter bezeichnet. Bei der Definition der Funktion spricht man von den formalen Parametern, in unserem Falle das `x`. Bei der Definition der formalen Parameter ist vor seinem Namen, sein Typ zu schreiben. Bisher arbeiten wir nur mit ganzen Zahlen, die in C mit `int` bezeichnet werden.

Beim Aufruf der Funktion `quadrat` wird nun das `x` durch einen konkreten Ausdruck, z.B. die Zahl `5` ersetzt.

In der Wahl des Namens für einen formalen Parameter ist der Programmierer weitgehendst frei. Dieser Name existiert nur innerhalb des Funktionsrumpfes. Verschiedene Funktionen können Parameter gleichen Namens haben. Diese Parameter sind dann trotzdem vollkommen unabhängig.

Wir können in C beliebig viele Funktionen definieren, und auch `main` ist eine Funktion, die bisher immer parameterlos war. Wir können, nachdem eine Funktion einmal definiert ist, sie beliebig oft und an fast beliebig vielen Stellen aufrufen, sogar mehrfach verschachtelt:

QuadratUndDoppel.c

```
1  #include <stdio.h>
2  int quadrat(int x){
3    return x*x;
4  }
5  int doppelt(int x){
6    return x*x;
7  }
8
9  int main(){
10   printf("%i\n",doppelt(quadrat(5)));
11   return 0;
12 }
```

Hier wird in der Funktion `main` zunächst die Funktion `quadrat` mit der Zahl `5` aufgerufen, das Ergebnis (die `25`) wird dann als Argument der Funktion `doppelt` benutzt.

Innerhalb einer Funktion können auch andere Funktionen aufgerufen werden.

SummeDerQuadrate.c

```
1  #include <stdio.h>
2  int quadrat(int x){
3    return x*x;
4  }
5
6  int summeDerQuadrate(int x,int y){
7    return quadrat(x)+quadrat(y);
8  }
9
10 int main(){
11   printf("%i\n",summeDerQuadrate(3,4));
12   return 0;
13 }
```

In diesem Programm ruft die zweistellige Funktion `summeDerQuadrate` für jedes ihrer Argumente die Funktion `quadrat` auf, und addiert anschließend die Ergebnisse.

Die erste Zeile (bis zur öffnenden Klammer) einer Funktion wird als Signatur bezeichnet. Sie beinhaltet den Rückgabetyp, den Namen und die Argumente der Funktion. Alles innerhalb der geschweiften Klammern wird als Rumpf bezeichnet.

Die Berechnungen, die eine Funktion vornimmt, können beliebig komplexe Ausdrücke sein:

Signum2.c

```
1  #include <stdio.h>
2  int signum(int i){
3    return (i>0)?1:((i<0)?-1:0);
4  }
```

```
5  int main(){
6    printf("signum(42) = %i\n",signum(42));
7    printf("signum(-42) = %i\n",signum(-42));
8    return 0;
9  }
```

Aufgabe 6 Schreiben Sie eine Funktion, mit zwei ganzzahligen Parametern. Die Rückgabe soll die größere der beiden Zahlen sein. Testen Sie die Funktionen in einer Hauptfunktion mit mindesten drei Aufrufen.

Lösung

Greater.c

```
1  #include <stdio.h>
2  int greater(int x,int y){
3    return x>y?x:y;
4  }
5
6  int main(){
7    printf("-17, 8: %i\n",greater(-17,8));
8    printf("8, -8: %i\n",greater(8,-8));
9    printf("8, 8: %i\n",greater(8,8));
10   return 0;
11 }
```

Aufgabe 7 In der formalen Logik ist ein häufiger Operator die Implikation $a \rightarrow b$. Hierfür gibt es in C keinen eigenen Operator. Implementieren Sie eine Funktion
`int implication(int a,int b)`,
die die Implikation aus beiden Argumenten berechnet. Die Implikation $a \rightarrow b$ ist dabei definiert als: $\neg a \vee b$.

Schreiben Sie ein paar kleine Tests ihrer Funktion.

Implication.c

```
1  #include <stdio.h>
2  int implication(int a,int b){
3   return !a || b;
4  }
5
6  int main(){
7    printf("implication(1==0,1>2): %i\n",implication(1==0,1>2));
8    printf("implication(1>2,1==0): %i\n",implication(1>2,1==0));
9    return 0;
10 }
```

2.2.1 Seiteneffekte

Bisher waren alle unsere Funktionen streng im mathematischen Sinne. Sie haben nichts gemacht, außer ihr Ergebnis zu berechnen. Das Ergebnis war deterministisch nur von den Argumenten

abhängig. Wir können aber auch Funktionen schreiben, die nebenbei noch etwas anderes machen. Bisher haben wir noch nicht viel Möglichkeiten etwas zu programmieren, außer Ausgaben auf die Kommandozeile zu geben. Wir können Funktionen so schreiben, dass sie nicht nur ihr Ergebnis berechnen, sondern zuvor noch eine Ausgabe auf der Kommandozeile machen.

Seiteneffekt.c

```
1  #include <stdio.h>
2  int positiv(int x){
3    printf("teste ob %i positiv ist.\n",x);
4    return x>0;
5  }
6
7  int main(){
8    printf("%i\n",positiv(6)&&positiv(5));
9    printf("%i\n",positiv(6)||positiv(5));
10   printf("%i\n",positiv(-6)&&positiv(5));
11   printf("%i\n",positiv(-6)||positiv(5));
12   printf("%i\n",positiv(6)&positiv(5));
13   printf("%i\n",positiv(6)|positiv(5));
14   printf("%i\n",positiv(-6)&positiv(5));
15   printf("%i\n",positiv(-6)|positiv(5));
16   return 0;
17 }
```

Wenn eine Funktion nicht nur ein Ergebnis berechnet, sondern quasi nebenher noch etwas anderes macht, so spricht man von einem Seiteneffekt. Seiteneffekte scheinen zunächst nicht besonders tragisch zu sein, sofern es sich nur um ein paar Ausgaben handelt. Wir werden in Kürze sehen, dass eine Funktion den Wert von Speicherstellen als Seiteneffekt ändern kann. Je mehr Seiteneffekte ein Programm enthält, umso schwerer wird es, das Programm zu verstehen. Deshalb sollte man Seiteneffekte möglichst bewusst einsetzen. Aus der Problematik heraus, dass Seiteneffekte ein Programm komplizierter zu verstehen machen, wurden Programmiersprachen entwickelt, in denen keine Seiteneffekte programmiert werden können. Solche Sprachen sind funktionale Sprachen wie *Haskell*[PHA+97] und *Clean*[Pv97]. Auch die Sprache ML[MTH90] ist weitgehendst Seiteneffektfrei. ML erlebt derzeit eine gewisse Renaissance, da Microsoft es für die *Dot Net* Architektur unter den Namen F# implementiert hat.

Betrachten wir die Ausgabe des obigen Programms einmal genau, so können wir etwas über die Auswertung der verschiedenen logischen Operatoren lernen:

```
sep@pc305-3:~/fh/c/student> ./a.out
teste ob 6 positiv ist.
teste ob 5 positiv ist.
1
teste ob 6 positiv ist.
1
teste ob -6 positiv ist.
0
teste ob -6 positiv ist.
teste ob 5 positiv ist.
1
teste ob 6 positiv ist.
teste ob 5 positiv ist.
1
```

```
teste ob 6 positiv ist.
teste ob 5 positiv ist.
1
teste ob -6 positiv ist.
teste ob 5 positiv ist.
0
teste ob -6 positiv ist.
teste ob 5 positiv ist.
1
sep@pc305-3:~/fh/c/student>
```

Die ersten zwei Zeilen der Ausgabe leuchten ein. Um herauszubekommen, ob beides die `6` und die `5` eine positive Zahl sind, muss für beide Zahlen die Funktion `positiv` ausgewertet werden. Beide Male kommt es als Seiteneffekt zu der Ausgabe auf der Kommandozeile.

Um aber zu testen, ob eine der Zahlen `6` und `5` eine positive Zahl ist, reicht es, wenn die Funktion `positiv` für das Argument `6` schon die `1` für *wahr* als Ergebnis zurück gegeben hat. Damit ist einer der beiden Operanden des *oder* bereits *wahr* und damit auch der gesamte Oderausdruck. Warum also noch testen, ob auch die `5` positiv ist? Und tatsächlich erhalten wir keinen Seiteneffekt, der angibt, dass auch die Funktion `positiv` für das Argument `5` ausgewertet wurde.

Ähnlich verhält es sich beim dritten Aufruf. Wenn der erste Operand des *und* schon nicht *wahr* ist, braucht man sich den zweiten gar nicht mehr anschauen.

Nun sieht man den Unterschied zwischen `&&` und `&` sowie zwischen `||` und `|`. Die einfachen Versionen werten beide Operanden aus und errechnen dann aus den Teilergebnissen der Operanden ein Gesamtergebnis. Die doppelten Versionen schauen sich zunächst den ersten Operanden an, und entscheiden, ob es notwendig ist für das Gesamtergebnis sich auch noch den zweiten Operanden anzuschauen. Die Ausgabe auf der Kommandozeile, die wir als Seiteneffekt in die Funktion `positiv` eingebaut haben, macht diesen Unterschied sichtbar.

Die einfache Version der logischen Operatoren werden auch als *strikt* bezeichnet, die doppelte Version als *nicht-strikt*.

Aufgabe 8 Schreiben Sie eine Funktion:

Condition.h

```
1 int condition(int cond,int altTrue, int altFalse);
```

Sie soll wenn der Parameter `cond` dem Wert *wahr* entspricht, den Parameter `altTrue` ansonsten den Parameter `altFalse` zurückgeben.

Führen Sie dann folgende Tests aus:

TestCondition.c

```
1 #include <stdio.h>
2 #include "Condition.h"
3
4 int trace(int x){
5   printf("zahl %i\n",x);
6   return x;
7 }
8
```

```
9  int main(){
10   printf("condition: %i\n"
11                  ,condition(trace(1),trace(17),trace(4)));
12   printf("operator:  %i\n",trace(1)?trace(17):trace(4));
13   return 0;
14 }
```

Was stellen Sie fest. Interpretieren Sie das Ergebnis.

Lösung

Die Implementierung ist natürlich die direkte Benutzung des Bedingungsoperators. In der Aufgabe benutze ich hier aus technischen bereits eine Headerdatei. Das sollen die Studenten noch nicht unbedingt machen, war ja noch nicht erklärt.

Condition.c

```
1 #include "Condition.h"
2 int condition(int cond,int altTrue, int altFalse){
3   return cond?altTrue:altFalse;
4 }
```

Interessant ist natürlich die Ausgabe:

```
sep@pc305-3:~/fh/c/tutor> ./testCondition
zahl 4
zahl 17
zahl 1
condition: 17
zahl 1
zahl 17
operator:  17
sep@pc305-3:~/fh/c/tutor>
```

Man stellt fest, dass ein Funktionsaufruf dazu führt, dass immer alle Argumente ausgewertet werden. Daher sehen wir die Ausgabe aller drei `trace` Aufrufe. Interessanter Weise werden die Argumente von rechts nach links ausgewertet.

Der Bedingungsoperator hingegen hat eine eingebaute Magie. Hier wird erst die Bedingung ausgewertet und dann nur noch der Parameter, der für das Gesamtergebnis benötigt wird. Nicht alle drei Operanden werden ausgewertet. Der Operator ist nicht-strikt implementiert, während die Funktion `condition` strikt ausgewertet wird.

2.2.2 Prozeduren

Manchmal will man gar keine Funktion schreiben, sondern ein mehrfach aufrufbares Stück Programmcode, das im Prinzip nur einen oder mehrere Seiteneffekt hat. Im Prinzip also Unterprogramme, die nichts Neues berechnen, sondern etwas machen, z.B. eine Mail verschicken, etwas in die eine Datenbank schreiben oder sonstige beliebige Aktionen. Bisher haben alle unsere Funktionen ein Ergebnis berechnet und zurückgegeben.

Wenn kein Ergebnis berechnet wird, so ist einer Funktion statt eines Rückgabetyps das Wort `void` voranzustellen.

printNumber.c

```
#include <stdio.h>
void printNumber(int x){
  printf("%i.\n",x);
}

int main(){
  printNumber(1);
  printNumber(17+4);
  printNumber((17+4)*2);
  return 0;
}
```

Solche Funktionen werden gerne als `void`-Funktionen bezeichnet. Sie brauchen keinen `return`-Befehl zum Abschluss.

In der Informatik werden solche Funktionen auch als *Prozeduren* bezeichnet. Besonders in der Programmiersprache Pascal[Wir] hat man deutlich schon syntaktisch zwischen Funktionen und Prozeduren unterschieden.

2.2.3 rekursive Funktionen

Sobald wir die Signatur einer Funktion oder Prozedur definiert haben, dürfen wir sie benutzen, sprich aufrufen. Damit ergibt sich eine sehr mächtige Möglichkeit der Programmierung. Wir können Funktionen bereits in ihren eigenen Rumpf aufrufen. Solche Funktionen werden rekursiv genannt. *Recurrere* ist das lateinische Wort für zurücklaufen. Eine rekursive Funktion läuft während ihrer Auswertung wieder zu sich selbst zurück.

Damit lassen sich wiederholt Programmteile ausführen. Das folgende Programm wird z.B. nicht müde, uns mit dem Wort `hallo` zu erfreuen.

HalloNerver.c

```
#include <stdio.h>
void halloNerver(){
  printf("hallo\n");
  halloNerver();
}

int main(){
  halloNerver();
  return 0;
}
```

Die Funktion `main` ruft die Funktion `halloNerver` auf. Diese druckt einmal das Wort `hallo` auf die Konsole und ruft sich dann selbst wieder auf. Dadurch wird wieder `hallo` auf die Konsole geschrieben und so weiter. Wir haben ein endlos laufendes Programm. Tatsächlich endlos? Lassen sie es mal möglichst lange auf ihren Rechner laufen.

Was zunächst wie eine Spielerei anmutet, kann verfeinert werden, indem mit Hilfe eines Arguments mitgezählt wird, wie oft die Prozedur bereits rekursiv aufgerufen wurde:

HalloZaehler.c

```
1  #include <stdio.h>
2  void halloZaehler(int i){
3    printf("hallo %i\n",i);
4    halloZaehler(i+1);
5  }
6
7  int main(){
8    halloZaehler(1);
9    return 0;
10 }
```

Auch dieses Programm läuft endlos.

Mit dem Bedingungsoperator haben wir aber ein Werkzeug, um zu verhindern, dass rekursive Programme endlos laufen. Wir können den Bedingungsoperator nämlich benutzen, um zu unterscheiden, ob wir ein weiteres Mal einen rekursiven Aufruf vornehmen wollen, oder nicht mehr.

Hierzu ein kleines Programm, in dem die rekursive Prozedur eine Zahl als Parameter enthält. Zunächst druckt die Prozedur den Wert des Parameters auf der Kommandozeile aus, dann unterscheidet die Prozedur. Wenn der Parameter den Wert `0` hat, dann wird nur noch das Wort `ende` ausgedruckt, ansonsten kommt es zu einen rekursiven Aufruf. Für den rekursiven Aufruf wird der Parameter eins kleiner genommen, als beim Originalaufruf.

CountDown.c

```
1  #include <stdio.h>
2  void countDown(int i){
3    printf("hallo %i\n",i);
4    i==0?printf("ende\n"):countDown(i-1);
5  }
6
7  int main(){
8    countDown(10);
9    return 0;
10 }
```

Damit verringert sich in jedem rekursiven Aufruf der Parameter um eins. Wenn er anfänglich positiv war, wird er schließlich irgendwann einmal `0` sein. Dann verhindert der Bedingungsoperator einen weiteren rekursiven Aufruf. Die Auswertung endet. Man sagt, das Programm terminiert.

In diesem Zusammenhang gibt es mehrere interessante Ergebnisse aus der theoretischen Informatik. Mit der Möglichkeit rekursive Funktionen zu schreiben und mit der Möglichkeit über den Bedingungsoperator zu entscheiden, wie weiter ausgewertet wird, hat eine Programmiersprache die Mächtigkeit, dass mit ihr alle berechenbaren mathematischen Funktionen programmiert werden können.

Allerdings haben wir dafür auch einen Preis zu zahlen: wir können jetzt Programme schreiben, die nicht terminieren. Und das muss auch so sein, denn nach Ergebnissen der Theoretiker, muss es in einer Programmiersprache, die alle berechenbaren mathematischen Funktionen ausdrücken kann, auch Programme geben, die nicht terminieren.

Nach soviel esoterisch anmutenden Ausflügen in die Theorie, wollen wir nun auch eine rekursive Funktion schreiben, die etwas interessantes berechnet. Hierzu schreiben wir einmal die Fakultätsfunktion, die mathematisch definiert ist als:

$$fac(n) = \begin{cases} 1 & \text{für } n = 0 \\ n * fac(n-1) & \text{für } n > 1 \end{cases}$$

Diese Definition läßt sich direkt in ein C Programm umsetzen:

Fakultaet.c

```
1  #include <stdio.h>
2  int fac(int i){
3    return i==0?1:i*fac(i-1);
4  }
5
6  int main(){
7    printf("fac(5) = %i\n",fac(5));
8    return 0;
9  }
```

Aufgabe 9 Nehmen Sie das Programm zur Berechnung der Fakultät und testen sie es mit verschiedenen größeren Argumenten.

Lösung

Fakultaet2.c

```
1  #include <stdio.h>
2  int fac(int i){
3    return i==0?1:i*fac(i-1);
4  }
5
6  int main(){
7    printf("fac(5) = %i\n",fac(5));
8    printf("fac(10) = %i\n",fac(10));
9    printf("fac(15) = %i\n",fac(15));
10   printf("fac(20) = %i\n",fac(20));
11   printf("fac(20000) = %i\n",fac(20000));
12   return 0;
13 }
```

Die Ausgabe zeigt, wie schnell die Werte für die Fakultät wachsen, so dass eine `int`-Zahl nicht mehr ausreicht, das Ergebnis zu speichern:

```
sep@pc305-3:~/fh/c/tutor> bin/Fakultaet2
fac(5) = 120
fac(10) = 3628800
fac(15) = 2004310016
fac(20) = -2102132736
fac(20000) = 0
sep@pc305-3:~/fh/c/tutor>
```

Aufgabe 10 (1 Punkt) Im Bestseller *Sakrileg (der Da Vinci Code)* spielen die Fibonaccizahlen eine Rolle. Für eine natürliche Zahl n ist ihre Fibonaccizahl definiert durch:

$$f(n) = \begin{cases} n & \text{für } n \leq 1 \\ f(n-1) + f(n-2) & \text{für } n > 1 \end{cases}$$

Programmieren Sie eine Funktion `int fib(int n)`, die für eine Zahl, die entsprechende Fibonaccizahl zurückgibt.

Geben Sie in der Hauptfunktion die ersten 20 Fibonaccizahlen aus.

Lösung

Tatsächlich ist die Lösung ein Einzeiler, es muß ja nur die Formel aus der Spezifikation direkt abgeschrieben werden.

Fib.c

```c
#include <stdio.h>

int fib(int n){
  return  n<=1?n:(fib(n-1)+fib(n-2));
}

int main(){
  printf("fib(0): %i\n",fib(0));
  printf("fib(1): %i\n",fib(1));
  printf("fib(2): %i\n",fib(2));
  printf("fib(3): %i\n",fib(3));
  printf("fib(4): %i\n",fib(4));
  printf("fib(5): %i\n",fib(5));
  printf("fib(6): %i\n",fib(6));
  printf("fib(7): %i\n",fib(7));
  printf("fib(8): %i\n",fib(8));
  printf("fib(9): %i\n",fib(9));
  printf("fib(10): %i\n",fib(10));
  printf("fib(11): %i\n",fib(11));
  printf("fib(12): %i\n",fib(12));
  printf("fib(13): %i\n",fib(13));
  printf("fib(14): %i\n",fib(14));
  printf("fib(15): %i\n",fib(15));
  printf("fib(16): %i\n",fib(16));
  printf("fib(17): %i\n",fib(17));
  printf("fib(18): %i\n",fib(18));
  printf("fib(19): %i\n",fib(19));
  return 0;
}
```

2.3 Variablen

Bisher haben wir uns der Programmierung von einer sehr mathematischen funktionalen Seite genähert. Ein Grundkonzept fast aller Programmiersprachen leitet sich aus unserem tatsächlichen Rechnermodell her. Ein Computer hat einen Speicher, der in verschiedene Speicherzellen eingeteilt ist. In den Speicherzellen sind Zahlen gespeichert. In eine Speicherzelle können neue Werte hineingeschrieben werden und sie können auch wieder ausgelesen werden.

Die meisten Programmiersprachen bilden dieses Modell ab, indem sie das Konzept der Variablen haben. Eine Variable steht dabei für eine Speicherzelle, in der Werte geschrieben und später wieder ausgelesen werden können.

2.3.1 Zuweisung

Um eine Variable zu definieren ist zunächst zu definieren, welche Art von Daten dort hinein geschrieben werden sollen. Bisher arbeiten wir nur mit ganzen Zahlen und schreiben deshalb `int`. Dann folgt ein frei wählbarer Name für die Variable. Schließlich kann in die Variable mit einem Gleichheitszeichen ein Wert hinein geschrieben werden. Dieses wird dann als Zuweisung bezeichnet.

Im folgenden Programm wird eine Variable `x` deklariert, der sogleich der Wert `0` zugewiesen wird. Anschließend wird die Variable benutzt. Sie wird manipuliert, indem ihr neue Werte zugewiesen werden.

Var1.c

```
1  #include <stdio.h>
2
3  int main(){
4    int x=0;
5    printf("x = %i\n",x);
6    x=2;
7    printf("x = %i\n",x);
8    x=x*x;
9    printf("x = %i\n",x);
10   return 0;
11 }
```

Interessant ist die Zeile 8. Hier wird der Variablen `x` ein neuer Wert zugewiesen. Dieser neue Wert berechnet sich aus dem Quadrat des alten Werts der Variablen `x`.

Wir sind natürlich in der Zahl der benutzen Variablen nicht beschränkt. Das folgende Programm benutzt zwei Variablen.

Var2.c

```
1  #include <stdio.h>
2
3  int main(){
4    int x=17;
5    int y=4;
6    printf("x+y = %i\n",x+y);
7    x=x+y;
```

```
8   printf("x = %i\n",x);
9   printf("y = %i\n",y);
10  y=x*2;
11  printf("y = %i\n",y);
12  return 0;
13 }
```

Variablen können natürlich dazu benutzt werden, um das Ergebnis einer Funktion zu speichern:

VarQuadrat.c

```
1  #include <stdio.h>
2  int quadrat(int x){
3    return x*x;
4  }
5
6  int main(){
7    int y = quadrat(quadrat(2));
8    printf("%i\n",y);
9    return 0;
10 }
```

Bisher haben wir direkt, nachdem eine Variable eingeführt wurde, ihr einen Wert zugewiesen. Man kann das auch in zwei Schritten machen. Erst eine (oder gleich mehrere) Variablen deklarieren, und sie dann benutzen, indem Ihnen Werte zugewiesen werden.

Var3.c

```
1  #include <stdio.h>
2
3  int main(){
4    int x,y;
5    x=17;
6    y=4;
7    printf("x+y = %i\n",x+y);
8    x=x+y;
9    printf("x = %i\n",x);
10   printf("y = %i\n",y);
11   y=x*2;
12   printf("y = %i\n",y);
13   return 0;
14 }
```

Nur welchen Wert hat eine Variable, bevor sie einen Wert explizit zugewiesen bekommt?

Aufgabe 11 Schreiben Sie ein Testprogramm, in dem sie eine Variable auf der Kommandozeile ausgeben, bevor dieser Variablen ein Wert zugewiesen wurde.

Lösung

NoInit.c

```
1  #include <stdio.h>
2  int main(){
3    int x,y,z;
4    printf("%i %i %i\n",x,y,z);
5    return 0;
6  }
7
```

```
sep@pc305-3:~/fh/c/tutor> bin/NoInit
134513600 1073834176 1075174352
sep@pc305-3:~/fh/c/tutor
```

2.3.2 lokale und globale Variablen

Bisher haben wir alle Variablen innerhalb der Funktion `main` deklariert. Damit konnten diese Variablen auch nur innerhalb der Funktion `main` benutzt werden. Nach dem Schließen der geschweiften Klammer war die Variable dann wieder unbekannt. Solche Variablen, die nur innerhalb einer Funktion (oder noch kleineren Einheit) bekannt sind, werden als lokale Variablen bezeichnet. Man spricht vom Gültigkeitsbereich der Variablen. Außerhalb des lokalen Bereichs, in dem sie definiert werden, sind die Variablen nicht mehr gültig. Dort kann dann eine Variable gleichen Namens deklariert werden. Diese Variablen gleichen Namens sind vollkommen unabhängig voneinander. Sie bezeichnen tatsächlich unterschiedliche Stellen im Speicher. Zusätzlich werden diese lokalen Variable bei jedem Aufruf der Funktion wieder neu angelegt. Den Wert, den die Variablen am Ende einer Ausführung hatten, werden sie am Anfang der nächsten Ausführung nicht mehr kennen.

Folgendes Programm veranschaulicht dieses. Es gibt tatsächlich drei Variablen mit Namen `y`.

Lokal.c

```
1  #include <stdio.h>
2  int f1(){
3    int y = 1;
4    y=y+1;
5    return y;
6  }
7
8  int f2(){
9    int y = 2;
10   y=y*2;
11   return y;
12 }
13
14 int main(){
15   int y = f1();
16   printf("%i\n",y);
17   y = f1();
18   printf("%i\n",y);
19   y = f2();
```

```
20    printf("%i\n",y);
21    return 0;
22  }
```

das Programm führt zu folgender Ausgabe:

```
sep@pc305-3:~/fh/c/student> bin/Lokal
2
2
4
sep@pc305-3:~/fh/c/student>
```

Im Gegensatz zu lokalen Variablen, kann man außerhalb von Funktionen auch globale Variablen deklarieren. Diese sind dann tatsächlich für alle Funktionen benutzbar und existieren dann nur einmal.

Global.c

```
1  #include <stdio.h>
2  int y = 1;
3
4  int f1(){
5    y=y+1;
6    return y;
7  }
8
9  int f2(){
10   y=y*2;
11   return y;
12 }
13
14 int main(){
15   y = f1();
16   printf("%i\n",y);
17   y = f1();
18   printf("%i\n",y);
19   y = f2();
20   printf("%i\n",y);
21   return 0;
22 }
```

Im Gegensatz zum Programm `Local` haben wir jetzt folgende Ausgabe:

```
sep@pc305-3:~/fh/c/student> bin/Global
2
3
6
```

Aufgabe 12 Probieren Sie aus, ob es möglich ist, eine globale Variable noch einmal als lokale Variable zu deklarieren.

Lösung

Allreadydefined.c

```
1  int x;
2  void f(int x){}
3
4  int main(){
5    int x;
6    for(;;){
7        int x;
8    }
9  }
```

Die Benutzung von globalen Variablen ist gefährlich. Jedes Mal, wenn in einer Funktion eine globale Variable benutzt und verändert wird, handelt es sich um einen Seiteneffekt. Jetzt wird es richtig kompliziert, darüber Buch zu führen, wann eine Variable durch welchen Funktionsaufruf aus welchem Grund verändert wird. Um die Absicht einer globalen Variablen zu verstehen, muß man das ganze Programm betrachten. Jede Funktion muß betrachtet werden, um zu sehen, ob sie die globale Variable verändert. Damit widersprechen die globalen Variablen der Idee, unabhängige Komponenten zu schaffen. Funktionen sind, so lange sie keine globale Variablen benutzen, solche unabhängige Komponenten, wenn sie allerdings globale Variablen benutzen, dann sind die Funktionen nicht mehr unabhängige Komponenten.

Daher sollte man weitgehendst versuchen, auf globale Variablen zu verzichten. Nur in ganz bestimmten Situationen, um eine Information zu speichern, die wirklich global ist, sollte man auch gobale Variablen verwenden.

2.3.3 statische Variablen

Es gibt auch eine Möglichkeit, Variablen so zu deklarieren, dass sie über mehrere Aufrufe einer Funktion hinweg ihren Wert behalten. Hierzu ist bei der Deklaration der Variablen das Wort `static` zu schreiben. Damit wird eine Variable als statisch deklariert. Statische Variablen existieren nicht für jeden Programmdurchlauf neu, sondern einmal für alle Aufrufe einer Funktion.

Man veranschauliche sich das am folgenden Programm.

Static.c

```
1  #include <stdio.h>
2  int f1(){
3    static int y= 0;
4    y=y+1;
5    return y;
6  }
7
8  int main(){
9    int y = f1();
10   printf("%i\n",y);
11   y = f1();
12   printf("%i\n",y);
13   y = f1();
```

```
14    printf("%i\n",y);
15    return 0;
16  }
```

Wie man der Ausgabe entnehmen kann, erhöht jeder Aufruf der Funktion `f1` die statische Variable `y` um eins. Der initiale Wert `0`, der der Variablen bei der Deklaration mitgegeben wird, wird nur beim allerersten Aufruf der Funktion der Variablen `y` gegeben.

Es kommt zu folgender Ausgabe:

```
sep@pc305-3:~/fh/c/student> bin/Static
1
2
3
sep@pc305-3:~/fh/c/student>
```

Statisch bedeutet also, eine Variable für alle möglichen Aufrufe der Funktion. Nicht statisch hingegen: jeder Aufruf der Funktion bekommt die Variable wieder neu und frisch erstellt und mit einem initialen Wert belegt.

Damit kann mit Hilfe statischer Variablen Information von einem Aufruf der Funktion zum nächsten Aufruf der Funktion behalten werden. Die Aufrufe der Funktion sind damit nicht mehr unabhängig voneinander. Im obigen Beispiel zählt die statische Variable durch, wie oft die Funktion bereits aufgerufen wurde.

Damit widersprechen statische Variablen auch der Idee, möglichst unabhängige Einheiten zu schaffen. Mit einer statischen Variablen ist eine Funktion nicht mehr deterministisch: der gleiche Aufruf einer Funktion nacheinander führt jeweils zu unterschiedlichen Ergebnisse (und damit handelt es sich schon nicht mehr um eine Funktion im mathematischen Sinne).

Aufgabe 13 (1 Punkt) Um eine Folge von Zahlen, die eine mehr oder weniger zufällige Verteilung haben, zu erzeugen, kann man sich einer rekursiven Gleichung bedienen. Ein sehr einfaches Verfahren benutzt drei konstante Werte:

- den Faktor a.
- das Inkrement c.
- den Modul m.

Zusätzliche wähle man einen Startwert x_0.

Dann sei die Folge $(x_0, x_1, x_2, \ldots)$ definiert durch die Gleichung: $x_{i+1} = (a * x_i + c)$ mod m.

Damit gilt $x_i \geq 0$ und $x_i < m$ für $i \geq 0$. Man erhält also eine Folge von Zahlen zwischen 0 und m.

Schreiben Sie unter Benutzung statischer lokaler Variablen eine Funktion:
`int zufall();`
Sie soll bei jeden Aufruf ein weiteres Element der obigen Folge berechnen. Es seien dabei folgende Werte gewählt: $a = 21$, $c = 17$, $m = 100$ und $x_0 = 42$.

Testen Sie ihre Funktion `zufall` mit folgenden Test:

```
int schleife(int i){
  printf("%i ",zufall());
  return i>0?schleife(i-1):0;
}
int main(){
  schleife(100);
  return 0;
}
```

Lösung

Zufall.c

```
#include <stdio.h>
int zufall(){
  static int x=42;
  static int a=21; //factor
  static int c=17; //increment;
  static int m=100;//modul;
  x=(a*x+c)%m;
  return x;
}

int schleife(int i){
  printf("%i ",zufall());
  return i>0?schleife(i-1):0;
}
int main(){
  schleife(100);
  return 0;
}
```

2.3.4 Auch Parameter sind Variablen

Wir haben bisher den Zuweisungsbefehl für Variablen kennengelernt. Erstaunlicher Weise kann man auch den Parametern einer Funktion neue Werte zuweisen:

AssignParam.c

```
#include <stdio.h>
int f1(int x){
  x=2*x;
  return x;
}
```

Hier wird also ein Parameter auch als Variable benutzt. Was bedeutet das, wenn die Funktion mit einer Variablen als Argument aufgerufen wird. Wird damit das Argument im Aufruf auch verändert? Betrachten wir hierzu folgende Funktion `main`:

AssignParam.c

```
6  int main(){
7    int y = 42;
8    int z = f1(y);
9    printf("y = %i\n",y);
10   printf("z = %i\n",z);
11   return 0;
12 }
```

Was passiert mit der Varaiblen `y` beim Aufruf der Funktion `f1`? Verdoppelt sich auch ihr Wert oder nicht? Betrachten wir hierzu die Ausgabe:

```
sep@pc305-3:~/fh/c/student> ./bin/AssignParam
y = 42
z = 84
sep@pc305-3:~/fh/c/student>
```

Man sieht, dass sich der Wert von `y` durch den Aufruf `f1(y)` nicht verändert hat. Der Parameter `x` der Funktion `f1` verhält sich wie eine lokale Variable, die beim Aufruf den Wert, mit dem die Funktion aufgerufen wird erhält. Ob dieser Wert aus einer anderen Variablen ausgelesen wurde, spielt dabei keine Rolle. Man bezeichnet dieses Verhalten als Parameterübergabe per Wert. Die aufgerufene Funktion erhält nur den übergebenen Wert und hat keinerlei Zugriff auf eventuelle Variablen, aus denen dieser Wert ursprünglich stammt.

2.3.5 Fragen des Stils

Es gibt vielfältige Möglichkeiten mit Variablen in C zu arbeiten. Welche Empfehlung kann man dazu geben. Eine Empfehlung haben wir bereits kennen gelernt. Möglichst keine globalen Variablen zu benutzen. Ebenso sollte man weitgehendst versuchen auf statische Variablen zu verzichten. Auch sie widersprechen der Idee, eigenständige unabhängige Komponenten zu schaffen.

Wie sieht es mit Parametern aus? Soll man sich wie Variablen benutzen und ihnen unter Umständen einen neuen Wert zuweisen. Auch hiervon ist besser abzuraten. Ein Programm ist leichter zu lesen, wenn ein Parameter immer nur den Wert enthält, den er beim Funktionsaufruf bekommen hat.

Desweiteren: welcher Stil ist nun besser? Erst alle Variablen deklarieren und ihnen dann Werte zuweisen, also: `int x; ... ;x=42;`, oder aber gleich einen Anfangswert der Variablen mitgeben, wenn sie deklariert wird: `int x=42;`? Und wo soll man die Variable deklarieren? Ganz oben im Programm oder erst da, wo man sie zum ersten Mal braucht?

Die Antworten auf diese Fragen können sehr unterschiedlich ausfallen, je nachdem, wen sie fragen. Mehrere Jahrzehnte wurde gelehrt, Variablen schön übersichtlich alle am Anfang des Programms zu deklarieren, damit man einen genauen Überblick hat, mit welchen Variablen man arbeitet. Es gab Programmiersprachen, die dieses sogar zur Pflicht erhoben haben. Zusätzlich ist der klassische C Stil, eine Variable erst einmal zu deklarieren, und dann erst bei Bedarf ihr einen Wert zuzuweisen. Seit geraumer Zeit wurden diese Empfehlungen gerade umgedreht. Heute wird vielfach propagiert: Variablen so spät wie möglich deklarieren, so dass sie einen möglichst begrenzten lokalen Gültigkeitsbereich bekommen. Zusätzlich soll man Variablen dann direkt ihren Wert zuweisen. Die Idee ist: warum soll ich mich am Anfang eines Programms mit Variablen

belasten, die ich irgendwann nur kurz brauche, um ein Zwischenergebnis abzuspeichern. Am saubersten ist, wenn Variablen gar nicht variabel sind, sondern nachdem sie einmal ihren Wert bekommen haben, diesen Wert gar nicht mehr verändern.

2.4 Kontrollstrukturen

Bisher sind uns die wichtigsten Grundkonzepte, die auf ähnliche Wiese in fast allen Programmiersprachen wiederzufinden sind, begegnet:

- Ausdrücke zum Rechnen mit Zahlen und Wahrheitswerten.
- Funktionen, als Möglichkeit ein Programm in unabhängige Komponenten zu strukturieren.
- Variablen, um Werte zwischenzuspeichern und später im Programm wiederzuverwenden.

Damit sind die wichtigsten Grundkonzepte der Programmierung bereits bekannt. Über den Trick der rekursiven Funktionen können wir sogar schon dafür sorgen, dass bestimmte Programmteile wiederholt ausgeführt werden. Mit den Bedingungsoperator ist es uns möglich, aufgrund eines bool'schen Werts zu entscheiden, mit welchem Ausdruck weiter gearbeitet werden kann.

In klassischen imperativen Programmiersprachen wie C, gibt es eingebaute zusammengesetzte Befehle, die den Programmfluß, wann welcher Programmteil abgearbeitet wird kontrollieren. Diese wollen wir in diesem Abschnitt präsentieren. Erstaunlicher Weise basieren fast alle diese zusammengesetzten Befehle auf einer Bedingung über einen Wahrheitswert und das, obwohl es einen eigenen Datentypen für Wahrheitswerte in C überhaupt nicht gibt.

2.4.1 if Verzweigungen

Die `if`-Verzweigung ist ein Konstrukt, das erlaubt, einen bestimmten Programmteil nur auszuführen, wenn eine bestimmte Bedingung `wahr` ist.

Die `if`-Verzweigung besteht aus dem Wort `if`, dann folgt in runden Klammern Ausdruck, der einen Wahrheitswert berechnet, und in geschweiften Klammern die Befehle, die ausgeführt werden sollen, wenn der Warheitswert *wahr* entsprach:

If1a.c

```
1  #include <stdio.h>
2
3  void f(int x){
4    if (x>0) {
5      printf("x ist groesser als 0\n");
6    }
7  }
8
9  int main(){
10   f(42);
11   f(-42);
12   return 0;
13 }
```

Auf die geschweiften Klammern im if-Konstrukt, kann verzichtet werden, wenn anschließend nur genau ein Befehl folgt. Das war im letzten Beispiel der Fall. Es kann daher auch wie folgt geschrieben werden:

If1b.c

```
#include <stdio.h>

void f(int x){
  if (x>0) printf("x ist groesser als 0\n");
}

int main(){
  f(42);
  f(-42);
  return 0;
}
```

Man sollte ein bißchen vorsichtig sein, wenn man auf die geschweiften Klammern in einem if-Befehl verzichten möchten. Vielleicht fügt man irgendwann einen weiteren Befehl im Rumpf des if-Befehls hinzu. Wenn man dann nicht die Klammern hinzufügt, aber beide Befehle eingerückt hat, dann sieht es eventuell so aus, als gehörten beide Befehle in den Rumpf des if-Befehls, tatsächlich steht aber nur der erste von beiden im Rumpf des if-Befehls.

Natürlich lassen sich in einer `if`-Verzweigung auch abhängig von einem Wahrheitswert, Variablen verändern:

If2.c

```
#include <stdio.h>

int main(){
  int x=1;
  int y=0;
  if (x>0) {
    printf("x ist groesser als 0\n");
    y=2*x;
  }
  printf("y = %i\n",y);
  return 0;
}
```

Optional kann eine `if`-Verzweigung noch eine `else`-Klausel enthalten, hierin folgen die Befehle, die auszuführen sind, wenn die Bedingung nicht wahr war:

Else.c

```
#include <stdio.h>
void teste(int x){
  if (x>0) {
    printf("x ist groesser als 0\n");
  }else {
    printf("x ist kleiner oder gleich 0\n");
  }
```

```
8  }
9
10 int main(){
11   teste(5);
12   teste(0);
13   teste(-5);
14   return 0;
15 }
```

Mit der `if`-Verzwigung haben wir nun eine weitere Möglichkeit zu kontrollieren, ob eine rekursive Funktion terminiert oder einen weiteren rekursiven Aufruf vornimmt. So läßt sich jetzt ziemlich einfach programmieren, dass ein bestimmter Befehl mehrfach ausgeführt werden soll:

WiederholtHallo1.c

```
1  #include <stdio.h>
2  void wiederhole(int x){
3    printf("hallo\n");
4    if (x>0) {
5      wiederhole(x-1);
6    }
7  }
8
9  int main(){
10   wiederhole(5);
11   return 0;
12 }
```

Mit der `else`-Klausel lässt sich ebenso programmieren, was eine rekursive Funktion im terminierenden Fall noch machen soll.

WiederholtHallo2.c

```
1  #include <stdio.h>
2  void wiederhole(int x){
3    printf("hallo\n");
4    if (x>0) {
5      wiederhole(x-1);
6    }else{
7      printf("nun ist aber schluss\n");
8    }
9  }
10
11 int main(){
12   wiederhole(5);
13   return 0;
14 }
```

Die folgenden Version dieses Programms gibt zusätzlich jeweils noch aus, wie oft die Rekursion noch durchlaufen wird.

WiederholtHallo3.c

```
#include <stdio.h>
void wiederhole(int x){
  printf("hallo  %i\n",x);
  if (x>0) {
    wiederhole(x-1);
  }
}

int main(){
  wiederhole(5);
  return 0;
}
```

Die vorherigen Programme bestanden alle aus einer rekursiven Prozedur. Mit der `if`-Verwzeigung lassen sich auch wunderbar rekursive Funktionen schreiben, die ein bestimmtes Ergebnis berechnen. Folgendes Programm summiert z.B. die Zahlen von `1` bis zum Parameter `x` auf.

Summe1.c

```
#include <stdio.h>
int summe(int x){
  int result=0;
  if (x<=0) {
    result=0;
  }else {
    int untersumme = summe(x-1);
    result=x+untersumme;
  }
  return result;
}

int main(){
  printf("summe(5) = %i\n",summe(5));
  printf("summe(0) = %i\n",summe(0));
  printf("summe(100) = %i\n",summe(100));
  return 0;
}
```

Um die Arbeitsweise einer solchen rekursiven Funktion besser zu verstehen, können wir noch eine Ausgabe hinzufügen, die die jeweiligen Teilergebnisse auf der Kommandozeile ausgibt:

Summe2.c

```
#include <stdio.h>
int summe(int x){
  int result=0;

  if (x<=0) {
    result=0;
  }else {
```

```
8     int untersumme = summe(x-1);
9     result=x+untersumme;
10    printf("  berechne summe(%i) = %i\n",x,result);
11  }
12
13  return result;
14 }
15
16 int main(){
17   printf("summe(5) = %i\n",summe(5));
18   printf("summe(0) = %i\n",summe(0));
19   printf("summe(100) = %i\n",summe(100));
20   return 0;
21 }
```

Die Ausgabe zeigt schön, wie nacheinander für immer größer werdende Zahlen die Summe errechnet wird und dann benutzt wird, um die Summe für die nächst höhere Zahl zu berechnen:

```
sep@pc305-3:~/fh/c> ./student/bin/Summe2
  berechne summe(1) = 1
  berechne summe(2) = 3
  berechne summe(3) = 6
  berechne summe(4) = 10
  berechne summe(5) = 15
summe(5) = 15...
```

Aufgabe 14 Das rekursive Programm zur Berechnung der Summe $\text{sum}(x) = \sum_{i=1}^{n} i$ läßt sich mit ein wenig Mathematik vollkommen ohne Rekursion, **if**-Befehl oder Schleife schreiben. Hierzu benötigt man die Summenformel aus der Mathematik. Suchen Sie diese aus dem Internet oder einer Formelsmmlung und schreiben sie einen Einzeiler für `int summe(int n);`.

Lösung

Die anzuwendende Formel ist:$\sum_{i=1}^{n} i = x * (x+1)/2$, also ist folgende Funktion zu schreiben:

SimpleSum.c

```
1  #include <stdio.h>
2
3  int sum(int x){ return x*(x+1)/2;}
4  int main(){
5    printf("sum(1)=%i\n",sum(1));
6    printf("sum(2)=%i\n",sum(2));
7    printf("sum(3)=%i\n",sum(3));
8    printf("sum(4)=%i\n",sum(4));
9    printf("sum(5)=%i\n",sum(5));
10   printf("sum(6)=%i\n",sum(6));
11   printf("sum(7)=%i\n",sum(7));
12   return 0;
13 }
```

unnötiges If

In Quelltext von Anfänger sieht man oft bei Funktionen, die einen Wahrheitswert zurückgeben den if-Befehl in folgender Weise benutzt.

NotLikeThis.c

```
1  #include <stdio.h>
2
3  int f(int x){
4    if (x>0) return 1;
5    else return 0;
6  }
7  int main(){
8    printf("%i\n",f(42));
9    return 0;
10 }
11
```

In der Funktion `f` wird geprüft, ob `x>0` ist. Wenn das Ergebnis davon *wahr* ist, also 1, dann soll *wahr* (also 1) zurückgegeben werden. Wenn `x>0` *falsch* (also 0) ist, dann soll *falsch* (also 0) zurückgegeben werden. Dann kann man aber gleich auf die if-Bedingung verzichten und direkt `x>0` zurückgeben. Das Programm ist besser zu schreiben als:

LikeThis.c

```
1  #include <stdio.h>
2  int f(int x){
3    return x>0;
4  }
5  int main(){
6    printf("%i\n",f(42));
7    return 0;
8  }
9
```

Aufgabe 15 Schreiben Sie das Programm `fakultaet` so um, dass Sie den Bedingungsoperator durch einen `if`-Befehl ausdrücken.

Lösung

Fakultaet3.c

```
1  #include <stdio.h>
2  int fac(int i){
3    if (i==0) return 1;
4    return i*fac(i-1);
5  }
6
7  int main(){
8    printf("fac(5) = %i\n",fac(5));
9    return 0;
10 }
```

2.4.2 Schleifen

Bisher kennen wir nur den Trick der Rekursion, um einen Programmteil mehrfach durchlaufen zu lassen. Die Terminierung der Rekursion haben wir dabei durch die `if`-Verzweigung oder durch den Bedingungsoperator gesteuert. Es gibt tatsächlich Programmiersprachen, in denen dieses der einzige Weg ist, bestimmte Programmteile wiederholt zu durchlaufen. In den meisten Programmiersprachen insbesondere in den sogenannten imperativen Sprachen, wozu C zu rechnen ist, gibt es eingebaute besondere zusammengesetzte Befehle, mit denen ohne eine Rekursion ausgedrückt werden kann, dass bestimmte Programmteile mehrfach auszuführen sind. Diese Befehle werden als Schleifen bezeichnet. Das Prinzip der Schleifen wird im Gegensatz zur Rekursion als Iteration bezeichnet. In C gibt es drei verschiedene Schleifenbefehle.

while-Schleife

Die wahrscheinlich gebräuchlichste Schleife ist die `while`-Schleife. Sie besteht aus dem Wort `while`, dem in runden Klammern ein Ausdruck folgt, der einen Wahrheitswert berechnet. Dieser Ausdruck wird als Schleifenbedingung bezeichnet. Schließlich folgt in geschweiften Klammern eine Folge von Befehlen, der sogenannte Rumpf der Schleife.

In folgendem ersten Beispiel ist die Schleifenbedingung rot und der Schleifenrumpf blau gedruckt:

CountDownWhile.c

```
1 #include <stdio.h>
2 int main(){
3   int x=10;
4   while (x>0){
5     printf("x = %i\n",x);
6     x=x-1;
7   }
8   return 0;
9 }
```

Für die Ausführung einer `while` Schleife wird zunächst die Bedingung ausgerechnet. Wenn diese nicht zu *wahr* ausgewertet wird, dann passiert gar nichts weiter. Wird die Bedingung allerdings zu *wahr* ausgewertet, so werden anschließend die Befehle des Rumpfes ausgeführt. Anschließend geht es wieder von vorne los. Es wird ein weiteres Mal das Ergebnis der Bedingung ausgerechnet und abhängig davon eine weiteres der Rumpf der Schleife und immer so weiter.

Ebenso wie bei einer Rekursion kann es auch bei einer Iteration dazu kommen, dass das Programm nicht terminiert. Dieses ist der Fall, wenn die Schleifenbedingung immer zu *wahr* ausgewertet wird, wie im folgenden Beispiel:

NonterminatingIteration.c

```
1 #include <stdio.h>
2 int main(){
3   while (1==1){
4     printf("im Schleifenrumpf\n");
5   }
6   return 0;
7 }
```

Die Bedingung `1==1` ist selbstverständlich immer *wahr*, und damit wird immer wieder der Rumpf der Schleife abgearbeitet. Damit eine Schleife terminiert, muß der Rumpf dafür sorgen, dass darin Variablen so verändert werden, dass irgendwann die Schleifenbedingung nicht mehr zu *wahr* ausgewertet wird. Hierzu muss eine eine Variable geben, die zur Auswertung der Schleifenbedingung benutzt wird und die ebenso im Schleifenrumpf benutzt wird. In unserem ersten Beispiel war dieses die Variable `x`. Es wurde darin so lange iteriert, bis diese Variable nicht mehr positiv war. Im Schleifenrumpf wurde die Variable um `1` verringert. Damit hat sie nach endlich vielen Durchgängen keinen positiven Wert mit und die Iteration terminiert.

Insofern sind Iterationen nur ein spezielles syntaktisches Konstrukt, um Rekursionen auszudrücken. Auch in einer Rekursion gibt es eine Variable, von der abhängt, ob ein weiterer rekursiver Durchlauf zu machen ist. Diese Variable ist in der Rekursion ein Parameter der Funktion. Der Wert des Parameters muss sich von rekusriven Durchlauf so verändern, dass die Terminierngsbedingung einer Rekursion irgendwann erreicht wird.

Somit läßt sich obige erstes Schleifenprogramm als Rekursion schreiben. Die Schleifenbedingung wandert hierbei in eine `if`-Verzweigung. Der Schleifenrumpf wird der Rumpf der `if`-Verzweigung. Statt die Variable `x` zu verringern, wird ein rekursiver Aufruf mit einem um `1` verringerten Parameter durchgeführt:

CountDownRecursive.c

```
1  #include <stdio.h>
2  void whileErsatz(int x){
3    if (x>0){
4      printf("x = %i\n",x);
5      whileErsatz(x-1);
6    }
7  }
8
9  int main(){
10   whileErsatz(10);
11   return 0;
12 }
```

Schleifen haben zumeist eine Variable, die steuert, wie oft die Schleife zu durchlaufen ist. Man spricht dabei auch von der Laufvariablen. Zusätzlich kann es noch weitere Variablen geben, in denen z.B. nach und nach ein zu berechnendes Ergebnis gespeichert wird.

Im folgenden Beispiel ist `x` die Laufvariable und in `y` wird das jeweilige Zwischenergebnis der eigentlichen Berechnnung gespeichert.

SummeWhile1.c

```
1  #include <stdio.h>
2  int main(){
3    int x=10;
4    int y=0;
5    while (x>0){
6      printf("x = %i y = %i\n",x,y);
7      y=y+x;
8      x=x-1;
9    }
10   return 0;
11 }
```

Machen wir uns einmal die Mühe, diese Schleife Stück für Stück von Hand nachzuvollziehen. Hierzu notieren wir einfach die Werte, die nach und nach die beiden Variablen `x` und `y` erhalten.

x	10	9	8	7	6	5	4	3	2	1	0
y	0	10	19	27	34	40	45	49	52	54	55

Die Ausgabe des Programms macht auch die Veränderung der beiden Variablen über die Zeit deutlich:

```
sep@pc305-3:~/fh/c> tutor/bin/SummeWhile1
x = 10 y = 0
x = 9 y = 10
x = 8 y = 19
x = 7 y = 27
x = 6 y = 34
x = 5 y = 40
x = 4 y = 45
x = 3 y = 49
x = 2 y = 52
x = 1 y = 54
sep@pc305-3:~/fh/c>
```

Bei der Wahl der Variablennamen sollten man versuchen schon deutlich zu machen, welche Variable die Laufvariable für die Bedingung ist und welche Ergebnisse der eigentlichen Anwendungslogik ist. In C werden häufig Variablennamen `i, j, k` für Laufvariablen benutzt. Für die Variablen der Anwendungslogik sollte man besser längere und sprechende Namen nehmen. Insbesondere bietet sich der Name `result` für Variablen an, in denen nach und nach das Ergebnis berechnet wird.

So ließe sich eine Funktion zur Berechnung der Summe, wie folgt schreiben:

SummeWhile2.c

```
1  #include <stdio.h>
2  int summe(int x){
3    int result=0;
4    int i = x;
5    while (i>0){
6      result=result+i;
7      i=i-1;
8    }
9    return result;
10 }
11
12 int main(){
13   printf("summe(5) = %i\n",summe(5));
14   printf("summe(0) = %i\n",summe(0));
15   printf("summe(100) = %i\n",summe(100));
16
17   return 0;
18 }
```

Aufgabe 16 Schreiben Sie die Fakultät jetzt, indem Sie die Funktion nicht über eine Rekursion, sondern über eine `while`-Schleife realisieren.

Lösung

FakIter.c

```
#include <stdio.h>
int fak(int x){
  int result=1;
  while (x>0){
    result=result*x;
    x=x-1;
  }
  return result;
}


int main(){
  printf("fak(0): %i\n",fak(0));
  printf("fak(1): %i\n",fak(1));
  printf("fak(2): %i\n",fak(2));
  printf("fak(3): %i\n",fak(3));
  printf("fak(4): %i\n",fak(4));
  printf("fak(5): %i\n",fak(5));
  return 0;
}

```

Aufgabe 17 Sie sollen eine Funktion zum Potenzieren realisieren:
`int potenz(int x,int y)`
Die Funktion soll x hoch y rechnen, also: potenz(x, y) = x^y. Implementieren Sie diese Funktion einmal rekursiv und einmal mit Hilfe einer `while`-Schleife und einmal mit Hilfe einer `for`-Schleife.

Lösung

Potenz.c

```
#include <stdio.h>

int potenz1(int x,int y){
  if (y==0) return 1;
  return x*potenz1(x,y-1);
}

int potenz2(int x,int y){
  int result=1;
  while (y>0){
    result=result*x;
    y=y-1;
  }
  return result;
}

```

```
17 int potenz3(int x,int y){
18   int result=1;
19   for (;y>0;y--){
20     result=result*x;
21   }
22   return result;
23 }
24
25 int main(){
26   printf("potenz1(3,9): %i\n",potenz1(3,9));
27   printf("potenz2(3,9): %i\n",potenz2(3,9));
28   printf("potenz2(3,9): %i\n",potenz3(3,9));
29   return 0;
30 }
31
```

for-Schleifen

Im letzten Abschnitt haben wir gesehen, dass im Rumpf der `while`-Schleife zwei Dinge zu geschehen haben:

- die Laufvariable muss so verändert werden, um irgendwann die Terminierung der Schleife zu gewährleisten.
- die eigentliche Berechnung für die Anwendungslogik ist durchzuführen.

Es gibt also im Rumpf der Schleife Code, zum Steuern der Anzahl der Schleifendurchläufe und Code zum Berechnen des Ergebnisses. Aus der Überlegung heraus, diese zwei Codeteile möglichst zu trennen, gibt es in C (und vielen Sprachen die darauf aufbauen, wie z.B. auch Java) eine besondere Schleife, die `for`-Schleife. Auch die `for`-Schleife hat einen Schleifenrumpf, allerdings sollte in ihrem Schleifenrumpf kein Code stehen, der steuert, wie oft die Schleife durchlaufen wird. Dieser Code soll möglichst komplett im sogenannten Schleifenkopf stehen Der Schleifenkopf ist in runden Klammern eingeschlossen und enthält drei durch Semikolon getrennte Teile:

- die Initialisierung der Laufvariablen
- die Schleifenbedingung
- die Veränderung der Schleifenvariable pro Schleifendurchgang

Im folgenden Beispiel sind die drei Teile der Schleifensteuerung farblich markiert.

CountDownFor.c

```
1 #include <stdio.h>
2 int main(){
3   int i;
4   for (i=10;i>0;i=i-1){
5     printf("i = %i\n",i);
6   }
```

```
7   return 0;
8 }
```

In der Ausführung der `for`-Schleife wird als erstes (und dann auch nicht mehr) die Initailisierung der Laufvariablen vorgenommen. Der rote Code im obigen Beispiel. Nun wird die Schleifenbedingung geprüft. Ist diese nicht *wahr*, so passiert nichts weiter. Ansonsten wird jetzt der Schleifenrumpf ausgeführt. Und dann erst der Code zur Veränderung der Schleifenvariablen. Im Beispiel oben mit grün markiert. Nun geht es wieder mit der Prüfung der Schleifenbedingung weiter.

Vielleicht läßt sich am einfachsten die Abarbeitung der `for`-Schleife nachvollziehen, wenn man sie in eine äquivalente `while`-Schleife umschreibt. Wir benutzen die gleichen farblichen Markierungen, wie in der `for`-Schleife:

CountDownWhile2.c

```
1  #include <stdio.h>
2  int main(){
3    int i;
4    i=10;
5    while (i>0){
6      printf("i = %i\n",i);
7      i=i-1;
8    }
9    return 0;
10 }
```

Man sieht gut, wie das Weiterschalten der Schleifenvariablen am Ende des Schleifenrumpfes geschieht, und die Initialisierung der Schleifenvariablen nur einmal vor der eigentlichen Schleife.

In neueren Programmiersprachen, sowie C++ und Java ist es möglich, die Laufvariable einer `for`-Schleife nur lokal direkt im Schleifenkopf zu deklarieren. Damit kann die Schleifenvariable nicht außerhalb der Schleife für andere Zwecke mißbraucht oder für eine weitere Schleife benutzt werden. Dieses entspricht der Idee, Variablen so lokal wie nur möglich zu deklarieren. Damit sähe das Programm wie folgt aus:

CountDownForNonC.c

```
1  #include <stdio.h>
2  int main(){
3    for (int i=10;i>0;i=i-1){
4      printf("i = %i\n",i);
5    }
6    return 0;
7  }
```

Leider entspricht das nicht dem Standard der Programmiersprache C wie sie unser Compiler benutzt und unser Compiler weist es auch mit einem Fehler zurück:

```
sep@pc305-3:~/fh/c/student> gcc src/CountDownForNonC.c
src/CountDownFor2.c: In function ‘main’:
src/CountDownFor2.c:3: error: ‘for’ loop initial declaration used outside C99 mode
sep@pc305-3:~/fh/c/student>
```

Wollen wir diese Art der Schleifenvariable benutzen, so müssen wir unseren Compiler auf der Kommandozeile beim Aufruf darauf hinweisen, dass wir nach dem C99 Standard programmieren wollen, dann verschwindet der Fehler:

```
sep@pc305-3:~/fh/c/student> gcc -std=c99 src/CountDownFor2.c
```

Ansonsten müssen wir Wohl oder Übel die Schleifenvariable vor der `for`-Schleife deklarieren, werden später in C++ und Java es aber erst im Kopf der Schleife machen, versprochen.

Bisher haben wir im Beispiel für die `for`-Schleife noch keine Berechnungen für ein Ergebnis vorgenommen, sondern nur als Seiteneffekt etwas auf der Kommandozeile ausgegeben. Nach all dem Vorausgesagten ist deutlich, wie sich das Programm zur Summenberechnung mit einer `for`-Schleife präsentiert:

SummeFor.c

```
1  #include <stdio.h>
2  int summe(int x){
3    int result=0;
4    int z;
5    for (z=1;x>=z;z=z+1){
6      result=result+z;
7    }
8    return result;
9  }
10
11 int main(){
12   printf("summe(5) = %i\n",summe(5));
13   printf("summe(0) = %i\n",summe(0));
14   printf("summe(100) = %i\n",summe(100));
15
16   return 0;
17 }
```

Interessanter Weise können von den drei Komponenten des Schleifenkkopfes durchaus auch welche leer sein. Dann steht da einfach nichts. Und eine `for`-Schleife mit leerer Initialisierung der Schleifenvariablen und leerer Weiterschaltung ist nichts anderes mehr als eine `while`-Schleife.

PseudoWhile.c

```
1  #include <stdio.h>
2  int main(){
3    int x=10;
4    for (;x>0;){
5      printf("x = %i\n",x);
6      x=x-1;
7    }
8    return 0;
9  }
```

do-while-Schleifen

Die `while`-Schleife (und als ihr Spezialfall auch die `for`-Schleife) prüfen beide ersteinmal eine Schleifenbedingung, bevor sie sich den Schleifenrumpf anschauen. Diese Art von Schleife wird daher auch als vorgeprüfte Schleife bezeichnet. Bevor der Schleifenrrumpf überhaupt in Betracht gezogen wird, wird ersteinmal die Bedingung geprüft. Ist diese anfangs nicht *wahr*, so wird der Rumpf kein einiziges Mal ausgeführt.

Im Gegensatz dazu gibt es auch noch eine besondere Schleife, die als nachgeprüft bezeichnet wird, die `do-while`-Schleife. Sie beginnt mit dem Wort `do`, dann folgt der Schleifenrumpf und nach dem Schleifenrumpf erst das Wort `while` mit der Schleifenbedingung. Bei der nachgeprüften Schleife, wird zunächst einmal der Schleifenrumpf einmal garantiert ausgeführt. Anschließend wird mit der Bedingung geprüft, ob ein weiterer Schleifendurchlauf zu bewerkstelligen ist.

DoWhile.c

```
1  #include <stdio.h>
2  int main(){
3    int x=10;
4    do {
5      printf("x = %i\n",x);
6      x=x-1;
7    } while (x>0);
8    return 0;
9  }
```

Viel Programmierer vernachlässigen die nachgeprüfte Schleife ein wenig und man sieht sie doch wesentlich seltener als die vorgeprüften Schleifen. Aber es soll auch wahre Fans der nachgeprüften Schleifen geben.

Eine typische Anwendung für die nachgerprüfte Schleife ist das Erfragen von Benutzereingaben. Hierzu gibt es eine quasi inverse Funktion zu `printf`, nämlich `scanf`, die nicht Ausgaben tätigt, sondern Eingaben abfragt. Es wird typischer Weise in einer vorgeprüften Schleife so lange nach einer Eingabe gefragt, bis eine verarbeitbare Zahl eingegeben wurde.

input.c

```
1  #include <stdio.h>
2
3  int f(int n){return n==0?1:n*f(n-1);}
4  int main(){
5    int x;
6    do{
7      printf("mit welcher Zahl soll gerechnet werden?\n");
8      scanf("%i",&x);
9    }while (x<0);
10   printf("f(x) = %i\n",f(x));
11   return 0;
12 }
13
```

Schleifen in Schleifen

Innerhalb des Schleifenrumpfs können beliebige Befehle stehen. Eine Schleife ist auch nur ein Befehl, demnach kann in einem Schleifenrumpf wieder eine Schleife stehen. Um eine solche verschachtelte Schleife zu verstehen, braucht man kein neues Wissen, sondern kann es aus dem Wissen über Schleifen ableiten. Ein zweidimensionaler Raum wird aufgebaut. Die äußere Schleife entspricht den Wertebereich der ersten, die innere dem Wertebereich der zweiten Dimension.

Dieses läßt sich an der Ausgabe des folgenden Programms gut ersehen:

ZweiDimensionaleSchleife.c

```
1  #include <stdio.h>
2  int main(){
3    int x;
4    for (x=0;x<10;x=x+1){
5      int y;
6      for (y=0;y<20;y=y+1){
7        printf("(x,y) = (%i,%i)\n",x,y);
8      }
9    }
10   return 0;
11 }
```

Zweidimensionale Räume begegnen einem bei der Programmierung natürlich permanent, bei Bildern und Grafiken, aber auch Spielbrettern. In diesen Fällen wird man immer zwei ineinander verschachtelte Schleifen benötigen.

Aufgabe 18 (1 Punkt) Schreiben Sie ein C Programm, das die folgenden Ausgabe auf der Kommandozeile hat:

```
sep@pc305-3:~/fh/c> ./a.out
.XXXXXXXXXXXXXXXXXX
X.XXXXXXXXXXXXXXXXX
XX.XXXXXXXXXXXXXXXX
XXX.XXXXXXXXXXXXXXX
XXXX.XXXXXXXXXXXXXX
XXXXX.XXXXXXXXXXXXX
XXXXXX.XXXXXXXXXXXX
XXXXXXX.XXXXXXXXXXX
XXXXXXXX.XXXXXXXXXX
XXXXXXXXX.XXXXXXXXX
XXXXXXXXXX.XXXXXXXX
XXXXXXXXXXX.XXXXXXX
XXXXXXXXXXXX.XXXXXX
XXXXXXXXXXXXX.XXXXX
XXXXXXXXXXXXXX.XXXX
XXXXXXXXXXXXXXX.XXX
XXXXXXXXXXXXXXXX.XX
XXXXXXXXXXXXXXXXX.X
XXXXXXXXXXXXXXXXXX.
sep@pc305-3:~/fh/c>
```

Lösung

Diagonal.c

```
#include <stdio.h>
void printDiagonale(){
  int x;
  int y;
  for (x=0;x<20;x++){
    for (y=0;y<20;y++){
      if (x==y){
        printf(".");
      }else printf("x");
    }
    printf("\n");
  }
}

int main(){
  printDiagonale();
  return 0;
}
```

Schleifen irregulär abbrechen

Wie oft eine Schleife durchlaufen wird, hängt von der Schleifenbedingung ab, aber leider nicht nur. Innerhalb einer Schleife kann der Befehl `break` benutzt werden, mit dem auch das Durchlaufen einer Schleife gesteuert werden kann. So kann man sogar eine while-Schleife schreiben, deren Bedingung immer *wahr* ist, die aber trotzdem terminiert. Der Befehl `break` kann beliebig innerhalb einer Schleife stehen, und führt bei seiner Ausführung zum sofortigen Abbruch der Schleife:

CountDownWhileBreak.c

```
#include <stdio.h>
int main(){
  int x=10;
  while (0==0){
    printf("x = %i\n",x);
    x=x-1;
    if (x<=0) break;
  }
  return 0;
}
```

Der Befehl `break` ist auch mit Vorsicht zu benutzen. Irgendwo im Schleifenrumpf kann er verstecken, dass die Schleife, unabhängig von der Schleifenbedingung, doch verlassen werden soll. Man umgeht damit die Schleifenbedingung. Die Schleifenbedingung suggeriert, dass sie steuert, wie oft die Schleife durchlaufen wird, und irgendwo im Rumpf wird etwas ganz anderes gemacht.

Ein akzeptabler Einsatz des Befehls `break` ist in echten Fehler- oder Susnahmesituationen. Normaler Weise steuert der Schleifenkopf, wie oft die Schleife durchlaufen wird, nur in besonderen Spezialfällen bricht die Schleife dann auch entgegen den Bedingungen im Schleifenkopf ab.

Schleifenrumpf temporär abbrechen

Ein noch brutalerer Befehl, der es erlaubt zu steuern, wie die Schleife durchlaufen wird, ist der Befehl `continue`. Er sorgt dafür, dass der Schleifenrumpf sofort verlassen wird, alle nachfolgenden Befehle des Rumpf werden nicht mehr ausgeführt. Die Schleife selbst wird aber nicht verlassen, sondern es wird mit der Schleifensteuerung weiter geprüft, ob der Rumpf ein weiteres Mal auszuführen ist.

Ein kleines Beispiel für den Befehl `continue`:

Continue.c

```
1  #include <stdio.h>
2  int main(){
3    int x=10;
4    while (x>=0){
5      printf("x = %i\n",x);
6      x=x-1;
7
8      if (x%2!=0){
9        continue;
10     }
11     printf("x ist eine ungerade Zahl\n");
12   }
13   return 0;
14 }
```

Der Nutzen des Befehls `continue` ist recht fraglich. Er macht Schleifen noch einmal komplexer zu verstehen. In meiner bisherigen Laufbahn als Programmierer habe ich ihn bisher noch kein einziges Mal benutzt, obwohl er es in Sprachen wie z.B. Java als altes C-Erbe geschafft hat.

Schleifen und return

Es gibt noch eine weitere Möglichkeit, eine Schleife zu verlassen; und dafür kann die Schleife gar nicht einmal etwas. Es handelt sich dabei nämlich nicht um einen schleifenspezifischen Befehl, sondern um den Befehl `return`, mit dem eine Funktion verlassen wird. Bisher haben wir jedes Mal den Befehl `return` als letzten Befehl des Rumpfes einer Funktion gehabt. Mit ihm wurde die Funktion verlassen und mit ihm wurde das Funktionsergebnis festgelegt. Der Befehl `return` kann aber tatsächlich an beliebiger Stelle eines Funktionsrumpfes stehen. Dann sorgt er auch dafür, dass die Berechnung des Funktionswertes beendet wird und kein weiterer Code im Funktionsrumpfs ausgeführt wird:

WhileReturn.c

```
1  #include <stdio.h>
2  int f(int x){
3    int i = x;
4    int result = 0;
```

```
5   while (0==0){
6     if (i==0) return result;
7     result= result+i;
8     i=i-1;
9   }
10  return result;
11 }
12
13 int main(){
14   printf("f(10) = %i\n",f(10));
15   return 0;
16 }
```

Man sieht, dass es schon mit den wenigen Konstrukten, die wir bisher kennengelernt haben, doch möglich ist, Programme zu schreiben, die in ihrer Ablaufsteuerung recht komplex sind. Die Programmiersprache C gibt dem Programmierer sehr viel Freiheiten, sich auszudrücken. Leider damit auch sehr viel Möglichkeiten sich vertrackt und kompliziert auszudrücken. Daraus machen sich manche Programmierer gar einen Sport und es gibt den sogenannten *obfuscated C contest*, in dem es darum geht, möglichst vertrackte und unverständliche Programme zu schreiben.

Normaler Weise möchte man aber verstanden werden und sollte sich mit einfachen Worten und kurzen Sätzen ausdrücken. Das gilt auch in der Programmierung. Auf undurchsichtige Schleifensteuerung sollte möglichst verzichtet werden.

2.4.3 switch Verzweigungen

Der letzte zusammengesetzte Befehl, den C kennt, ist eine besondere Art der Verzweigung, der `switch`-Befehl. Er erlaubt es, abhängig vom Wert einer Zahl nach verschiedenen Fällem, die konstant festgelegt sind, zu unterscheiden. Er beginnt mit dem Wort `switch`, in runden Klammern gefolgt von einem Ausdruck, der zu einer Zahl auswertet. Dann folgen in geschweiften Klammern eingeschlossen die verschiedenen Fälle. Jeder Fall beginnt mit dem Wort `case`. Dann kommt eine Konstante, die den Fall beschreibt, der hier behandelt wird. Nach einem Doppelpunkt folgen dann die spezifischen Befehle für diesen Fall.

Ein besonderer Fall wird mit dem Wort `default` bezeichnet. Er trifft immer zu, egal welchen Wert der in Frage kommende Ausdruck, nach dem die Fallunterscheidung vorgenommen wurde, hat.

Ein erstes kleines Beispiel für die `switch`-Anweisung:

Switch1.c

```
1  #include <stdio.h>
2
3  void f(int x){
4    switch (x) {
5      case 1: printf("eins ");
6      case 2: printf("zwei ");
7      case 3: printf("drei ");
8      case 4: printf("vier ");
9      case 5: printf("fuenf ");
10     case 6: printf("sechs ");
```

```
11      case 7: printf("sieben ");
12      case 8: printf("acht ");
13      case 9: printf("neun ");
14      case 0: printf("null ");
15      default: printf("keine Ziffer");
16    }
17    printf("\n");
18  }
19
20  int main(){
21    f(8);
22    f(3);
23    f(19);
24    return 0;
25  }
```

Dieses Programm führt nicht ganz zu der vom vernünftigen Menschenverstand erwartete Ausgabe:

```
sep@pc305-3:~/fh/c/student> ./Switch1
acht neun null keine Ziffer
drei vier fuenf sechs sieben acht neun null keine Ziffer
keine Ziffer
sep@pc305-3:~/fh/c/student>
```

Wie man sieht, wird zwar direkt zu dem Fall gesprungen, der für den Parameter `x` zutrifft, allerdings wird nicht nur dieser Fall ausgeführt, sondern alle folgenden Fälle auch noch. Wenn man nicht möchte, dass nachdem zu einem bestimmten Fall gesprungen wurde, auch noch alle folgenden Fälle auszuführen sind, so muss man am Ende eines Falls immer noch den uns bereits von den Schleifen bekannte Befehl `break` setzten:

Switch2.c

```
1   #include <stdio.h>
2
3   void f(int x){
4     switch(x) {
5       case 1: printf("eins ");break;
6       case 2: printf("zwei ");break;
7       case 3: printf("drei ");break;
8       case 4: printf("vier ");break;
9       case 5: printf("fuenf ");break;
10      case 6: printf("sechs ");break;
11      case 7: printf("sieben ");break;
12      case 8: printf("acht ");break;
13      case 9: printf("neun ");break;
14      case 0: printf("null ");break;
15      default: printf("keine Ziffer");break;
16    }
17    printf("\n");
18  }
19
```

```
20 int main(){
21   f(8);
22   f(3);
23   f(19);
24   return 0;
25 }
```

Dann Funktioniert das Programm so, wie man es doch tatsächlich erwartet hätte:

```
sep@pc305-3:~/fh/c/student> bin/Switch2
acht
drei
keine Ziffer
sep@pc305-3:~/fh/c/student>
```

Tatsächlich ist es ein häufiger Fehler, dass das `break` am Ende eines Falls vergessen wurde.

Aufgabe 19 (1 Punkt) Schreiben Sie eine Prozedur `alsText`, die eine Zahl als Argument erhält. Sie soll die Ziffern dieser Zahl als Text auf der Kommandozeile ausdrucken. Der Aufruf `alsText(-142)` soll zur Ausgabe `minus eins vier zwei` führen.

Hinweis: Schreiben Sie dabei folgende Hilfsfunktionen:
`int hoechsteZehnerPotenz(int i);`
Sie soll für einstellige Zahlen `1`, für zweistellige Zahlen `10`, für dreistellige Zahlen `100` usw. als Ergebnis haben.

`void printZiffer(int i);`
Hier gehen Sie davon aus, dass $0 \leq i \leq 9$. Diese Ziffer soll dann als Wort auf der Kommandozeile ausgegeben werden.

Lösung

AlsText.c

```
1  #include <stdio.h>
2  void printZiffer(int i){
3    switch (i) {
4      case 0: printf("null");break;
5      case 1: printf("eins");break;
6      case 2: printf("zwei");break;
7      case 3: printf("drei");break;
8      case 4: printf("vier");break;
9      case 5: printf("fünf");break;
10     case 6: printf("sechs");break;
11     case 7: printf("sieben");break;
12     case 8: printf("acht");break;
13     case 9: printf("neun");break;
14     default: ;
15   }
16 }
17
18 int hoechsteZehnerPotenz(int i){
```

```
19    int result = 1;
20    while (i>=10){
21      result=result*10;
22      i=i / 10;
23    }
24    return result;
25  }
26
27  void alsText(int i){
28    if (i<0){
29      printf("minus ");
30      i= -i;
31    }
32    for (int hoechsteStelle = hoechsteZehnerPotenz(i)
33        ;hoechsteStelle>0
34        ;hoechsteStelle=hoechsteStelle/10) {
35      printZiffer(i/hoechsteStelle);
36      printf(" ");
37      i=i%hoechsteStelle;
38    }
39    printf("\n");
40  }
41
42  void alsTextrec1(int i);
43
44  void alsTextrec(int i){
45    if (i==0){ printf("null");
46    }else if (i<0){
47      printf("minus");
48      alsTextrec1(-i);
49    }else alsTextrec1(i);
50    printf("\n");
51  }
52
53  void alsTextrec1(int i){
54    if (i!=0) {
55      alsTextrec1(i/10);
56      printf(" ");
57      printZiffer(i%10);
58    }
59  }
60
61  int main(){
62    alsText(-142);
63    alsText(1024);
64    alsText(0);
65    alsText(-0);
66    alsText((17+4)*2);
67
68    alsTextrec(-142);
```

```
69    alsTextrec(1024);
70    alsTextrec(0);
71    alsTextrec(-0);
72    alsTextrec((17+4)*2);
73    return 0;
74  }
```

2.5 Zuweisung und Operatoren

Wir haben dieses Kapitel mit Operatoren begonnen, die nach unserer Kenntnis von mathematischen Operatoren funktionierten. Operatorausdrücke hatten zumeist zwei Operanden. Aus diesen Operanden wurde ein Ergebnis des Operatorausdrucks berechnet. Die Operanden wurden dabei in keinster Weise berührt. Im Zusammenhang mit Variablen haben wir dann den Zuweisungsbefehl kennengelernt. Auf seiner linken Seite stand dabei eine Variable, der der Wert der rechten Seite zugewiesen wurde.

In C gibt es jetzt eine Reihe von seltsamen Operatoren, die zwei Dinge auf einmal machen: eine Operation ausrechnen und eine Variable verändern.

2.5.1 Zuweisungsoperatoren

Für fast alle Operatoren in C gibt es eine Version, die den Operator und eine Zuweisung vereint. Will man zu einer Variablen `x` etwa eine Zahl hinzuzählen, so ist zunächst die Addition durchzuführen `x+5`, um dann schließlich das Ergebnis der Variablen wieder zuzuweisen, also `x=x+5`. Diese Berechnung und anschließende Zuweisung kann man in C abkürzen mit Hilfe des Additionszuweisungsoperators `+=`. Obiger Ausdruck kann dann schließlich als `x+=5` geschrieben werden. Interessanter Weise hat `x+=5` das gleiche Ergebnis wie `x+5`. Zusätzlich hat der Ausdruck noch den Seiteneffekt, dass dieses Ergebnis der Variable `x` zugewiesen wird.

Ein paar Beispiele der Zuweisungsoperatoren für arithmetische Ausdrücke:

AssignOperation.c

```
1   #include <stdio.h>
2   int main(){
3     int x=42;
4     printf("x+=2 : %i\n",x+=2);
5     printf("x    : %i\n",x);
6     printf("x-=2 : %i\n",x-=2);
7     printf("x    : %i\n",x);
8     printf("x*=2 : %i\n",x*=2);
9     printf("x    : %i\n",x);
10    printf("x/=2 : %i\n",x/=2);
11    printf("x    : %i\n",x);
12    printf("x%%=2 : %i\n",x%=2);
13    printf("x    : %i\n",x);
14    return 0;
15  }
```

Wie man der Ausgabe entnehmen kann, wird die Variable tatsächlich verändert.

```
sep@pc305-3:~/fh/c/student> bin/AssignOperation
x+=2 : 44
x    : 44
x-=2 : 42
x    : 42
x*=2 : 84
x    : 84
x/=2 : 42
x    : 42
x%=2 : 0
x    : 0
sep@pc305-3:~/fh/c/student>
```

Wenn auf der linken Seite eines Zuweisungsoperators keine Variable steht, kann auch nichts entsprechendes zugewiesen werden. Deshalb verbietet der C-Compiler Zuweisungsausdrücke, die auf der linken seite keine Variable haben:

AssignError.c

```
1 #include <stdio.h>
2 int main(){
3   printf("5+=6 : %i\n",5+=6);
4 }
```

Es kommt zu folgender Fehlermeldung:

```
sep@pc305-3:~/fh/c/student> gcc src/AssignError.c
src/AssignError.c: In function ‘main’:
src/AssignError.c:3: error: invalid lvalue in assignment
sep@pc305-3:~/fh/c/student>
```

Wir haben Seiteneffekte schon als gefährlich gegeißelt: Seiteneffekte machen Programme komplizierter zu verstehen. Daher sollen Funktionen mit Seiteneffekten nur ganz bewußt und sparsam eingesetzt werden. Das gleiche gilt im verstärkten Maße für die Zuweisungsoperatoren. Sie sollten tatsächlich nur als abkürzende Schreibweise für `x=x+5` auf einer eigenen Zeile benutzt werden. Auf komplexere Asdrücke wie `42+(x*=x+=5)` sollte verzichtet werden, es sei denn man möchte am *obfuscated C*-Wettbewerb teilnehmen.

2.5.2 Inkrement und Dekrement

Es gibt sogar noch speziellere Formen, eine Variable zu modifizieren. Sehr oft tritt die Situation ein, dass eine Variable um eins erhöht oder verringert werden soll. Hierzu kennt C eigene einstellige Operatoren, die auf Variablen angewendet werden können, die Operatoren `++` und `--`. Dabei können die Operatoren nach der Variable stehen.

Betrachten wir den Unterschied am Beispiel:

IncDec.c

```
1 #include <stdio.h>
2 int main(){
3   int x=42;
4   printf("x++ : %i\n",x++);
5   printf("x   : %i\n",x);
6   printf("x-- : %i\n",x--);
```

```
7   printf("x   : %i\n",x);
8   printf("++x : %i\n",++x);
9   printf("x   : %i\n",x);
10  printf("--x : %i\n",--x);
11  printf("x   : %i\n",x);
12  return 0;
13 }
```

An der Ausgabe kann man erkennen, dass die Operatoren eine Variable tatsächlich um eins erhöhen oder verringern.

```
sep@pc305-3:~/fh/c/student> bin/IncDec
x++ : 42
x   : 43
x-- : 43
x   : 42
++x : 43
x   : 43
--x : 42
x   : 42
sep@pc305-3:~/fh/c/student>
```

Das Ergebnis des Operatorausdruck hängt dabei davon ab, ob der Operator der Variablen vor oder nachgestellt war. Ist er der Variablen nachgestellt, so ist das Ergebnis des Ausdrucks der Wert vor Erhöhen bzw. Verringern der Variable. Ist der Operator nachgestellt, so hat der Gesamtausdruck den Wert der Variable bevor sie verändert wird. Ist der Operator vorangestellt, so hat der Gesamtausdruck den Wert der Variablen nach der Modifikation.

Auch für diese beiden Operatoren gilt: sparsam und nur in recht klaren Situationen benutzen. Die einzige Stelle, in der er von mir tatsächlich benutzt wird ist im Kopf einer `for`-Schleife:

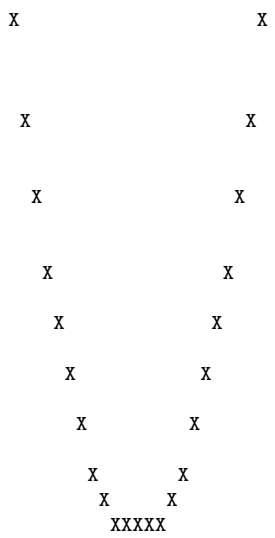
CountDownFor2.c

```
1 #include <stdio.h>
2 int main(){
3   int i;
4   for (i=10;i>0;i--){
5     printf("i = %i\n",i);
6   }
7   return 0;
8 }
```

Hier sind diese beiden Operatoren recht gut aufgehoben. Eine Variable wird in einem Bereich herauf und heruntergezählt.

Aufgabe 20 (Zusatzaufgabe)
In dieser Aufgabe machen wir einen kleinen Vorgriff auf Funktionen als Argumente.

a) Schreiben Sie ein Programm, mit dem Sie auf der Kommandozeile den Graphen einer bestimmten Funktion ausgeben können. Es soll für die Funktion:
`int f1(int x){return x*x/6-12;}`
zu folgender Ausgabe kommen:



b) Im ersten Teil haben Sie den Graphen für genau eine Funktion auf der Kommandozeile angezeigt. Überlegen Sie jetzt, wie Sie eine allgemeine Funktion zum Zeichnen von Graphen beliebiger Funktionen schreiben können. Hierzu müssen Sie eine Funktion als Parameter übergeben. Sie sollen eine Prozedur mit folgender Signatur schreiben:
`void druckeGraph(int f(int));`

Rufen Sie dann diese Prozedur mit mehreren unterschiedlichen Funktionen in einem Test auf.

Lösung

fun.c

```
1  #include <stdio.h>
2
3  int f1(int x){return x*x/6-12;}
4  int f2(int x){return x*x*x/100;}
5
6  void druckeGraph(int f(int)){
7    for (int y=0;y<30;y++){
8      for (int x=0;x<50;x++){
9        int y2=15-y;
10       printf(f(x-25)==y2?"X":" ");
11     }
12     printf("\n");
13   }
14 }
15
16 int main(){
17   printf("Funktionsgraph für f1\n");
18   druckeGraph(f1);
19   printf("Funktionsgraph für f2\n");
20   druckeGraph(f2);
```

```
21    return 0;
22  }
23
```

Kapitel 3

Softwarentwicklung

3.1 Programmieren

3.1.1 Disziplinen der Programmierung

Mit dem Begriff Programmierung wird zunächst die eigentliche Codierung eines Programms assoziiert. Eine genauerer Blick offenbart jedoch, daß dieses nur ein kleiner Teil von vielen recht unterschiedlichen Schritten ist, die zur Erstellung von Software notwendig sind:

- **Spezifikation:** Bevor eine Programmieraufgabe bewerkstelligt werden kann, muß das zu lösende Problem spezifiziert werden. Dieses kann informell durch eine natürlichsprachliche Beschreibung bis hin zu mathematisch beschriebenen Funktionen geschehen. Gegen die Spezifikation wird programmiert. Sie beschreibt das gewünschte Verhalten des zu erstellenden Programms.
- **Modellieren:** Bevor es an die eigentliche Codierung geht, wird in der Regel die Struktur des Programms modelliert. Auch dieses kann in unterschiedlichen Detaillierungsgraden geschehen. Manchmal reichen Karteikarten als hilfreiches Mittel aus, andernfalls empfiehlt sich eine umfangreiche Modellierung mit Hilfe rechnergestützter Werkzeuge. Für die objektorientierte Programmierung hat sich UML als eine geeignete Modellierungssprache durchgesetzt. In ihr lassen sich Klassendiagramme, Klassenhierarchien und Abhängigkeiten graphisch darstellen. Mit bestimmten Werkzeugen wie *Together* oder *Rational Rose* läßt sich direkt für eine UML-Modellierung Programmtext generieren.
- **Codieren:** Die eigentliche Codierung ist in der Regel der einzige Schritt, der direkt Code in der gewünschten Programmiersprache von Hand erzeugt. Alle anderen Schritte der Programmierung sind mehr oder weniger unabhängig von der zugrundeliegenden Programmiersprache.

  Für die Codierung empfiehlt es sich, Konventionen zu verabreden, wie der Code geschrieben wird, was für Bezeichner benutzt werden, in welcher Weise der Programmtext eingerückt wird. Entwicklungsabteilungen haben zumeist schriftlich verbindlich festgeschriebene Richtlinien für den Programmierstil. Dieses erleichtert, den Code der Kollegen im Projekt schnell zu verstehen.
- **Testen:** Beim Testen sind generell zu unterscheiden:

  - *Entwicklertests:* Diese werden von den Entwicklern während der Programmierung selbst geschrieben, um einzelne Programmteile (Methoden, Funktionen) separat zu testen. Es gibt eine Schule, die propagiert, Entwicklertests vor dem Code zu schreiben (*test first*). Die Tests dienen in diesem Fall als kleine Spezifikationen.
  - *Qualitätssicherung:* In der Qualitätssicherung werden die fertigen Programme gegen ihre Spezifikation getestet (*black box tests*). Hierzu werden in der Regel automatisierte Testläufe geschrieben. Die Qualitätssicherung ist personell von der Entwicklung getrennt. Es kann in der Praxis durchaus vorkommen, daß die Qualitätsabteilung mehr Mitarbeiter hat als die Entwicklungsabteilung.

- **Optimieren:** Sollten sich bei Tests oder in der Praxis Performanzprobleme zeigen, sei es durch zu hohen Speicherverbrauch als auch durch zu lange Ausführungszeiten, so wird versucht, ein Programm zu optimieren. Hierzu bedient man sich spezieller Werkzeuge (*profiler*), die für einen Programmdurchlauf ein Raum- und Zeitprofil erstellen. In diesem Profil können Programmteile, die besonders häufig durchlaufen werden, oder Objekte, die im großen Maße Speicher belegen, identifiziert werden. Mit diesen Informationen lassen sich gezielt inperformante Programmteile optimieren.

- **Verifizieren:** Eine formale Verifikation eines Programms ist ein mathematischer Beweis der Korrektheit bezüglich der Spezifikation. Das setzt natürlich voraus, daß die Spezifikation auch formal vorliegt. Man unterscheidet:

  - *partielle Korrektheit:* wenn das Programm für eine bestimte Eingabe ein Ergebnis liefert, dann ist dieses bezüglich der Spezifikation korrekt.
  - *totale Korrektheit:* Das Programm ist partiell korrekt und terminiert für jede Eingabe, d.h. liefert immer nach endlich langer Zeit ein Ergebnis.

  Eine formale Verifikation ist notorisch schwierig und allgemein nicht automatisch durchführbar. In der Praxis werden nur in ganz speziellen kritischen Anwendungen formale Verifikationen durchgeführt, z.B. bei Steuerungen gefahrenträchtiger Maschinen, so daß Menschenleben von der Korrektheit eines Programms abhängen können.

- **Wartung/Pflege:** den größten Teil seiner Zeit verbringt ein Programmierer nicht mit der Entwicklung neuer Software, sondern mit der Wartung bestehender Software. Hierzu gehören die Anpassung des Programms an neue Versionen benutzter Bibliotheken oder des Betriebssystems, auf dem das Programm läuft, sowie die Korrektur von Fehlern.

- **Debuggen:** Bei einem Programm ist immer damit zu rechnen, daß es Fehler enthält. Diese Fehler werden im besten Fall von der Qualitätssicherung entdeckt, im schlechteren Fall treten sie beim Kunden auf. Um Fehler im Programmtext zu finden, gibt es Werkzeuge, die ein schrittweises Ausführen des Programms ermöglichen *(debugger)*. Dabei lassen sich die Werte, die in bestimmten Speicherzellen stehen, auslesen und auf diese Weise der Fehler finden.

- **Internationalisieren (I18N)[1]:** Softwarefirmen wollen möglichst viel Geld mit ihrer Software verdienen und streben deshalb an, ihre Programme möglichst weltweit zu vertreiben. Hierzu muß gewährleistet sein, daß das Programm auf weltweit allen Plattformen läuft und mit verschiedenen Schriften und Textcodierungen umgehen kann. Das Programm sollte ebenso wie mit lateinischer Schrift auch mit Dokumenten in anderen Schriften umgehen

[1] I18N ist eine Abkürzung für das Wort *internationalization*, das mit einem i beginnt, mit einem n endet und dazwischen 18 Buchstaben hat.

können. Fremdländische Akzente und deutsche Umlaute sollten bearbeitbar sein. Aber auch unterschiedliche Tastaturbelegungen bis hin zu unterschiedlichen Schreibrichtungen sollten unterstützt werden.

Die Internationalisierung ist ein weites Feld, und wenn nicht am Anfang der Programmerstellung hierauf Rücksicht genommen wird, so ist es schwer, nachträglich das Programm zu internationalisieren.

- **Lokalisieren (L12N):** Ebenso wie die Internationalisierung beschäftigt sich die Lokalisierung damit, daß ein Programm in anderen Ländern eingesetzt werden kann. Beschäftigt sich die Internationalisierung damit, daß fremde Dokumente bearbeitet werden können, versucht die Lokalisierung, das Programm komplett für die fremde Sprache zu übersetzen. Hierzu gehören Menueinträge in der fremden Sprache, Beschriftungen der Schaltflächen oder auch Fehlermeldungen in fremder Sprache und Schrift. Insbesondere haben verschiedene Schriften unterschiedlichen Platzbedarf; auch das ist beim Erstellen der Programmoberfläche zu berücksichtigen.

- **Portieren:** Oft wird es nötig, ein Programm auf eine andere Plattform zu portieren. Ein unter Windows erstelltes Programm soll z.B. auch auf Unix-Systemen zur Verfügung stehen.

- **Dokumentieren:** Der Programmtext allein reicht in der Regel nicht aus, damit das Programm von Fremden oder dem Programmierer selbst nach geraumer Zeit gut verstanden werden kann. Um ein Programm näher zu erklären, wird im Programmtext Kommentar eingefügt. Kommentare erklären die benutzten Algorithmen, die Bedeutung bestimmter Datenfelder oder die Schnittstellen und Benutzung bestimmter Methoden.

  Es ist zu empfehlen, sich anzugewöhnen, Quelltextdokumentation immer auf Englisch zu schreiben. Es ist oft nicht abzusehen, wer einmal einen Programmtext zu sehen bekommt. Vielleicht ein japanischer Kollege, der das Programm für Japan lokalisiert, oder der irische Kollege, der, nachdem die Firma mit einer anderen Firma fusionierte, das Programm auf ein anderes Betriebssystem portiert, oder vielleicht die englische Werksstudentin, die für ein Jahr in der Firma arbeitet.

3.1.2 Was ist ein Programm

Die Frage danach, was ein Programm eigentlich ist, läßt sich aus verschiedenen Perspektiven recht unterschiedlich beantworten. Wir haben schon eine ganze Reihe von Programmen geschrieben, so dass zumindest schon ein erstes intuitives Gefühl dafür, was ein Programm eigentlich ist, existieren sollte. Tatsächlich lassen sich Computerprogramme unter unterschiedlichsten Aspekten betrachten. Was ist also eigenbtlich ein Programm?

pragmatische Antwort

Eine Textdatei, die durch ein anderes Programm in einen ausführbaren Maschinencode übersetzt wird. Dieses andere Programm ist ein Übersetzer, engl. *compiler*.

mathematische Antwort

Eine Funktion, die deterministisch für Eingabewerte einen Ausgabewert berechnet.

sprachwissenschaftliche Antwort

Ein Satz einer durch eine Grammatik beschriebenen Sprache. Für dieses syntaktische Konstrukt existiert eine Bedeutung in Form einer operationale Semantik.

operationale Antwort

Eine Folge von durch den Computer ausführbaren Befehlen, die den Speicher des Computers manipulieren.

3.1.3 Klassifizierung von Programmiersprachen

Es gibt mittlerweile mehr Programmiersprachen als natürliche Sprachen. In Wikipedia (`http://de.wikipedia.org/wiki/Liste_der_Programmiersprachen`) existiert eine recht umfangreiche Liste von Programmiersprachen. Die meisten Sprachen führen entsprechend nur ein Schattendasein und die Mehrzahl der Programme konzentriert sich auf einige wenige Sprachen. Programmiersprachen lassen sich nach den unterschiedlichsten Kriterien klassifizieren.

Hauptklassen

Im folgenden eine hilfreiche Klassifizierung in fünf verschiedene Hauptklassen.

- **imperativ** (C[KR78], Pascal[Wir], Fortran, Cobol): das Hauptkonstrukt dieser Sprachen sind Befehle, die den Speicher manipulieren. Dieses äußert sich primär durch einen Zuweisungsbefehl, der es erlaubt, Variablen neue Werte zuzuweisen.
- **objektorientiert** (Java, C++, C#, Eiffel, Smalltalk): Daten werden in Form von Objekten organisiert. Diese Objekte bündeln mit den Daten auch die auf diesen Daten anwendbaren Methoden.
- **funktional** (Lisp, ML[MTH90], Haskell[PJ03], Scheme, Erlang, Clean, F#): Programme werden als mathematische Funktionen verstanden und auch Funktionen können Daten sein. Dieses Programmierparadigma versucht, sich möglichst weit von der Architektur des Computers zu lösen. Veränderbare Speicherzellen gibt es in rein funktionalen Sprachen nicht und erst recht keine Zuweisungsbefehle.
- **Skriptsprachen** (Perl, AWK): solche Sprachen sind dazu entworfen, einfache kleine Programme schnell zu erzeugen. Sie haben meist kein Typsystem und nur eine begrenzte Zahl an Strukturierungsmöglichkeiten, oft aber eine mächtige Bibliothek, um Zeichenketten zu manipulieren.
- **logisch** (Prolog): aus der KI (künstlichen Intelligenz) stammen logische Programmiersprachen. Hier wird ein Programm als logische Formel, für die ein Beweis gesucht wird, verstanden.

Ausführungsmodelle

Der Programmierer schreibt den lesbaren Quelltext seines Programmes. Um ein Programm auf einem Computer laufen zu lassen, muß es erst in einen Programmcode übersetzt werden, den der Computer versteht. Für diesen Schritt gibt es auch unterschiedliche Modelle:

- **kompiliert** (C, Cobol, Fortran): in einem Übersetzungsschritt wird aus dem Quelltext direkt das ausführbare Programm erzeugt, das dann unabhängig von irgendwelchen Hilfen der Programmiersprache ausgeführt werden kann.
- **interpretiert** (Lisp, Scheme): der Programmtext wird nicht in eine ausführbare Datei übersetzt, sondern durch einen Interpreter Stück für Stück anhand des Quelltextes ausgeführt. Hierzu muß stets der Interpreter zur Verfügung stehen, um das Programmm auszuführen. Interpretierte Programme sind langsamer in der Ausführung als übersetzte Programme.
- **abstrakte Maschine über *byte code*** (Java, ML): dieses ist quasi eine Mischform aus den obigen zwei Ausführungsmodellen. Der Quelltext wird übersetzt in Befehle nicht für einen konkreten Computer, sondern für eine abstrakte Maschine. Für diese abstrakte Maschine steht dann ein Interpreter zur Verfügung. Der Vorteil ist, daß durch die zusätzliche Abstraktionsebene der Übersetzer unabhängig von einer konkreten Maschine Code erzeugen kann und das Programm auf auf allen Systemen laufen kann, für die es einen Interpreter der abstrakten Maschine gibt.

Es gibt Programmiersprachen, für die sowohl Interpreter als auch Übersetzer zur Verfügung stehen. In diesem Fall wird der Interpreter gerne zur Programmentwicklung benutzt und der Übersetzer erst, wenn das Programm fertig entwickelt ist.

3.2 Präprozessor und Linker: hilfreiche Freunde des Compilers

Wenn wir bisher tatsächlich Programme übersetzt haben, um eine ausführbare Datei zu erhalten, haben wir genaugenommen nicht nur den Compiler aktiviert, sondern auch zwei gute kleine Helfer des Compilers: den Präprozessor und den Linker. Der Präprozessor bereitet den Quelltext noch ein wenig auf, bevor er ihn den Compiler gibt, der Linker fügt erst die einzelnen verschiedenen übersetzten Programmteile, die der Compiler erzeugt hat, zu einer ausführbaren Datei zusammen. In diesem Kapitel wollen wir ein wenig genaueres Augenmerk auf diese beiden Freunden nehmen.

3.2.1 Funktionsprototypen

Betrachtet wir hierzu zunächst ein ganz simples C Programm, in dem gerade mal eine einfache Funktion geschrieben und anschließend aufgerufen wird.

Answer1.c

```
1 #include <stdio.h>
2
3 int getAnswer(){
```

```
4   return 42;
5 };
6
7 int main(){
8   printf("Die Antwort lautet: %i\n", getAnswer());
9   return 0;
10 }
```

Wir definieren, die parameterlose Funktion `getAnswer` und rufen sie in der Funktion `main` auf.

In höheren Programmiersprachen spielt die Reihenfolge der Funktionen in einem Programm keine Rolle. Der Kompilierer hat jeweils bereits die ganze Quelldatei studiert, bevor er die Rümpfe der Methoden übersetzt. Wir können versuchen, ob dieses für C auch gilt. Hierzu vertauschen wir die Reihenfolge der beiden Funktionen im letzten Programm.

Answer2.c

```
1  #include <stdio.h>
2
3  int main(){
4    printf("Die Antwort lautet: %i\n",getAnswer());
5    return 0;
6  }
7
8  int getAnswer(){
9    return 42;
10 };
```

Der Versuch dieses Programm zu übersetzen führt zumindest zu einer Warnung:

```
sep@swe10:~/fh/c> gcc -Wall -o ../obj/Answer2.o -c ./Answer2.c
Answer2.c: In function ‘main’:
Answer2.c:4: warning: implicit declaration of function ‘getAnswer’

sep@swe10:~/fh/c>
```

Offensichtlich können in C Funktionen nur benutzt werden, nachdem sie vorher im Programm definiert wurden. Diese Einschränkung hat sicherlich mit den Stand der Compilerbautechnik am Anfang der 70er Jahre zu tun. Ein Compiler, der nur einmal sequentiell das Programm von vorne nach hinten durchgeht, läßt sich leichter bauen, als ein Compiler, der mehrmals über eine Quelldatei gehen muß, um nacheinander verschiedene Informationen zu sammeln.

Wollen wir die Reihenfolge der beiden Funktionen, wie wir sie in der Datei `Answer2.c` geschrieben haben, beibehalten, so können wir das in C, indem wir die Signatur der Funktion `getAnswer` vorwegnehmen. Man kann in C in ein und derselben Datei die Signatur einer Funktion deklarieren und später in der Datei diese Funktion erst implementieren.

Answer3.c

```
1  #include <stdio.h>int getAnswer();
2
3  int main(){
4    printf("Die Antwort lautet: %i\n", getAnswer());
```

```
5   return 0;
6 }
7
8 int getAnswer(){
9   return 42;
10 }
```

Jetzt übersetzt das Programm wieder fehlerfrei. Mit dieser Technik simuliert man gewissermaßen, was ein moderner Compiler sieht, wenn er mehrfach über einen Quelltext geht. Zunächst sieht er die Signaturen aller Funktionen und erst dann betrachtet er ihre Rümpfe. Sobald die Signatur einer Funktion bekannt ist, kann die Funktion benutzt werden.

3.2.2 Header-Dateien

Nach dem geraden gelernten, scheint es sinnvoll in einem C Programm zunächst die Signaturen aller Funktionen zu sammeln und dann die einzelnen Funktionen zu implementieren. Die Sammlung der Signaturen ergibt eine schöne Zusammenfassung aller von einem Programm angebotener Funktionen. Es liegt nahe, diese in eine getrennte Datei auszulagern. Hierzu bedient man sich in C der sogenannten *header*-Dateien, die mit der Dateiendung `.h` abgespeichert werden.

Für unser letztes Beispiel können wir eine *header*-Datei schreiben, in der die Signatur der Funktion `getAnswer` steht:

Answer4.h

```
1 #include <stdio.h>
2
3 int getAnswer();
```

Statt jetzt in einem Programm diese Signatur am Anfang des Programms zu schreiben, sagen wir dem Compiler, er soll die *header*-Datei mit in das Programm einfügen. Hierzu benutzen wir das `#include` Konstrukt. Unser Programm sieht dann wie folgt aus:

Answer4.c

```
1  #include "Answer4.h"
2
3  int main(){
4    printf("Die Antwort lautet: %i\n",getAnswer());
5    return 0;
6  }
7
8  int getAnswer(){
9    return 42;
10 }
```

Wichtig ist hier zu verstehen, daß mit dem `include`, bevor der Compiler anfängt das Programm zu übersetzen, die Datei `Answer4.h` komplett wörtlich an der entsprechenden Stelle eingefügt wird. Dieses Einfügen wird nicht vom eigentlichen Compiler gemacht, sondern vom einem Programm, das vor dem Compiler den Quelltext manipuliert, den sogenannten Präprozessor. Der

Präprozessor liest die Zeilen des Quelltext, die mit dem Zeichen `#` beginnen. Diese nennt man Direktiven. Sie sind nicht Teil des eigentlichen Programms, sondern erklären dem Präprozessor, wie das eigentliche Programm zusammenzusetzen ist.

3.2.3 Objektdateien und Linker

Wer unsere bisherigen Programme übersetzt hat, wird festgestellt haben, daß Dateien mit der Endung `.o` ins Dateisystem geschrieben wurden. Diese Dateien heißen Objektdateien. Der eigentliche Compiler erzeugt nicht das ausführbare Programm, sondern Objektdateien. Die Objektdateien werden dann erst durch den sogenannten Linker zu ausführbaren Dateien zusammengefasst.

Mit der Option `-c` kann man den Compiler dazu bringen, nur die Objektdatei zu erzeugen und nicht anschließend noch das ausführbare Programm zu erzeugen.

Der Compiler kann Objektdateien erzeugen, in denen noch nicht jede Funktion implementiert ist. Wir können das letzte Programm schreiben, ohne die Funktion `getAnswer` implementiert zu haben:

Answer5.c

```
1 #include "Answer4.h"
2
3 int main(){
4   printf("Die Antwort lautet: %i\n", getAnswer());
5   return 0;
6 }
```

Mit der Option `-c` kann der Compiler fehlerfrei die Objektdatei erzeugen.

```
sep@swe10:~/fh/prog3/examples> gcc -c Hello6.c
sep@swe10:~/fh/prog3/examples> ls -l Hello6.*
-rw-r--r--    1 sep      users          84 Sep  3 17:36 Hello6.c
-rw-r--r--    1 sep      users        8452 Sep  3 17:40 Hello6.o
sep@swe10:~/fh/prog3/examples>
```

Wollen wir hingegen ein ausführbares Programm erzeugen, so bekommen wir einen Fehler, und zwar nicht vom Compiler sondern vom Linker. Der Linker versucht jede deklarierte Funktion mit konkreten Programmcode aufzulösen. Wenn er einen solchen Code nicht findet, so meldet er einen Fehler:

```
sep@swe10:~/fh/prog3/examples> gcc Hello6.c
/tmp/ccRKWUqK.o: In function ‘main’:
/tmp/ccRKWUqK.o(.text+0xe): undefined reference to ‘getHallo(void)’
collect2: ld returned 1 exit status
sep@swe10:~/fh/prog3/examples>
```

Um einen solchen Fehler zu produzieren, bedarf es keiner *header*-Datei. Wir können in einem Programm eine Funktion deklarieren ohne sie zu implementieren.

Answer6.c

```
1 #include <stdio.h>
2
```

```
3  int getAnswer();
4
5  int main(){
6    printf("Die Antwort lautet: %i\n", getAnswer());
7    return 0;
8  }
9
```

Wir bekommen wieder einen Fehler des Linkers:

```
sep@swe10:~/fh/c> gcc Hello7.c
/tmp/ccdm3ng0.o: In function ‘main’:
/tmp/ccdm3ng0.o(.text+0xe): undefined reference to ‘getHallo(void)’
collect2: ld returned 1 exit status
sep@swe10:~/fh/prog3/examples>
```

Der Linker hat im Gegensatz zum Compiler meherere Dateien als Eingabe, die er gemeinsam verarbeitet.

3.2.4 Mehrfach inkludieren

Genauso wie Funktionen können auch Variablen in Headerdateien deklariert sein.

C1.h

```
1  int x = 42;
```

Ebenso können Headerdateien weitere Headerdateien inkludieren:

C2.h

```
1  #include "C1.h"
```

Hier kann es zu einem Fehler kommen, wenn beide diese Headerdateien auf die eine oder andere Art in eine Datei inkludiert wird.

C3.c

```
1  #include "C1.h"
2  #include "C2.h"
3
4  int main(){};
```

Der Versuch dieses Programm zu übersetzen, führt zu einem Fehler.

```
sep@linux:~/fh/c> gcc -Wall -LDIR../obj -o ../bin/C3 ./C3.c
In file included from C2.h:1,
                 from C3.c:2:
C1.h:1: error: redefinition of ‘x’
C1.h:1: error: ‘x’ previously defined here
strip: ’../bin/C3’: No such file
sep@linux:~/fh/c>
```

Um solche Fehler, die entstehen, wenn ein und dieselbe Datei mehrfach inkludiert wird, von vorneherein auszuschließen, bedient man sich standardmäßig einer weiteren Direktive. Mit der Direktive `#define` können Werte definiert werden. Mit der Direktive `#ifndef` kann abgefragt werden, ob ein Wert bereits definiert wurde. Nun kann man in einer Headerdatei eine für diese Datei spezifische Variable definieren. Üblicher Weise bildet man diese mit zwei Unterstrichen, gefolgt vom komplett in Großbuchstaben geschriebenen Dateinamen und der Endung `_H`. Den kompletten Inhalt der Headerdatei klammert man nun in einer `#ifndef`-Direktive. So wird der Inhalt einer Headerdatei nur dann eingefügt, wenn die spezifische Variable noch nicht definiert wurde. Also höchstens einmal.

Beherzigen wir diese Vorgehensweise, so erhalten wir statt der Datei `C1.h` von oben, eine entsprechende Datei, die wir zur Unterscheidung als `D.h` speichern.

D1.h

```
1 #ifndef __D1_H
2
3 #define __D1_H ;
4
5 int x = 42;
6
7 #endif /* __D1_H */
```

Die Variable, die markiert, ob die Datei schon inkludiert wurde, heißt nach unserer entsprechenden Konvention `__D1_H`.

Diese Datei läßt sich jetzt woanders inkludieren.

D2.h

```
1 #include "D1.h"
```

Eine mehrfache Inklusion führt jetzt nicht mehr zu einen Fehler.

D3.c

```
1 #include "D1.h"
2 #include "D2.h"
3
4 int main(){
5   return 0;
6 };
```

Das Programm läßt sich jetzt fehlerfrei übersetzen:

```
sep@linux:~/fh/prog3/examples/src> gcc -o D3 D3.c
sep@linux:~/fh/prog3/examples/src> ./D3
```

In größeren C-Projekten kann man diese Vorgehensweise standardmäßig angewendet sehen.

3.2.5 Konstantendeklaration

Bisher haben wir die Direktiven des benutzt, um andere Dateien zu inkludieren und um über die Bedingung `ifndef` zu verhindern, dass eine header-datei mehrfach inkludiert wird. Eine sehr häufig verwendete Direktive, erlaubt es konstante Werte zu definieren. Hierzu dient das `define`, das bereits oben zu wehen war. Der Präprzessor ersetzt jedesmal wenn ein Wort, das durch ein `define` einen Wrt erhalten hat, dieses Wort durch den Wert. Der Compiler bekommt dieses Wort also gar nie zu sehen, sondern nur nuch den Wert. Damit lassen sich nun die Wörter `true` und `false` als Wahrheitswerte definieren.

Boolean.h

```
1 #ifndef __BOOLEAN_H
2 #define __BOOLEAN_H
3 #define true 1
4 #define false 0
5 #endif
```

Inkludieren wir diese header-Datei, so können wir die bool'schen Werte benutzen.

TestBoolean.c

```
1  #include <stdio.h>
2  #include "Boolean.h"
3
4  int odd(int i){
5    if (i%2==1)return true;
6    return false;
7  }
8
9  int main(){
10   if (odd(42)) printf("42 ist ungerade\n");
11   else  printf("42 ist gerade\n");
12   return 0;
13 }
14
```

Tatsächlich in großen Projekten werden die Direktiven des Präprozessors exzessiv genutzt. Das Programm sieht damit manchmal etwas seltsam aus und erinnert wenig an C. Eigentlich ist der Präprozessor nur nötig, weil der C Compiler viele Funktionalität nicht selbst anbietet.

3.3 Kompilierung: Mach mal!

Es gibt ein wunderbares kleines Programm mit Namen `make`. Starten wir dieses Programm einmal mit einem Argument:

```
panitz@fozzie(~)$ make love
make: *** No rule to make target ‘love’.  Stop.
panitz@fozzie(~)$
```

Nun, etwas anderes hätten wir von einem Computer auch kaum erwartet. Wenn er etwas machen soll, so müssen wir ihm in der Regel sagen, wie er das machen kann. Hierzu liest das Programm `make` standardmäßig eine Datei mit Namen `Makefile`. In dieser sind Ziele beschrieben und die Operationen, die zum Erzeugen dieser Ziele notwendig sind.

Ein Ziel ist in der Regel eine Datei, die zu erzeugen ist. Syntaktisch wird ein Ziel so definiert, daß der Zielname in der ersten Spalte einer Zeile steht. Ihm folgt ein Doppelpunkt, dem auf derselben Zeile Ziele aufgelistet folgen, die zuerst erzeugt werden müssen, bevor dieses Ziel gebaut werden kann. Dann folgt in den folgenden Zeilen durch einen Tabulatorschritt eingerückt, welche Befehle zum Erreichen des Ziels ausgeführt werden müssen. Eine einfache Makedatei zum Bauen des Programms `HalloFreunde` sieht entsprechend wie folgt aus.

makeHalloFreunde.

```
1 HalloFreunde: HalloFreunde.o
2         gcc -o HalloFreunde HalloFreunde.o
3
4 HalloFreunde.o: HalloFreunde.c
5         gcc -c HalloFreunde.c
```

Es gibt in dieser Makedatei zwei Ziele: die ausführbare Datei `HalloFreunde` und die Objektdatei `HalloFreunde.o`. Zum Erzeugen der Objektdatei, muß die Datei `HalloFreunde.c` vorhanden sein, zum Erzeugen der ausführbaren Datei, muß erst die Objektdatei generiert worden sein. Wir können jetzt `make` bitten, unsere `HalloFreunde`-Applikation zu bauen. Die Makedatei nicht `Makefile` heißt, welches die standardmäßig von `make` als Steuerdatei gelesene Datei ist, müssen wir `make` mit der Option `-f` als Argument mitgeben, in welcher Datei die Ziele zum Machen definiert sind.

Für die Erzeugung des Programms `HalloFreunde` reicht also folgender Aufruf von `make`:

```
sep@pc216-5:~/fh/cpp/student/src> make -f makeHalloFreunde
gcc    -c -o HalloFreunde.o HalloFreunde.c
gcc -o HalloFreunde HalloFreunde.o
sep@pc216-5:~/fh/cpp/student/src> ./HalloFreunde
hello world
sep@pc216-5:~/fh/cpp/student/src> make -f makeHalloFreunde
make: ■HalloFreunde■ ist bereits aktualisiert.
sep@pc216-5:~/fh/cpp/student/src>
```

Gibt man `make` nicht als zusätzliches Argument an, welches Ziel zu bauen ist, so baut `make` das erste in der Makedatei definierte Ziel.

Die wahre Stärke von `make` ist aber, daß nicht stur alle Schritte zum Erzeugen eines Ziels durchgeführt werden, sondern anhand der Zeitstempel der Dateien geschaut wird, ob es denn überhaupt notwendig ist, bestimmte Ziele neu zu bauen, oder ob sie schon aktuell im Dateisystem vorliegen. So kommt es, daß beim zweiten Aufruf von `make` oben, nicht erneut kompiliert wurde.

Um gute und strukturierte Makedateien zu schreiben, kennt `make` eine ganze Anzahl weiterer Konstrukte. Zuvorderst Kommentare. Kommantarzeilen beginnen mit einem Gattersymbol `#`. Dann lassen sich Konstanten definieren. Es empfiehlt sich in einer Makedatei alle Pfade und Werkzeuge, wie z.B. der benutzte Compiler als Konstante zu definieren, damit diese leicht global geändert werden können. Schließlich lassen sich noch allgemeine Regeln, wie man von einem Dateitypen zu einem anderen Dateitypen gelangt, definieren; z.B. das der Aufruf `gcc -c` zu benutzen ist, für eine C-Datei eine Objektdatei zu generieren.

Eine typische Makedatei, die all diese Eigenschaften ausnutzt, sieht für ein C-Projekt dann wie folgt aus.

Makefile.

```
1  #der zu benutzende C++ Compiler
2  CC = gcc
3  EXE = TestReadRoman
4
5  #das Hauptprogramm
6  TestReadRoman: TestReadRoman.o ReadRoman.o
7          $(CC) -o TestReadRoman TestReadRoman.o ReadRoman.o
8
9  #allgemeine Regel zum Aufruf des Compilers zum generieren
10 #der Objektdatei aus der Quelltextdatei
11 .c.o:
12         $(CC) -c $<
13
14 #alles wieder löschen
15 clean:
16         rm $(EXE)
17         rm *.o
18
19 #just for fun
20 love:
21         @echo "I do not know, how to make love.";
22
```

Für eine genaue Beschreibung der Möglichkeiten einer Makedatei konsultiere man die Dokumentation von `make`. Nahezu alle C++-Projekte benutzen `make` zum Bauen des Projektes. Für Javaprojekte hat sich hingegen das Programm `ant` (`http://ant.apache.org/`) durchgesetzt.

Auch die verschiedenen Versionen dieses Skripts werden mit Hilfe von `make` erzeugt.

Aufgabe 21 Kopieren Sie sich den Quelltextordner dieses Skripts und rufen Sie die darin enthaltene obige Makefile-Datei auf. Rufen Sie dann `make` direkt nocheinmal auf. Ändern Sie anschließend die Datei `ReadRoman.c` und starten Sie erneut `make`. Was beobachten Sie in Hinblick auf `TestReadRoman.o`. Starten Sie schließlich `make` mit dem Ziel `love`.

3.4 Dokumentation

3.4.1 Kommentare

Schön wäre es sicher, wenn ein Programmtext immer so klar und eindeutig wäre, dass man sofort ohne weitere Hilfe erkennen könnte, was die einzelnen Programmzeilen ausdrücken und bewirken. Davon ist in der Regel nicht auszugehen. Daher ist es unerläßlich, dass man im Programmtext Kommentarzeilen schreibt. Diese helfer Kollegen und einem selbst mit etwas zeitlichen Abstand das Programm weiterhin zu verstehen.

In C gibt es mitlerweile zwei Arten, Kommentare im Programmtext zu schreiben. Für einzeilige Kommentare gibt es den doppelten Schrägstrich. Von diesem an wird der Rest der Zeile als Kommentar betrachtet.

Längere Kommentarblöcke werden mit den Zeichen `/*` begonnen und enden mit den Zeichen `*/`. Hier zwischen können beliebig viele Zeichen in mehreren Zeilen als Kommentar stehen.

Ein kleines Beispiel eines kommentierten Programms.

Comments.c

```
1  #include <stdio.h>
2
3   /*
4    Berechnet die Fakultät vom Eingabeparameter.
5    Der Eingabeparameter wird als nicht-negative Zahl erwartet.
6    Ansonsten kann es zur Nichtterminierung der Funktion kommen.
7    */
8    int f(int n){
9    // Lösung iterativ mit einer for-Schleife
10
11   int result=1; //akkumuliert das Ergebnis
12   int i=n;      //Laufvariable für die Schleife
13   for (;i>0;i++){
14     result = result*i;
15   }
16   return result;
17 }
```

3.4.2 Benutzung von Doxygen

Heutzutage schreibt niemand mehr ein einzelnen Programm ganz allein. In der Regel wird komplexere Software entwickelt. Die meiste Software wird damit nicht einmal ein Programm sein, sondern eine Bibliothek, die in anderen Softwareprodukten verwendet wird.

Um die von einer Bibliothek bereitgestellte Funktionalität sinnvoll verwenden zu können, ist diese auf einheitliche Weise zu dokumentieren. Zur Dokumentation von C-Bibliotheken kann man das Werkzeug Doxygen (`http://www.stack.nl/~dimitri/doxygen/index.html`) einsetzen. Dieses Programm liest die C Kopfdateien und generiert eine umfassende Dokumentation in verschiedenen Formaten. Am häufigsten wird dabei wahrscheinlich die HTML-Dokumentation verwendet.

Doxygen kommt mit einem Kommandozeilenprogramm `doxygen`. Dieses erwartet als Eingabe eine Steuerdatei zur Konfiguration. In dieser Konfigurationsdatei können eine große Menge von Parametern eingestellt werden, wie die Sprache, in der die Dokumentation erstellt werden soll oder auch den Projektnamen, des zu dokumentierenden Projekts. Zum Glück ist `doxygen` in der Lage, sich selbst eine solche Konfigurationsdatei zu generieren, die man dann manuell nacheditieren kann. Hierzu benutzt man das Kommandozeilenargument `-g` von `doxygen`.

Ein kurzer Start mit `doxygen` sieht also wie folgt aus:

- Generierung einer Konfigurationsdatei mit:

```
sep@pc216-5:~/fh/c> doxygen -g MeinProjekt
```

```
Configuration file 'MeinProjekt' created.

Now edit the configuration file and enter

  doxygen MeinProjekt

to generate the documentation for your project

sep@pc216-5:~/fh/c>
```

- Anschließend editiert man die erzeugte Datei `MeinProjekt`. Für den Anfang sei zu empfehlen `EXTRACT_ALL = YES` zu setzen. Selbst wenn keine eigentlich für Doxygen gedachte Dokumentation im Quelltext steht, werden alle Programmteile in der Dokumentation aufgelistet.
- Die Eigentliche Generierung der Dokumentation.

```
sep@pc216-5:~/fh/c> doxygen  MeinProjekt
sep@pc216-5:~/fh/c>
```

Jetzt befindet sich die generierte Dokumentation im in der Konfigurationsdatei angegebenen Ausgabeordner.

Aufgabe 22 Laden Sie sich die kompletten Quelltextdateien dieses Skriptes von der Webseite und generieren Sie mit `Doxygen` eine Dokumentation für diese.

3.4.3 Dokumentation des Quelltextes

Bisher konnte sich *Doxygen* lediglich auf den Quelltext beziehen, um eine Dokumentation zu erzeugen. Natürlich reicht das nicht aus. Daher kann man im Quelltext Kommentare anbringen, die *Doxygen* auswertet. Es gibt verschiedene Syntax hierzu, z.B. die klassische aus C bekannte Art zu kommentieren:

DoxygenMe.h

```
1  /**
2    Dies ist eine einfache Funktion.
3  Sie ist in der Lage ganz tolle Dinge zu tun.
4   */
5  void f();
```

In diesen Kommentarblöcken, die über dem zu dokumentierenden Code stehen, können mit dem Backslashsymbol bestimmte Attribute gesetzt werden. So gibt es z.B. das Attribut `\param` für einen Funktionsparameter oder das Attribut `\return` für das Funktionsergebnis.

DoxygenMe.h

```
6  /**
7    Die Additionsfunktion.
8    Sie kann dazu benutzt werden, um einen Funktionszeiger
9    auf die Addition
10   zur Verfügung zu haben. Sie ruft in ihrem Rumpf lediglich den
```

```
11    Additionsoperator auf.
12
13    \param x der erste Operand.
14    \param y der zweite (rechte) Operand.
15    \return die Summe der beiden Operanden.
16  */
17  int add(int x, int y);
```

Aufgabe 23 Betrachten Sie, die für die Funktionen `f` und `add` in der letzten Aufgabe generierte Dokumentation. *Doxygen* kennt viele solcher Attribute und viele verschiedene mächtige unterschiedliche Weisen der Dokumentation. Zum Glück ist *Doxygen* selbst recht gut erklärt, so daß mit dem hier gegebenen Einstieg es hoffentlich leicht möglich ist, sich mit dem Programm auseinanderzusetzen.

Aufgabe 24 Beschäftigen Sie sich mit der Dokumentation von *Doxygen* und testen einige der darin beschriebenen Möglichkeiten der Dokumentation.

3.5 Debuggen

Wie Ernie in der Sesamstraße bereits gesungen hat: *Jeder macht mal Fehler ich und du*; so müssen wir davon ausgehen, daß unsere Programme nicht auf Anhieb die gewünschte Funktionalität bereitstellen. Insbesondere da C uns relativ viel Möglichkeiten und Freiheiten in der Art der Programmierung ermöglicht, läßt es uns auch die Freiheiten, viele Fehler zu machen.

Um einen Fehler aufzuspüren können wir auf zwei Techniken zurückgreifen:

- Logging: Während der Ausführung wird in bestimmten Programmzeilen eine Statusmeldung in eine Log-Datei geschrieben. Diese läßt sich hinterher analysieren. Die einfachste Form des Loggings ist, ab und zu eine Meldung auf die Konsole auszugeben.
- Debuggen: mit Hilfe eines Debuggers läßt sich schrittweise das Programm ausführen und dabei in den Speichern schauen.

Der Begriff *debuggen*, den man wörtlich mit *Entwanzen* übersetzen könnte, bezieht sich auf die Benutzung des Wortes *bug* für Macken in einem technischen System. Der Legende nach stammt diese Benutzung des Wortes *bug* aus der Zeit der ersten Rechner, die noch nicht mit Halbleitern geschaltet haben, sondern mit Relais, ähnlich, wie man sie von Flipperautomaten kennt. Es soll einmal zu einem Fehler im Programmablauf gekommen sein, der sich schließlich darauf zurückführen ließ, daß ein Käfer zwischen ein Relais geklettert war. Eine schöne Ankdote, die leider falsch ist, denn aus schon früheren Dokumenten geht hervor, daß der Begriff *bug* schon im 19. Jahrhundert für technische Fehler gebräuchlich war.

Um ein Programm mit einem Debugger zu durchlaufen, ist es notwenig, daß der Debugger Informationen im ausführbaren Code findet, die sich auf den Quelltext beziehen. Dieses sind umfangreiche Informationen. Der `gcc` generiert die notwendige Information in die Objektdateien, wenn man ihn mit der Option `-g` startet.

3.5.1 Debuggen auf der Kommandozeile

Die rudimentärste Art des Debuggens ist mit Hilfe des Kommandozeilenprogramms `gdb`. Wir wollen das einmal ausprobieren. Hierzu brauchen wir natürlich ein entsprechendes Programm, welches hier zunächst definiert sei. Es handelt sich dabei um ein Programm zur Konvertierung von als Strings vorliegenden römischen Zahlen in die entsprechende arabische Zahl als `int`.

ReadRoman.h

```
1  #ifndef READ_ROMAN__H
2  #define  READ_ROMAN__H ;
3
4  int romanToInt(char* roman);
5  #endif
```

Eine rudimentäre Implementierung dieser Kopfdatei:

ReadRoman.c

```
1  #include "ReadRoman.h"
2
3  int romanCharToInt(char c){
4    switch (c){
5      case 'I': return 1;
6      case 'V': return 5;
7      case 'X': return 10;
8      case 'L': return 50;
9      case 'C': return 100;
10     case 'D': return 500;
11     case 'M': return 1000;
12   }
13   return -1;
14 }
15
16 int romanToInt(char* roman){
17   int result=0;
18   int last=0;
19   char* i;
20   for (i=roman;*i != '\0';i++){
21     int current=romanCharToInt(*i);
22     if (current>last) result = result-2*last;
23     result=result+current;
24     last=current;
25   }
26   return result;
27 }
```

Wir benutzen hier schon ein paar wenige C Eigenschaften, die uns aber vorerst nicht stören sollen und nicht weiter hinterfragt werden sollen. Es folgt ein minimaler Test dieses Programms:

TestReadRoman.c

```
1  #include "ReadRoman.h"
2  #include <stdio.h>
```

```
3  int main(){
4    printf("romanToInt(\"MMMCDXLIX\") = %i\n",romanToInt("MMMCDXLIX"));
5    return 0;
6  }
```

Aufgabe 25 Übersetzen Sie das Programm mit der `-g`-Option des `gcc` und lassen Sie das Programm ausführen.

Jetzt können sie den `gnu debugger` mit dem Programm `TestReadRoman` aufrufen. Sie geraten in einen Kommandozeileninterpreter des Debuggers. Der Debugger kennt eine Reihe Kommandos, über die er mit dem Kommando `help` gerne informiert:

```
sep@pc216-5:~/fh/cpp/student/bin> gdb TestReadRoman
GNU gdb 6.1
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i586-suse-linux"...
  Using host libthread_db library "/lib/tls/libthread_db.so.1".

(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands

Type "help" followed by a class name for a list of commands in that class.
Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
(gdb)
```

Das naheliegendste Kommando ist das Kommando `run`. Es führt dazu, daß das Programm ausgeführt wird.

```
(gdb) run
Starting program: /home/sep/fh/cpp/student/bin/TestReadRoman
3449

Program exited normally.
(gdb)
```

Wir wollen aber nicht, daß das Programm einmal komplett ausgeführt wird, sondern schrittweise die Ausführung nachverfolgen. Hierzu müssen wir einen sogenannten *breakpoint* setzen. Wir können diesen auf die erste Zeile des Programms setzen:

```
(gdb) break 4
Breakpoint 1 at 0x804839c: file TestReadRoman.c, line 4.
(gdb)
```

Wenn wir jetzt `run` sagen, stoppet der Debugger in der Zeile vier:

```
(gdb) run
Starting program: /home/sep/fh/c/tutor/src/TestReadRoman

Breakpoint 1, main () at TestReadRoman.c:4
4         printf("romanToInt(\"MMMCDXLIX\") = %i\n",romanToInt("MMMCDXLIX"));
(gdb)
```

Jetzt können wir z.B. Zeile für Zeile das Programm vom Debugger ausführen lassen. Hierzu gibt es den Befehl `step`:

```
(gdb) step
romanToInt (roman=0x80485b8 "MMMCDXLIX") at ReadRoman.c:17
17        int result=0;
(gdb) step
18        int last=0;
(gdb) step
20        for (i=roman;*i != '\0';i++){
(gdb)
```

Wie man sieht gibt der Debugger immer die aktuelle Zeile aus, und beim Funktionsaufruf auch den Wert des Parameters. Wollen wir zusätzlich in den Speicher schauen, so gibt es den Befehl `print`, der dazu da ist, sich den Wert einer Variablen anzuschauen:

```
(gdb) step
21          int current=romanCharToInt(*i);
(gdb) step
romanCharToInt (c=77 'M') at ReadRoman.c:4
4         switch (c){
(gdb) step
11          case 'M': return 1000;
(gdb) step
14      }
(gdb) step
romanToInt (roman=0x80485b8 "MMMCDXLIX") at ReadRoman.c:22
22          if (current>last) result = result-2*last;
(gdb) step
23          result=result+current;
(gdb) step
24          last=current;
(gdb) print result
$1 = 1000
(gdb)
```

Wenn auch mühselig, so können wir so doch ganz genau verfolgen, wie das Programm arbeitet.

Aufgabe 26 Spielen Sie die Session des Debuggers `gdb` aus dem Skript einmal nach und experimentieren sie mit den verschiedenen Befehlen des Debuggers.

3.5.2 Graphische Debugging Werkzeuge

Aufbauend auf die Möglichkeiten des Debuggens über die Kommandozeilenschnittstelle, lassen sich leicht konfortable graphische Debugger implementieren. Ein recht schöner solcher Debugger ist das Programm `ddd`. Hier können Breakpoints mit der Maus im Programmtext gesetzt werden und die verschiedenen Speicherbelegungen betrachtet werden. Abbildung 3.1 zeigt das Programm in Aktion.

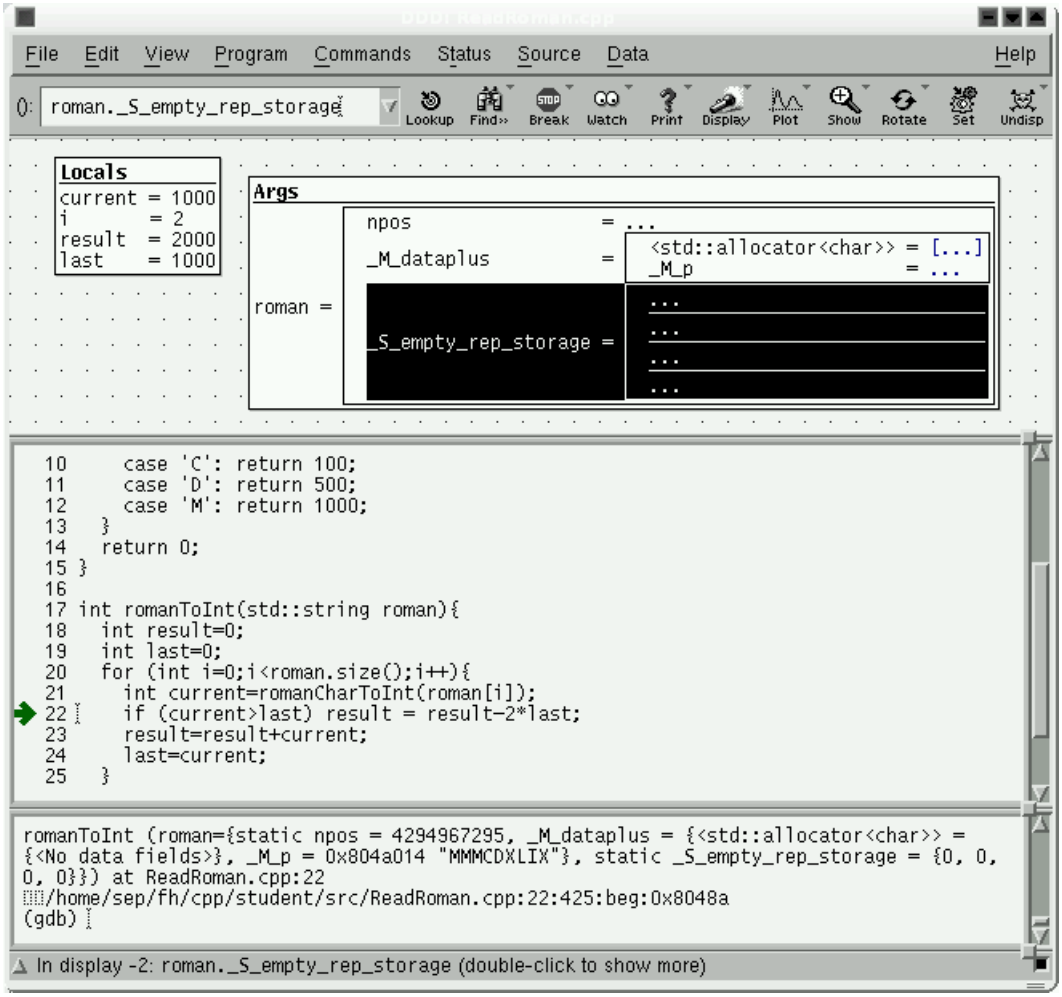


Abbildung 3.1: ddd in Aktion.

Aufgabe 27 Üben Sie den Umgang mit dem Debugger `ddd` anhand des Programms `TestReadRoman`.

3.6 Programmierumgebung: Arbeiten mit Eclipse

In großen Projekten existieren viele hundert Quelltextdateien und Header-Dateien, die alle zum Gesamtprogramm beitragen. Viele Bibliotheken werden benutzt. Es ist dann etwas mühseelig nur mit einem Editor und der Kommandozeile zu arbeiten. Um hier die Arbeit zu erleichtern gibt es Entwicklungsumgebungen. Entwicklungsumgebungen enthalten in der Regel einen integrierten Editor, der eine gute Syntaxhervorhebung durchführt, öffnende und schließende Klammern paart und Fehler direkt im Programmtext markiert. Sie unterstützen den Build-Prozess und übersetzen in der Regel das ganze Projekt bei einer neu abgespeicherten Änderung, so dass Fehler sofort erkannt werden können.

Eine recht mächtige Programmierumgebung ist *eclipse*. Sie ist frei verfügbar und kann unter diesen Link (`http://download.eclipse.org/`) heruntergeladen werden. *eclipse* ist ursprünglich eine von *IBM* zur Entwicklung von Javaprojekten gedachte Entwicklungsplatform. *eclipse* ist in Java entwickelt. Daher braucht man, um mit *eclipse* arbeiten zu können auch die Java-Laufzeitumgebung (`http://java.sun.com/javase/downloads/index.jsp`) .

Durch den stark komponentenbasierten Aufbau der Architektur von *eclipse* ist es möglich Zusätze zu Entwickeln, die aus *eclipse* eine Entwicklungsumgebung für Projekte in anderen Sprachen als Java zu benutzen. Ein solches sogenanntes Plugin ermöglicht die C und C++-Entwicklung mit *eclipse*. Es ist dies das Pugin CDT (`http://www.eclipse.org/cdt/`) .

Eclipse mit CDT ist auf unseren Übungsrechnern installiert. Wenn man eclipse startet, wird zumeist nach einem Arbeitsordner gefragt. In diesem Ordner will eclipse alle seine Projekte verwalten.

Zum Start mit eclipse ist zunächst ein neues Projekt zu definieren. Hierzu wähle man im Menu den Punkt: *new managed c project*. Einem solchem Projekt können nnun neue `.c` und `.h` Quelltextdateien zugefügt wreden, die in einem Editorfenster ediert werden können. In der Regel versucht eclipse sobald eine Datei neu gespeichert wird, mit dem C compiler das Projekt neu zu bauen und zeigt eventuelle Fehler direkt an.

Aufgabe 28 Legen sie ein *eclipse* Projekt an und Programmieren die nächste Aufgaben dieses Blattes mit *eclipse*.

Kapitel 4

Daten

Bisher haben wir außer Zahlen und direkt ausgegebenen Zeichenketten noch keine Daten in unseren Programmen verarbeitet. Daten sind natürlich das A und O der Programmierung. Das spiegelt sich auch in der historischen Entwicklung von Programmiersprachen wieder. Während die Grundkonzepte, wie Funktionen, Ausdrücke und Schleifenbefehle sich schon vor mehreren Jahrzehnten in allen Programmiersprachen auf ähnliche Weise befanden, sind sämtliche Fortschritte in der Programmierung im Bezug auf Definition und Modifikation von Daten bezogen. Objektorientierte Programmierung zeichnet sich als eine datenzentrierte Programmierung aus. Aber auch in funktionalen Programmiersprachen liegt der Fortschritt insbesondere in den sogenannten algebraischen Datentypen, die es erlauben, strukturierte Daten einfach zu definieren, zu erzeugen und zu bearbeiten. Nicht von ungefähr ist mit XML [T. 04] eine der jüngsten Sprachen, die in der Informatik definiert wurde, eine Sprache zum Beschreiben strukturierter Daten entstanden.

In diesem Kapitel werden wir die Möglichkeiten der Definition und Verarbeitung von Daten in C kennenlernen.

4.1 Basistypen

Bevor wir uns strukturierten Daten widmen können, betrachten wir noch einmal die primitiven Basistypen zur Zahlendarstellung. Bisher haben wir uns beim Rechnen auf einen Typen für ganze Zahlen beschränkt, den Typ `int`. C stellt vielerlei Typen zur Repräsentation von Zahlen zur Verfügung.

Die im Folgenden vorgestellten Typen nennt man primitive Typen. Sie sind fest von C vorgegeben.

Um Daten der primitiven Typen aufschreiben zu können, gibt es jeweils Literale für die Werte dieser Typen. Für den Typ `int` haben wir einfach die Zahlen als Literale hinschreiben können.

4.1.1 Zahlenmengen in der Mathematik

In der Mathematik sind wir gewohnt, mit verschiedenen Mengen von Zahlen zu arbeiten:

- **natürliche Zahlen** $\mathbb{N}$: Eine induktiv definierbare Menge mit einer kleinsten Zahl, so daß es für jede Zahl eine eindeutige Nachfolgerzahl gibt.
- **ganze Zahlen** $\mathbb{Z}$: Die natürlichen Zahlen erweitert um die mit einem negativen Vorzeichen behafteten Zahlen, die sich ergeben, wenn man eine größere Zahl von einer natürlichen Zahl abzieht.
- **rationale Zahlen** $\mathbb{Q}$: Die ganzen Zahlen erweitert um Brüche, die sich ergeben, wenn man eine Zahl durch eine Zahl teilt, von der sie kein Vielfaches ist.
- **reelle Zahlen** $\mathbb{R}$: Die ganzen Zahlen erweitert um irrationale Zahlen, die sich z.B. aus der Quadratwurzel von Zahlen ergeben, die nicht das Quadrat einer rationalen Zahl sind.
- **komplexe Zahlen** $\mathbb{C}$: Die reellen Zahlen erweitert um imaginäre Zahlen, wie sie benötigt werden, um einen Wurzelwert für negative Zahlen darzustellen.

Es gilt folgende Mengeninklusion zwischen diesen Mengen:

$$\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$$

Da bereits $\mathbb{N}$ nicht endlich ist, ist keine dieser Mengen endlich.

4.1.2 Zahlenmengen im Rechner

Da wir nur von einer endlich großen Speicherkapazität ausgehen können, lassen sich für keine der aus der Mathematik bekannten Zahlenmengen alle Werte in einem Rechner darstellen. Wir können also schon einmal nur Teilmengen der Zahlenmengen darstellen.

Von der Hardwareseite stellt sich heute zumeist die folgende Situation dar: Der Computer hat einen linearen Speicher, der in Speicheradressen unterteilt ist. Eine Speicheradresse bezeichnet einen Bereich von 32 Bit. Wir bezeichnen diese als ein Wort. Die Einheit von 8 Bit wird als Byte bezeichnet[1]. Heutige Rechner verwalten also in der Regel Dateneinheiten von 32 Bit. Hieraus ergibt sich die Kardinalität der Zahlenmengen, mit denen ein Rechner als primitive Typen rechnen kann. Soll mit größeren Zahlenmengen gerechnet werden, so muß hierzu eine Softwarelösung genutzt werden.

4.1.3 natürliche Zahlen

Natürliche Zahlen werden in der Regel durch Zeichenketten von Symbolen, den Ziffern, eines endlichen Alphabets dargestellt. Die Größe dieser Symbolmenge wird als die Basis b der Zahlendarstellung bezeichnet. Für die Basis b gilt: $b > 1$. Die Ziffern bezeichnen die natürlichen Zahlen von 0 bis $b - 1$.

Der Wert der Zahl einer Zeichenkette $a_{n-1} \ldots a_0$ berechnet sich für die Basis b nach folgender Formel:

$$\sum_{i=0}^{n-1} a_i * b^i = a_0 * b^0 + \ldots + a_{n-1} * b^{n-1}$$

Gebräuchliche Basen sind:

[1] ein anderes selten gebrauchtes Wort aus dem Französischen ist: Oktett

- 2: Dualsystem
- 8: Oktalsystem
- 10: Dezimalsystem
- 16: Hexadezimalsystem[2]

Zur Unterscheidung wird im Zweifelsfalle die Basis einer Zahlendarstellung als Index mit angegeben.

Beispiel 4.1.1 *Die Zahl 10 in Bezug auf unterschiedliche Basen dargestellt:*

$$(10)_{10} = (A)_{16} = (12)_8 = (1010)_2$$

Darüberhinaus gibt es auch Zahlendarstellungen, die nicht in Bezug auf eine Basis definiert sind, wie z.B. die römischen Zahlen.

Betrachten wir wieder unsere Hardware, die in Einheiten von 32 Bit agiert, so lassen sich in einer Speicheradresse durch direkte Anwendung der Darstellung im Dualsystem die natürlichen Zahlen von 0 bis $2^{32} - 1$ darstellen.

C kennt tatsächlich Datentypen um natürliche Zahlen darzustellen. Ein solcher Typ wird mit zwei Wörtern als `unsigned int` bezeichnet.

4.1.4 ganze Zahlen

Zur Darstellung ganzer Zahlen bedarf es einer Darstellung vorzeichenbehafteter Zahlen in den Speicherzellen des Rechners. Es gibt mehrere Verfahren, wie in einem dualen System vorzeichenbehaftete Zahlen dargestellt werden können.

Vorzeichen und Betrag

Die einfachste Methode ist, von den n für die Zahlendarstellung zur Verfügung stehenden Bit eines zur Darstellung des Vorzeichens und die übrigen $n - 1$ Bit für eine Darstellung im Dualsystem zu nutzen.

Beispiel 4.1.2 *In der Darstellung durch Vorzeichen und Betrag werden bei einer Wortlänge von 8 Bit die Zahlen* 10 *und* -10 *durch folgende Bitmuster repräsentiert:* 00001010 *und* 10001010.

Wenn das linkeste Bit das Vorzeichen bezeichnet, ergibt sich daraus, daß es zwei Bitmuster für die Darstellung der Zahl 0 gibt: 10000000 und 00000000.

In dieser Darstellung lassen sich bei einer Wortlänge n die Zahlen von $-2^{n-1} - 1$ bis $2^{n-1} - 1$ darstellen.

Die Lösung der Darstellung mit Vorzeichen und Betrag erschwert das Rechnen. Wir müssen zwei Verfahren bereitstellen: eines zum Addieren und eines zum Subtrahieren (so wie wir in der Schule schriftliches Addieren und Subtrahieren getrennt gelernt haben).

[2] Eine etwas unglückliche Namensgebung aus der Mischung eines griechischen mit einem lateinischen Wort.

Beispiel 4.1.3 *Versuchen wir, das gängige Additionsverfahren für* 10 *und* -10 *in der Vorzeichendarstellung anzuwenden, so erhalten wir:*

0	0	0	0	1	0	1	0
1	0	0	0	1	0	1	0
1	0	0	1	0	1	0	0

Das Ergebnis stellt keinenfalls die Zahl 0 *dar, sondern die Zahl* -20.

Es läßt sich kein einheitlicher Algorithmus für die Addition in dieser Darstellung finden.

Einerkomplement

Ausgehend von der Idee, daß man eine Zahlendarstellung sucht, in der allein durch das bekannte Additionsverfahren auch mit negativen Zahlen korrekt gerechnet wird, kann man das Verfahren des Einerkomplements wählen. Die Idee des Einerkomplements ist, daß für jede Zahl die entsprechende negative Zahl so dargestellt wird, indem jedes Bit gerade andersherum gesetzt ist.

Beispiel 4.1.4 *Bei einer Wortlänge von 8 Bit werden die Zahlen* 10 *und* -10 *durch folgende Bitmuster dargestellt:* 00001010 *und* 11110101.

Jetzt können auch negative Zahlen mit dem gängigen Additionsverfahren addiert werden, also kann die Subtraktion durch ein Additionsverfahren durchgeführt werden.

0	0	0	0	1	0	1	0
1	1	1	1	0	1	0	1
1	1	1	1	1	1	1	1

Das errechnete Bitmuster stellt die negative *Null dar.*

In der Einerkomplementdarstellung läßt sich zwar fein rechnen, wir haben aber immer noch zwei Bitmuster zur Darstellung der 0. Für eine Wortlänge n lassen sich auch wieder die Zahlen von $-2^{n-1}-1$ bis $2^{n-1}-1$ darstellen..

Ebenso wie in der Darstellung mit Vorzeichen und Betrag erkennt man in der Einerkomplementdarstellung am linkesten Bit, ob es sich um eine negative oder um eine positive Zahl handelt.

Zweierkomplement

Die Zweierkomplementdarstellung verfeinert die Einerkomplementdarstellung, so daß es nur noch ein Bitmuster für die Null gibt. Im Zweierkomplement wird für eine Zahl die negative Zahl gebildet, indem zu ihrer Einerkomplementdarstellung noch 1 hinzuaddiert wird.

Beispiel 4.1.5 *Bei einer Wortlänge von 8 Bit werden die Zahlen* 10 *und* -10 *durch folgende Bitmuster dargestellt:* 00001010 *und* 11110110.

Jetzt können weiterhin auch negative Zahlen mit dem gängigen Additionsverfahren addiert werden, also kann die Subtraktion durch ein Additionsverfahren durchgeführt werden.

0	0	0	0	1	0	1	0
1	1	1	1	0	1	1	0
0	0	0	0	0	0	0	0

Das errechnete Bitmuster stellt die Null dar.

Die *negative* Null aus dem Einerkomplement stellt im Zweierkomplement keine Null dar, sondern die Zahl -1, wie man sich vergegenwärtigen kann, wenn man von der 1 das Zweierkomplement bildet.

Das Zweierkomplement ist die in heutigen Rechenanlagen gebräuchlichste Form der Zahlendarstellung für ganze Zahlen. In modernen Programmiersprachen spiegelt sich dieses in den Wertebereichen primitiver Zahlentypen wieder.

Typname	**Länge**	**Wertebereich**
`byte`	8 Bit	$-128 = -2^7$ bis $127 = 2^7 - 1$
`short`	16 Bit	$-32768 = -2^{15}$ bis $32767 = 2^{15} - 1$
`int`	32 Bit	$-2147483648 = -2^{31}$ bis $2147483647 = 2^{32} - 1$
`long`	64 Bit	-9223372036854775808 bis 9223372036854775807

In der Programmiersprache Java sind die konkreten Wertebereiche für die einzelnen primitiven Typen in der Spezifikation festgelegt. In anderen Programmiersprachen wie z.B. C ist dies nicht der Fall. Hier hängt es vom Compiler und dem konkreten Rechner ab, welchen Wertebereich die entsprechenden Typen haben.

Es gibt Programmiersprachen wie z.B. Haskell[PJ03], in denen es einen Typ gibt, der potentiell ganze Zahlen von beliebiger Größe darstellen kann.

Frage eines Softwaremenschen an die Hardwarebauer

Gängige Spezifikationen moderner Programmiersprachen sehen einen ausgezeichneten Wert *nan* für *not a number* in der Arithmetik vor, der ausdrücken soll, dass es sich bei dem Wert nicht mehr um eine darstellbare Zahl handelt. In der Arithmetik moderne Prozessoren ist ein solcher zusätzlicher Wert nicht vorgesehen. Warum eigentlich nicht?! Es sollte doch möglich sein, einen Prozessor zu bauen, der beim Überlauf des Wertebereichs nicht stillschweigend das durch den Überlauf entstandene Bitmuster wieder als Zahl interpretiert, sondern den ausgezeichneten Wert *nan* als Ergebnis hat. Ein Programmierer könnte dann entscheiden, wie er in diesem Fall verfahren möchte. Eine große Fehlerquelle wäre behoben. Warum ist eine solche Arithmetik noch nicht in handelsüblichen Prozessoren verwirklicht?

4.1.5 Kommazahlen

Wollen wir Kommazahlen, also Zahlen aus den Mengen $\mathbb{Q}$ und $\mathbb{R}$, im Rechner darstellen, so stoßen wir auf ein zusätzliches Problem: es gilt dann nämlich nicht mehr, daß ein Intervall nur endlich viele Werte enthält, wie es für ganze Zahlen noch der Fall ist. Bei ganzen Zahlen konnten wir immerhin wenigstens alle Zahlen eines bestimmten Intervalls darstellen. Wir können also nur endlich viele dieser unendlich vielen Werte in einem Intervall darstellen. Wir werden also das Intervall *diskretisieren*.

Festkommazahlen

Vernachlässigen wir für einen Moment jetzt einmal wieder negative Zahlen. Eine einfache und naheliegende Idee ist, bei einer Anzahl von n Bit, die für die Darstellung der Kommazahlen zur Verfügung stehen, einen Teil davon für den Anteil vor dem Komma und den Rest für den Anteil nach dem Komma zu benutzen. Liegt das Komma in der Mitte, so können wir eine Zahl aus n Ziffern durch folgende Formel ausdrücken:

Der Wert der Zahl einer Zeichenkette $a_{n-1} \dots a_{n/2}, a_{n/2-1} \dots a_0$ berechnet sich für die Basis b nach folgender Formel:

$$\sum_{i=0}^{n-1} a_i^{i-n/2} = a_0 * b^{-n/2} + \ldots + a_{n-1} * b^{n-1-n/2}$$

Wir kennen diese Darstellung aus dem im Alltag gebräuchlichen Zehnersystem. Für Festkommazahlen ergibt sich ein überraschendes Phänomen. Zahlen, die sich bezüglich einer Basis darstellen lassen, sind bezüglich einer anderen Basis nicht darstellbar. So läßt sich schon die sehr einfach zur Basis 10 darstellbare Zahl $0,1$ nicht zur Basis 2 als Festkommazahl darstellen, und umgekehrt können Sie einfach zur Basis 2 als Festkommazahl darstellbare Zahlen nicht zur Basis 10 darstellen.

Dem Leser wird natürlich nicht entgangen sein, daß wir keine irrationale Zahl über die Festkommadarstellung darstellen können.

Festkommazahlen spielen in der Rechnerarchitektur kaum eine Rolle. Ein sehr verbreitetes Anwendungsgebiet für Festkommazahlen sind Währungsbeträge. Hier interessieren in der Regel nur die ersten zwei oder drei Dezimalstellen nach dem Komma.

Fließkommazahlen

Eine Alternative zu der Festkommadarstellung von Zahlen ist die Fließkommadarstellung. Während die Festkommadarstellung einen Zahlenbereich der rationalen Zahlen in einem festen Intervall durch diskrete, äquidistant verteilte Werte darstellen kann, sind die diskreten Werte in der Fließkommadarstellung nicht gleich verteilt.

In der Fließkommadarstellung wird eine Zahl durch zwei Zahlen charakterisiert und ist bezüglich einer Basis b:

- die Mantisse für die darstellbaren Ziffern. Die Mantisse charakterisiert die Genauigkeit der Fließkommazahl.
- der Exponent, der angibt, wie weit die Mantisse hinter bzw. vor dem Komma liegt.

Aus Mantisse m, Basis b und Exponent exp ergibt sich die dargestellte Zahl durch folgende Formel:

$$z = m * b^{exp}$$

Damit lassen sich mit Fließkommazahlen sehr große und sehr kleine Zahlen darstellen. Je größer jedoch die Zahlen werden, desto weiter liegen sie von der nächsten Zahl entfernt.

Für die Fließkommadarstellung gibt es in Java zwei Zahlentypen, die nach der Spezifikation des IEEE 754-1985 gebildet werden:

- **float**: 32 Bit Fließkommazahl nach IEEE 754. Kleinste positive Zahl: 2^{-149}. Größte positive Zahl: $(2 - 2^{-23}) * ^{127}$
- **double**: 64 Bit Fließkommazahl nach IEEE 754. Kleinste positive Zahl: 2^{-1074}. Größte positive Zahl: $(2 - 2^{-52}) * ^{1023}$

Im Format für `double` steht das erste Bit für das Vorzeichen, die nächsten 11 Bit markieren den Exponenten und die restlichen 52 Bit kodieren die Mantisse.

Im Format für `float` steht das erste Bit für das Vorzeichen, die nächsten 8 Bit markieren den Exponenten und die restlichen 23 Bit kodieren die Mantisse.

Bestimmte Bitmuster charakterisieren einen Wert für negative und positive unbeschränkte Werte (unendlich) sowie Zahlen, Bitmuster, die charakterisieren, daß es sich nicht mehr um eine Zahl handelt.

4.1.6 Zahlentypen in C

C kennt vier verschiedene Basistypen um Zahlen darzustellen. Zwei Typen zur Darstellung ganzer Zahlen und zwei zur Darstellung von Fließkommazahlen. Eine Übersicht findet sich auch in Anhang B.

int und char

Ganze Zahlen werden mit dem Typ `int` bezeichnet. Wieviel Byte im Speicher zur Darstellung einer ganzen Zahl tatsächlich benutzt wird, ist im C-Standard nicht spezifiziert. Es hängt von der benutzten Hardware und dem benutzten Compiler ab.

Der zweite Typ, der in der Lage ist, ganze Zahlen zu speichern, ist der Typ `char`. Wie der Name (character, Zeichen) schon andeutet, wird dieser Typ primär benutzt um alphabetische Zeichen zu speichern. Er kann aber ebenso zum Rechnen mit Zahlen benutzt werden.

float und double

C bietet zwei Typen zur Speicherung von Fließkommazahlen an: `float` und `double`. Ihr Unterschied liegt in dem Platz, der im Speicher benutzt wird. In der Regel wird für den Typ `float` weniger Speicherplatz benutzt, es kann aber Compiler geben, die für beide Typen die gleiche interne Darstellung benutzen.

Beim Rechnen mit Fließkommazahlen sollte man heutzutage immer den Typ `double` verwenden

Literale

ganzzahlige Literale Ganze Zahlen können in C auf drei verschiedene Weise notiert werden. Diese drei Arten der Literale unterscheiden sich in der Basis zu der die Zahlen notiert sind.

- zur Basis 10 (dezimal): beginnt mit einer Ziffer ungleich 0 gefolgt von einer (möglicherweise leeren) Folge von Ziffern 0–9.
- zur Basis 16 (hexadecimal):beginnt mit `0x` oder `0X` gefolgt von einer nichtleeren Folge von hexadezimalen Ziffern 0–9,A,B,C,D,E,F.

- zur Basis 16 (oktal): beginnt mit einer 0 gefolgt von einer (möglicherweise leeren) Folge von Ziffern 0–7.

So kann die Zahl 63 in C als `63`, `0x3F` und `077` notiert werden.

Den ganzzahligen Literalen kann ein Buchstaben angehänt sein, der den Datentyp des Literals genauer spezifiziert.

`u` oder `U`	markiert das Literal als *unsigned*
`l` oder `L`	markiert das Literal als `long`
`ll` oder `LL`	markiert das Literal als `long long` (erst seit dem Standard C99)

Fließkommaliterale Fließkommaliterale benutzen den Punkt `.` als Dezimalkomma. Mit dem Buchstaben `e` oder `E` kann ein ganzzahliger Exponent notiert werden. So bezeichnen die folgenden Literale Fließkommazahlen vom Typ `double`: `3.1415926535897932`, `4.0`, `6.022e+23`.

Hängt an einem Fließkommaliteral noch der Buchstabe `f`, so wird das Literal als Zahl vom Typ `float` gespeichert.

Zeichenliterale Einzelne Zeichen des Typs `char` können in einfachen Hochkommas eingeschlossen notiert werden. Zeichen, für die es keine einfache sichtbare Repräsentation gibt, wie z.B. das Zeilenendezeichen, werden mit einer Fluchtsequenz beginnend mit dem rückwärtigen Schrägstrich bezeichnet.

Typ Modifizierer

Von den bisher vorgestellten primitiven Typen gibt es modifizierte Versionen. Zum einen können sie modifiziert werden, in der Speichergröße, die für sie verwendet wird, zum anderen ob sie Vorzeichen behaftete Zahlen speichern können oder nicht.

long und short Der Typ `int` kann noch mit den Attribute `long` oder `short` behaftet sein. Das Attribut `long` bewirkt dann, dass auf bestimmten Systemen eventuell mehr Byte zur Darstellung einer Zahl genutzt werden. Garantiert doppelt so viel Byte wie ein `int` hat schließlich der Typ `long long int`.

Das Attribut `short` signalisiert, dass nur halb soviel Byte zu Zahlendarstellung genutzt werden sollen, wie für normale `int` Zahlen. Ebenso kann ein System einen `long double` Typ zur Verfügung stellen, der größer ist als der normale `double` Datentyp.

unsigned und signed Die Datentypen für ganze Zahlen können so modifiziert werden, dass in Ihnen nur natürliche Zahlen (inklusive 0) gespeichert werden können. Dieses geschieht durch das Attribut `unsigned`. Ebenso mit dem Attribute `signed` signalisiert werden, dass z.B. beim Datentyp `char` auch negative Zahlen darstellbar sein sollen. Für die Typen `float` und `double` führt das Attribut `unsigned` zu einem Fehler des Compilers.

weitere Literalschreibweisen Den ganzzahligen Literalen kann ein Buchstaben angehänt sein, der den Datentyp des Literals genauer spezifiziert.

`u` oder `U`	markiert das Literal als *unsigned*
`l` oder `L`	markiert das Literal als `long`
`ll` oder `LL`	markiert das Literal als `long long` (erst seit dem Standard C99)

der Operator sizeof Wie zu sehen war, ist es gar nicht genau festgelegt, wie viel Byte zur Darstellung der einzelnen primitiven Typen benutzt werden. Das kann von System zu System variieren. Will man wissen, wieviel Speicherplatz für bestimmte Datentypen tatsächlich bereit gehalten werden, so bietet C eine eingebaute Funktion an, die dieses Information bereit hält: die Funktion `sizeof`. Ihr kann als Parameter ein primitiver Typ übergeben werden. Als Rückgabe erhält man die Anzahl der Byte, die für diesen Typ zur Darstellung verwendet werden.

printf

Zur Ausgabe von Zahlen benutzen wir die Funktion `printf`. Bisher haben wir als Steuerzeichen für `printf` nur `%i` benutzt um ganze Zahlen auszugeben. Die Prozedur `printf` ist eine sehr ausgeklügelte Prozedur. In ihr kann sehr genau spezifiziert werden, auf welche Weise Zahlen formattiert werden sollen. Zusätzlich zum angegebenen Zahlentypen der dargestellt werden soll, kann auch angegeen werden, wieviel Platz zur Darstellung der Zahl benutzt werden soll. Folgende Tabelle gibt einen kleinen Überblick:

Zeichen	**Argumenttyp**	**darstellt als**
d,i	int	Dezimal mit Vorzeichen.
o	int	Oktal ohne Vorzeichen (ohne führende Null).
x,X	int	Hexadezimal ohne Vorzeichen (ohne führendes 0x oder 0X) mit abcdef bei 0x oder ABCDEF bei 0X.
u	int	Dezimal ohne Vorzeichen.
c	int	einzelnes Zeichen, nach Umwandlung in unsigned char.
f	double	Dezimal als [-]mmm.ddd, wobei die Genauigkeit die Anzahl der d festlegt. Voreinstellung ist 6; bei 0 entfällt der Dezimalpunkt.
e,E	double	Dezimal als [-]m.dddddde±xx oder [-]m.ddddddE±xx, wobei die Genauigkeit die Anzahl der d festlegt. Voreinstellung ist 6; bei 0 entfällt der Dezimalpunkt.
g,G	double	%e oder %E wird verwendet, wenn der Exponent kleiner als -4 oder nicht kleiner als die Genauigkeit ist; sonst wird %f benutzt. Null und Dezimalpunkt am Schluß werden nicht ausgegeben.

Es ist an der Zeit, all das in diesem Kapitel angesprochene einmal auszuprobieren. Im folgenden Programm werden von den unterschiedlichen primitiven Typen einmal Variablen angelegt.

ArithTypes.c

```
1  #include <stdio.h>
2
3  int main(){
4    char c   = 'a';
5    int  i   = 42;
6    float f  = 42.0f;
7    double d = 42.0;
8
9    long int  li   = 42;
10   long long int  lli   = 42;
11   long double ld = 42.0;
12
13   short int  si   = 42;
14
15   unsigned char uc   = 'a';
16   unsigned int  ui   = 42;
17
18   signed char sic   = 'a';
```

```
19    signed int  sii   = 42;
20
21    printf("sizeof(c): %3i value: %10c\n",sizeof(c),c);
22    printf("sizeof(i): %3i value: %10i\n",sizeof(i),i);
23    printf("sizeof(f): %3i value: %10f\n",sizeof(f),f);
24    printf("sizeof(d): %3i value: %10f\n",sizeof(d),d);
25    printf("\n");
26    printf("sizeof(li): %2i value: %10i\n",sizeof(li),li);
27    printf("sizeof(lli):%2i value: %10i\n",sizeof(lli),lli);
28    printf("sizeof(ld): %2i value: %10f\n",sizeof(ld),ld);
29    printf("\n");
30    printf("sizeof(si): %2i value: %10i\n",sizeof(si),si);
31    printf("\n");
32    printf("sizeof(uc): %2i value: %10u\n",sizeof(uc),uc);
33    printf("sizeof(ui): %2i value: %10u\n",sizeof(ui),ui);
34    printf("\n");
35    printf("sizeof(sic): %1i value: %10i\n",sizeof(sic),sic);
36    printf("sizeof(sii): %1i value: %10i\n",sizeof(sii),sii);
37
38    return 0;
39  }
```

Die Ausgabe des Programms gibt darüber Aufschluß, wie viel Byte für die einzelnen unterschiedlichen Datentypen der `gcc` auf Linux belegt.

```
sep@pc305-3:~/fh/c/tutor> bin/ArithTypes
sizeof(c):   1 value:          a
sizeof(i):   4 value:         42
sizeof(f):   4 value:  42.000000
sizeof(d):   8 value:  42.000000

sizeof(li):  4 value:         42
sizeof(lli): 8 value:         42
sizeof(ld): 12 value:  -0.000000

sizeof(si):  2 value:         42

sizeof(uc):  1 value:         97
sizeof(ui):  4 value:         42

sizeof(sic): 1 value:         97
sizeof(sii): 4 value:         42
sep@pc305-3:~/fh/c/tutor>
```

Rechnen

ganze Zahlen Auf dem Typen `int` haben wir bereits die bekannten arithmetischen Operatoren kennengelernt. Diese existieren auch für die anderen Zahlentypen und erstaunlicher Weise auch für den Typen `char`. So kann man eine Variable sehr schön von A bis Z laufen lassen:

ABisZ.c

```
1  #include <stdio.h>
2  int main(){
```

```
3   char abstand='A'-'a';
4   for (char c='A';c<='Z';c++) printf("%c",c-abstand);
5   printf("\n");
6   return 0;
7 }
```

Typkonversion Zahlen, die von einem Typ mit einem kleineren Zahlenbereich sind, können problemlos als Zahlen eines Typs mit einem größeren Zahlenbereich betrachtet werden. Umgekehrt ist das nicht so. Eine `int`-Zahl kann so zum Beispiel zu groß sein, um als ein `char`-Wert gespeichert zu werden. Es ist also Vorsicht walten zu lassen, wenn man zwischen den verschiedenen Typen die Daten hin und her bewegt.

Es gibt eine explizite Notation, um eine Zahl zu einem anderen Zahlentypen zu konvertieren. Hierzu wird in runden Klammern der neue gewünschte Typ für einen Ausdruck vor diesen geschrieben. Diese Operation wird als *cast* bezeichnet. Man muß bei dieser expliziten Typkonvertierung genau darauf acht geben, welchen Ausdruck man genau konvertieren möchte. So bedeutet `(double)(3/2)`, dass erst die ganzzahlige Division durchzuführen ist und dieses Ergebnis dann als Fließkommazahl betrachtet wird; dahingegen bedeutet `(double)3/2)`, dass zunächst die `3` als Fließkommazahl zu betrachten ist und dann für Fließkommazahlen die Division durchzuführen ist.

Man überzeuge sich davon im folgenden textprogramm:

Convert.c

```
1  #include <stdio.h>
2
3  int main(){
4    int x = 224;
5    double d = x;
6    char c = x;
7
8    printf("%i\n",x);
9    printf("%f\n",d);
10   printf("%i\n",c);
11
12   printf("%f\n",(double)(3/2));
13   printf("%f\n",(double)3/2);
14   printf("%f\n",3/2);
15   printf("%i\n",3/2);
16   printf("%i\n",(int)(3.0/2.0));
17   return 0;
18 }
```

Die Ausgabe offenbahrt den fundamentalen unterschied.

```
sep@pc305-3:~/fh/c> ./Convert
224
224.000000
-32
1.000000
1.500000
```

```
1.500000
1
sep@pc305-3:~/fh/c>
```

Tatsächlich ist meine persönliche Erfahrung: bei einem *cast* lieber eine Klammer zu viel setzen als zu wenig.

Fließkommazahlen Der folgende Test zeigt, daß bei einer Addition von zwei Fließkommazahlen die kleinere Zahl das Nachsehen hat:

DoubleTest.c

```
1  #include <stdio.h>
2  int main(){
3    double x = 325e200;
4    double y = 325e-200;
5    double z = -325e+200;
6    printf("x: %6.20e\n",x);
7    printf("y: %6.20e\n",y);
8    printf("x+y: %6.20e\n",(x+y));
9    printf("x+z: %6.20e\n",(x+z));
10   printf("x+100000: %6.20e\n",x+100000.0);
11   return 0;
12 }
```

Wie man an der Ausgabe erkennen kann: selbst die Addition der Zahl `100000` bewirkt keine Veränderung auf einer großen Fließkommazahl:

```
sep@pc305-3:~/fh/c/student> ./DoubleTest
x: 3.2499999999999978946e+202
y: 3.2500000000000017969e-198
x+y: 3.2499999999999978946e+202
x+z: 0.0000000000000000000e+00
x+100000: 3.2499999999999978946e+202
sep@pc305-3:~/fh/c/student>
```

Das folgende kleine Beispiel zeigt, inwieweit und für den Benutzer oft auf überraschende Weise die Fließkommadarstellung zu Rundungen führt:

Rounded.c

```
1  #include <stdio.h>
2  int main(){
3    printf("8.0f =%20f \n",8.0f);
4    printf("88.0f =%20f \n",88.0f);
5    printf("888.0f =%20f \n",888.0f);
6    printf("8888.0f =%20f \n",8888.0f);
7    printf("88888.0f =%20f \n",88888.0f);
8    printf("888888.0f =%20f \n",888888.0f);
9    printf("8888888.0f =%20f \n",8888888.0f);
10   printf("88888888.0f =%20f \n",88888888.0f);
11   printf("888888888.0f =%20f \n",888888888.0f);
12   printf("8888888888.0f =%20f \n",8888888888.0f);
```

```
13    printf("88888888888.0f =%20f \n",88888888888.0f);
14    printf("888888888888.0f =%20f \n",888888888888.0f);
15
16    printf("1.0f+1000000000000.0f-1000000000000.0f =%15f \n"
17          ,1.0f+1000000000000.0f-1000000000000.0f);
18    return 0;
19  }
```

Das Programm hat die folgende Ausgabe. Insbesondere in der letzten Zeile fällt auf, daß Addition und anschließende Subtraktion ein und derselben Zahl nicht die Identität ist. Für Fließkommazahlen gilt nicht: $x + y - y = x$.

```
sep@pc305-3:~/fh/c/student> ./bin/Rounded
8.0f =            8.000000
88.0f =           88.000000
888.0f =          888.000000
8888.0f =         8888.000000
88888.0f =        88888.000000
888888.0f =       888888.000000
8888888.0f =      8888888.000000
88888888.0f =     88888888.000000
888888888.0f =    888888896.000000
8888888888.0f =   8888889344.000000
88888888888.0f =  88888885248.000000
888888888888.0f = 888888885248.000000
1.0f+1000000000000.0f-1000000000000.0f =       0.000000
sep@pc305-3:~/fh/c/student>
```

Bibliothek mathematischer Funktionen Oft reichen die Grundrechenarten nicht aus. Insbesondere in der graphischen Datenverarbeitung muß an in der Lage sein, die trigeometrischen Funktionen wie $\cos, \sin, \tan, \ldots$ für Zahlen zu berechnen. Desweiteren werden oft die Zahl π benötigt. Hierzu gibt es in C eine Standardbibliothek, in der eine Vielzahl von mathematischen Funktionen enthalten sind. Die Bibliothek ist in der Kopfdatei `math.h` definiert.

MathExample.c

```
1  #include <stdio.h>
2  #include <math.h>
3  int main(){
4    printf("%f\n",sin(M_PI));
5    printf("%f\n",sin(90));
6    printf("%f\n",sqrt(64));
7    printf("%f\n",pow(2,10));
8    return 0;
9  }
```

Tatsächlich kann es zu Fehlern kommen, wenn der Linker ein Programm mit den mathematischen Funktionen linken soll.

```
sep@pc305-3:~/fh/c/tutor> gcc src/MathExample.c
/tmp/ccs63nup.o(.text+0x23): In function ‘main’:
: undefined reference to ‘sin’
/tmp/ccs63nup.o(.text+0x51): In function ‘main’:
```

```
: undefined reference to ‘sin’
/tmp/ccs63nup.o(.text+0x7f): In function ‘main’:
: undefined reference to ‘sqrt’
/tmp/ccs63nup.o(.text+0xb9): In function ‘main’:
: undefined reference to ‘pow’
collect2: ld returned 1 exit status
sep@pc305-3:~/fh/c/tutor>
```

Hierzu ist dem Compiler für den Linker noch die Information mitzugeben, dass die entsprechende mathematische Standardbibliothek mit einzulinken ist. Das entsprechende *Flag* heisst: `-lm`.

```
sep@pc305-3:~/fh/c/tutor> gcc -lm src/MathExample.c
sep@pc305-3:~/fh/c/tutor>
```

Aufgabe 29 Schreiben Sie eine Funktion
`double kreisUmfang(double radius);`
Sie soll für einen gegebenen Radius den Umfang eines Kreises berechnet.

Lösung

Umfang.c

```
1  #include <stdio.h>
2  #include <math.h>
3
4  double kreisUmfang(double radius){
5    return 2*M_PI*radius;
6  }
7
8  int main(){
9    printf("der Umfang bei einem radius 5 ist: %f\n",kreisUmfang(5));
10   return 0;
11 }
```

Aufgabe 30 Schreiben Sie eine Funktion
`double kreisFlaeche(double radius);`
Sie soll für einen gegebenen Radius den Flächeninhalt eines Kreises berechnet.

Lösung

Flaeche.c

```
1  #include <stdio.h>
2  #include <math.h>
3
4  double kreisFlaeche(double radius){
5    return M_PI* pow(radius,2.0);
6  }
7
8  int main(){
9    printf("die Flaeche bei einem radius 5 ist: %f\n",kreisFlaeche(5));
```

```
10    return 0;
11  }
```

Aufgabe 31 (1 Punkt) Schreiben Sie eine Prozedur
`void printDual(int i);`
Sie soll den Parameter `i` als Dualzahl auf der Konsole ausdrucken.

Hinweis: Leiten Sie Ihre Lösung aus der Lösung für die Funktion `alsText` ab. Jetzt ist eine Funktion:
`int hoechsteZweierPotenz(int i);`
hilfreich.

Lösung

AlsDual.c

```
1   #include <stdio.h>
2
3   int hoechsteZweierPotenz(int i){
4     int result = 1;
5     while (i>=2){
6       result=result*2;
7       i=i/2;
8     }
9     return result;
10  }
11
12  void printDual(int i){
13    if (i<0){
14      printf("-");
15      i= -i;
16    }
17
18    for (int hoechsteStelle = hoechsteZweierPotenz(i)
19        ;hoechsteStelle>0
20        ;hoechsteStelle=hoechsteStelle/2) {
21      printf("%i",i/hoechsteStelle);
22      i=i%hoechsteStelle;
23    }
24    printf("\n");
25  }
26
27  int main(){
28    printDual(-142);
29    printDual(1024);
30    printDual(0);
31    printDual(-0);
32    printDual((17+4)*2);
33    return 0;
34  }
```

4.2 Neue Namen für Typen einführen

Wir schon im letzten Abschnitt zu sehen war, können in C Typennamen aus mehreren Wörtern bestehen. Wie noch zu sehen wird, können in C Typnamen noch weitgehend komplizierter zu notieren sein. Daher ist es eine schöne Eigenschaft von C, dass man neue Namen für Typen definieren kann. Hierzu gibt es das `typedef`-Konstrukt. Nach dem Wort `typedef` ist der Typ zu schreiben, für den ein neuer Name eingeführt wird. Schließlich ist der neue Typnme zu schreiben.

Im folgenden kleinen Programm wird der neue Name `nat` für den Typ `unsigned int` eingeführt. Von da an kann im Programm der neue Name `nat` statt des komplizierteren Namen `unsigned int` benutzt werden.

Nat.c

```
1 #include <stdio.h>
2
3 typedef unsigned int nat;
4
5 int main(){
6   nat n=42;
7   printf("n = %u\n",n);
8   return 0;
9 }
```

Das `typedef`-Konstrukt in C ist äußerst sinnvoll. der Vorteil liegt nicht nur in einfacheren sprechenden Namen, die für Typen eingeführt werden können. Es ist so auch möglich den physikalischen Typ flexibel zu halten. So kann ein Name für

4.3 Strukturen

Bisher haben wir gänzlich auf den eingebauten primitiven Typen in C für Zahlen aufgebaut. Bei der Programmierung von Anwendungen werden zumeist aber Daten bearbeitet, die nicht nur aus einer Zahl bestehen, sondern oft ein Tupel aus mehereren Zahlen sind. So könnte es z.B. sein, dass wir als Aufgabe ein Programm für eine Wetterstation schreiben sollen. Die Wetterstation ermittelt dabei mehrere unterschiedliche Messwerte. Es wird z.B. Temperatur, Luftdruck, Luftfeuchtigkeit und die Windstärke gemessen. Die Wetterstation liefert zu einem Zeitpunkt also vier Meßzahlen mit unterschiedlicher Bedeutung. Diese vier Zahlen zusammen sind damit ein Datensatz der Wetterstation. Gerne wollen wir diese vier Zahlen nun auch als ein einziges Datenobjekt betrachten können. Hierzu bietet C die Möglichkeit über eine Struktur, den sogenannten *structs* einen neuen Typ zu definieren, der aus mehreren bereits bestehenden Typen zusammengesetzt ist.

4.3.1 Definition von Strukturen

Eine Struktur wird definiert mit dem Schlüsselwort `struct`, dann folgt der frei wählbare Name, den die Struktur haben soll. Anschließend wird in geschweiften Klammern der Inhalt der Struktur definiert. Der Inhalt sind im Prinzip die Definition struktureigener Variablen.

So kann die Struktur definiert werden, die die vier verschiedenen Messwerte einer Wetterstation zusammenfasst:

Messwerte.c

```
1  #include <stdio.h>
2
3  struct Messwert{
4    int temperatur;
5    int luftfeuchtigkeit;
6    int luftdruck;
7    int windstaerke;
8  };
```

Die Definition einer Struktur endet immer noch mit einem abschließenden Semikolon. Dieses wird häufig vergessen und führt mitunter zu verwirrenden Fehlermeldungen.

Strukturen werden in anderen Programmiersprachen üblicher Weise auch als `record` bezeichnet.

Die einzelnen Teile einer Struktur werden auch als ihre Attribute oder auch als ihre Felder bezeichnet. In dieser Terminologie hat die Struktur `Messwert` vier Felder, bzw. vier Attribute.

Nachdem einmal eine Struktur definiert wurde, steht sie als neuer Typname zur Verfügung. Der neue Typname ist dann in unserem Beispiel: `struct Messwert`, besteht also aus zwei Wörtern, so wie ja auch schon der Typname für natürliche Zahlen aus zwei den Wörtern `unsigned int` besteht.

Dieser neue eigene Typ kann nun genauso verwendet werden, wie z.B. Typ `int` bisher: Variablen von diesem Typ können definiert werden und Parameter von diesem Typ können definiert werden.

4.3.2 Erzeugen von Strukturobjekten

Wir haben uns angewöhnt, möglichst gleich bei der Delkaration einer Variablen, dieser Variablen auch einen Wert zu geben. Das geht auch mit Variablen von einem `struct`-Typ. Hierzu sind dann die Werte der einzelnen Attribute aufzuzählen. Diese Aufzählung ist durch Komma getrennt innerhalb geschweifter Klammern vorzunehmen. Die Reihenfolge der einzelnen Werte entspricht der Reihenfolge, in der die Attribute in der Struktur definiert wurden.

Messwerte.c

```
9  int main(){
10   struct Messwert m1 = {22,41,997,8};
```

Hiermit wurde ein Objekt der Struktur `Messwert` angelegt, das eine Temperaturwert von `22`, Feuchtigkeitswert von `41`, einen Luftdruck von `997` und eine Windstärke von `8` speichert.

Ebenso wie bei einfachen Variablen vom Typ `int` können wir zunächst aber auch erst einmal die Variable deklarieren und dann schrittweise die einzelnen Werte zuweisen. Hierzu gibt es den Punkt-Operator, mit dem man auf die einzelnen Attribute einer Struktur einzeln zugreifen und diese dann einzeln auch mit Werten belegen kann:

Messwerte.c

```
11   struct Messwert m2;
12   m2.temperatur=17;
13   m2.luftfeuchtigkeit=57;
14   m2.luftdruck=1004;
15   m2.windstaerke=4;
```

Hiermit wurde ein zweites Objekt der Struktur Messwert angelegt und die einzelnen Felder mit Werten belegt.

4.3.3 Zugriff auf Strukturkomponenten

Die Punkt-Notation kann natürlich nicht nur genutzt werden, um den einzelnen Felder einer Struktur Werte zuzuweisen, sondern auch um auf die darin gespeicherten Werte zuzugreifen. So können wir die Werte einzelner Felder unserer Strukturobjekte ausgeben:

Messwerte.c

```
16   printf("temperatur: %i\n",m1.temperatur);
17   printf("m1.luftdruck: %i\n",m1.luftdruck);
18   m1.temperatur = m1.temperatur-2;
19   printf("m1.temperatur: %i\n",m1.temperatur);
20   printf("m2.temperatur: %i\n",m2.temperatur);
21   return 0;
22 };
```

4.3.4 Strukturen als Funktionsparameter

Wir haben gelernt, dass C bei Funktionsaufrufen eine Parameterübergabe per Wert vornimmt. Das ist für einfache Zahlen, also `int`-Parameter recht einsichtig. Wie sieht dieses aber für Variablen eines Strukturtyps aus? Wir können dies einmal mit einer simplen Struktur testen. Hierzu sei eine Struktur definiert, in der die `x`- und `y`-Koordinate eines Punktes im zweidimensionalen Raum gespeichert werden können:

Punkt1.c

```
1  #include <stdio.h>
2  struct Punkt{
3    int x;
4    int y;
5  };
```

Für diese Struktur seien ein Paar Funktionen definiert. Zunächst eine schöne Ausgabe eines Punktes auf der Kommandozeile:

Punkt1.c

```
6  void printPunkt(struct Punkt p){
7    printf("(%i,%i)\n",p.x,p.y);
8  }
```

Eine zweite Funktion soll einen Punkt um eins auf der x-Achse und um 2 auf der y-Achse verschieben.

Punkt1.c

```
9  void verschiebe(struct Punkt p){
10   p.x=p.x+1;
11   p.y=p.y+2;
12 }
```

Nun seien die beiden Funktionen einmal ausgetestet. Hierzu legen wir einen Punkt an, geben diesen aus, verschieben ihn und geben ihn erneut aus:

Punkt1.c

```
13 int main(){
14   struct Punkt p={17,4};
15   printPunkt(p);
16   verschiebe(p);
17   printPunkt(p);
18   return 0;
19 }
```

Tatsächlich wird zweimal der gleiche Punkt ausgegeben:

Die Funktion `verschiebe` hat den Punkt `p` gar nicht verändert. Was man hier sieht, ist, dass auch Strukturobjekte in C per Wert übergeben werden. Da eine Struktur über ihre Felder mehrere Werte repräsentiert, werden also tatsächlichen sämtliche Werte eines Strukturobjektes übergeben. Anders ausgedrückt: es wird eine komplette Kopie des Strukturobjektes vorgenommen.[3] Jeder Aufruf einer Funktion in C mit einer Struktur als Parameter führt zur Erzeugung einer Kopie dieser Struktur. Die Kopie ist dann vollkommen unabhängig zum Originalstrukturobjekt.

Da Parameter auch nur eine Form von Variable sind, gilt obiges in gleicher Weise für die Zuweisung einer Variable. Auch dabei wird eine Kopie der Daten vorgenommen.

Hierzu betrachte man folgendes Programm:

Punkt2.c

```
20 #include <stdio.h>
21 struct Punkt{
22   int x;
23   int y;
24 };
25 void printPunkt(struct Punkt p){
26   printf("(%i,%i)\n",p.x,p.y);
27 }
28 int main(){
29   struct Punkt p1={17,4};
30   struct Punkt p2=p1;
31   printPunkt(p1);
32   printPunkt(p2);
33   p2.x=0;
34   printPunkt(p1);
35   printPunkt(p2);
36   return 0;
37 }
```

[3] Bei Javaprogrammierern, die Neulinge in C sind, führt in der Regel das obige Testprogramm zu einiger Bestürzung.

Mit der Zuweisung `p2=p1` werden die Werte des Strukturobjekts `p1` genommen um ein neues Strukturobjekt `p2` zu erzeugen. Von da an, haben `p1` und `p2` nichts mehr miteinander zu tun. Wird eine der Variablen verändert, so bleibt die andere davon unberührt, wie an der Ausgabe des Programms zu sehen ist:

```
sep@pc305-3:~/fh/c/tutor> bin/Punkt2
(17,4)
(17,4)
(17,4)
(0,4)
sep@pc305-3:~/fh/c/tutor>
```

4.3.5 Einwortnamen für Strukturen

Die Typnamen von Strukturen bestehen in C aus zwei Wörtern: dem Wort `struct` und dem eigentlichen Namen der Struktur. Das ist relativ umständlich. Man kann sich hier das Leben durch einen Trick vereinfachen, indem man gleichzeitig mit der Deklaration einer Struktur über das `typedef`-Konstrukt einen Kurznamen für diese Struktur erfindet. Hierzu betrachte man noch einmal das Beispiel der simplen Struktur für zweidimensionale Punkte.

Punkt3.c

```
38 #include <stdio.h>
39
40 typedef struct {
41   int x;
42   int y;
43 } Punkt;
44
45 void printPunkt( Punkt p){
46   printf("(%i,%i)\n",p.x,p.y);
47 }
48
49 int main(){
50   Punkt p={17,4};
51   printPunkt(p);
52   return 0;
53 }
```

Hier konnte bei der Deklaration von Variablen des Typs `Punkt` auf das Wort `struct` verzichtet werden.

Aufgabe 32 (1 Punkt)
Auf diesen Übungsblatt sollen Sie eine kleine Bibliothek für komplexe Zahlen schreiben und austesten. Komplexe Zahlen werden meist in der Form $a + b * i$ dargestellt, wobei a und b reelle Zahlen sind und i die imaginäre Einheit ist. a wird dabei als Realteil und b als Imaginärteil bezeichnet.

a) Schreiben Sie eine Struktur, mit der komplexe Zahlen dargestellt werden kann. Die Struktur soll unter den Typnamen `Complex` benutzbar sein.

b) Schreiben Sie eine Funktion `void printComplex(Complex c)`, die eine komplexe Zahl auf der Kommandozeile ausgibt.

c) Für komplexe Zahlen sind Addition und Multiplikation wie folgt definiert:

- $(a + b\,\mathrm{i}) + (c + d\,\mathrm{i}) = (a + c) + (b + d)\,\mathrm{i}$
- $(a + b\,\mathrm{i}) - (c + d\,\mathrm{i}) = (a - c) + (b - d)\,\mathrm{i}$
- $(a + b\,\mathrm{i}) \cdot (c + d\,\mathrm{i}) = (ac - bd) + (ad + bc) \cdot \mathrm{i}$
- $\frac{a+b\,\mathrm{i}}{c+d\,\mathrm{i}} = \frac{(a+b\,\mathrm{i})(c-d\,\mathrm{i})}{(c+d\,\mathrm{i})(c-d\,\mathrm{i})} = \frac{ac+bd}{c^2+d^2} + \frac{bc-ad}{c^2+d^2} \cdot \mathrm{i}$

Implementieren Sie die Funktionen `add`, `sub`, `mult` und `div`, die zwei komplexe Zahlen als Argument haben und eine komplexe Zahl nach obigen Definitionen als Ergebnis zurückgeben.

d) Die Norm einer komplexen Zahl erhält man, wenn zum Quadrat des Realteils das Quadrat des Imaginärteils addiert wird. Schreiben Sie eine Funktion `norm`, die für eine komplexe Zahl ihre Norm berechnet.

e) Für die Erzeugung der als Apfelmännchen bekannten Fraktale spielt die folgende Rekursionsgleichung auf komplexen Zahlen eine entscheidene Rolle: $z_{n+1} = z_n^2 + c$, wobei z_0 die komplexe Zahl $0 + 0i$ mit dem Real- und Imaginärteil 0 ist.
Schreiben Sie eine Funktion
`unsigned int schwelle(Complex c,double schwellwert)`
die das kleinste n berechnet, für das gilt: $norm(z_n) > schwellwert$.

f) Schreiben Sie umfangreiche Tests in einer Funktion `main`.

Lösung

Complex.h

```
1  #ifndef COMPLEX_H__
2  #define COMPLEX_H__
3
4  typedef struct{
5    double real;
6    double imag;
7  } Complex;
8
9  void printComplex(Complex c);
10
11 Complex add(Complex x,Complex y);
12 Complex sub(Complex x,Complex y);
13 Complex mult(Complex x,Complex y);
14 Complex divC(Complex x,Complex y);
15 double norm(Complex x);
16 unsigned int schwelle(Complex c,double schwellwert);
17
18
19 Complex mapComplex(Complex c,double f(double x));
20 #endif
```

Complex.c

```
1  #include "Complex.h"
2  #include <stdio.h>
3
4  void printComplex(Complex c){
5    printf("(%f,%f)",c.real,c.imag);
6  }
7
8  Complex add(Complex x,Complex y){
9    Complex result = {x.real+y.real,x.imag+y.imag};
10   return result;
11 }
12 Complex sub(Complex x,Complex y){
13   Complex result = {x.real-y.real,x.imag-y.imag};
14   return result;
15 }
16 Complex mult(Complex x,Complex y){
17   Complex result = {x.real*y.real-x.imag*y.imag
18                    ,x.real*y.imag+x.imag*y.real};
19   return result;
20 }
21 Complex divC(Complex x,Complex y){
22   Complex result = { (x.real*y.real+x.imag*y.imag)
23                     /(y.imag*y.imag+y.real*y.real)
24                    , (x.imag*y.real-x.real*y.imag)
25                     /(y.imag*y.imag+y.real*y.real)};
26   return result;
27 }
28
29 double norm(Complex x){
30   return x.real*x.real+x.imag*x.imag;
31 }
32 unsigned int schwelle(Complex c,double schwellwert){
33   unsigned int result = 0;
34   Complex z = {0,0};
35   while(!(norm(z)> schwellwert) && result<50){
36     z=add(mult(z,z),c);
37     result=result+1;
38   }
39   return result;
40 }
41
42 Complex mapComplex(Complex c,double f(double x)){
43   Complex result={f(c.real),f(c.imag)};
44   return result;
45 }
46
```

TestComplex.c

```
#include "Complex.h"
#include <stdio.h>

void nl(){printf("\n");}

double doppel(double d){return 2*d;}

int main(){
  Complex c1={-0.3,-0.76868};
  printComplex(c1);
  nl();
  printComplex(add(c1,c1));
  nl();
  printComplex(mult(c1,c1));
  nl();
  printComplex(divC(c1,c1));
  nl();
  printComplex(sub(c1,c1));
  nl();
  printf("Norm(c1) = %f\n",norm(c1));
  printf("schwelle(c1,4): %i\n",schwelle(c1,4));
  printComplex(mapComplex(c1,doppel));
  nl();
  return 0;
}

```

4.4 Aufzählungen

Oft kommt es vor, dass man Daten hat, von denen es eine endliche Menge unterschiedlicher Ausprägung gibt. Diese stehen dann oft auch in einer bestimmten Reihenfolge. Ein typisches solches Beispiel sind die sieben Wochentage. Ein Wochentag ist genau einer dieser sieben Werte, und die sieben Werte stehen in einer festen Reihenfolge.

Um solche Daten adequat auszudrücken, gibt es in C die Möglichkeit Aufzählungen zu deklarieren.

4.4.1 Definition von Aufzählungen

Die Deklaration einer Aufzählung beginnt mit dem Wort `enum`, dann folgt der Name, den der Aufzählungstyp haben soll. Schießlich kommen in geschweiften Klammern per Komma getrennt die verschiedenen Namen der Aufzählungselemente. So läßt sich eine Aufzählung für die sieben Wochentage wie folgt definieren.

Tage.c

```
#include <stdio.h>
enum Tag
 {Montag,Dienstag,Mittwoch,Donnerstag,Freitag,Sonnabend,Sonntag};
```

Wie auch schon bei Strukturen sind Aufzählungstypen Typen, die aus zwei Wörtern bestehen: erst das Wort `enum` und dann der eigentliche Name des Typs. Es lassen sich nun die Namen der einzelnen Elemente als Werte dieses neuen Typs benutzen.

Tage.c

```
4  int main(){
5    enum Tag heute = Montag;
```

Ein wenig Enttäuschung macht sich breit, wenn wir anfangen mit diesen Werten zu rechnen. Tatsächlich sind die Werte einer Aufzählung nichts weiter als einfache natürliche Zahlen. Wir können auf diesen Zahlen rechnen, können Werte bekommen, die gar nicht in den Aufzählungsrahmen passen und können Aufzählungsvariablen auch einfach mit willkürlichen Werten belegen, die gar nichts mit der Aufzählung zu tun haben.

Tage.c

```
6    printf("heute ist %i\n",heute);
7    heute=heute+1;
8    printf("heute ist %i\n",heute);
9    heute=heute+6;
10   printf("heute ist %i\n",heute);
11   heute = 42;
12   printf("heute ist %i\n",heute);
13   return 0;
14 }
```

Die Ausgabe dieses Programms zeigt deutlich, dass wir es nur mit einfachen Zahlen zu tun haben.

```
sep@pc305-3:~/fh/c/student> bin/Tage
heute ist 0
heute ist 1
heute ist 7
heute ist 42
sep@pc305-3:~/fh/c/student>
```

4.4.2 Fallunterscheidung für Aufzählungen

Eine schöne Eigenschaft von Aufzählungen ist, dass sich für Fallunterscheidungen sehr gut der `switch`-Befehl nutzen läßt. Hierzu betrachte man folgendes Beispiel:

Tage1.c

```
1  #include <stdio.h>
2  enum Tag
3    {Montag,Dienstag,Mittwoch,Donnerstag,Freitag,Sonnabend,Sonntag};
4
5  int istWerktag(enum Tag t){
6    switch (t){
7      case Sonnabend:
8      case Sonntag: return 0;
9      default:  return 1;
10   }
```

```
11 }
12
13 int main(){
14   printf("heute ist %i\n",Montag);
15   if (istWerktag(Montag)){
16     printf("und das ist ein Werktag\n");
17   }
18   if (!istWerktag(Sonntag)){
19     printf("Sonntag ist kein Werktag\n");
20   }
21   return 0;
22 }
```

4.4.3 Einwortnamen für Aufzählungen

Die Zweiwortnamen eines Typen sind unpraktisch. Ebenso wie für Strukturen kann man auch für Aufzählungen mit dem Trick über das `typedef`-Konstrukt bei der Deklaration einer Aufzählung gleich den Einworttypnamen für die Aufzählung einführen.

Obiges Programm wird damit etwas kompakter zu:

Tage2.c

```
1  #include <stdio.h>
2  typedef enum
3    {Montag,Dienstag,Mittwoch,Donnerstag,Freitag,Sonnabend,Sonntag}
4  Tag;
5  int istWerktag(Tag t){
6    switch (t){
7      case Sonnabend:
8      case Sonntag: return 0;
9      default:  return 1;
10   }
11 }
12 int main(){
13   printf("heute ist %i\n",Montag);
14   if (istWerktag(Montag)){
15     printf("und das ist ein Werktag\n");
16   }
17   return 0;
18 }
```

4.4.4 Bool'sche Werte als Aufzählung

Wir haben schon mehrfach über einen Typen gesprochen, der sich geradezu dazu eignet, als Aufzählung dargestellt zu werden. Es ist der Typ für Wahrheitswerte, von dem es genau zwei Werte gibt. Wir können, da es ja keinen Typ `bool` in C gibt, diesen in einer winzigen Bibliothek als Aufzählung definieren:

Bool.h

```
#ifndef Bool__H_
#define BOOL__H_
typedef enum {false,true} bool;

void printBool(bool b);
#endif

```

Die Implementierung der Prozedur `printBool` ist entsprechend einfach:

Bool.c

```
#include "Bool.h"
#include <stdio.h>
void printBool(bool b){
  if (b) {printf("true\n");
  }else  {printf("false\n");}
}
```

Inkludieren wir diese kleine Bibliothek so können wir endlich auch in C mit den Wrten `true` und `false` arbeiten.

TestBool.c

```
#include <stdio.h>
#include "Bool.h"
int main(){
  bool b = 1>2;
  printBool(b);
  b=b||true;
  printBool(b);
  b=!b;
  printBool(b);
  return 0;
}
```

4.5 Funktionen als Daten

Wir befinden uns im Kapitel über Daten. Trotzdem sollten wir uns noch einmal den Funktionen widmen. C hat eine wunderbare Eigenschaft, dass auch Funktionen als Daten betrachtet werden können. Das ist auch gar nicht einmal so verwunderlich, denn auch Funktionen liegen ja irgendwo im Speicher unseres Computers. Damit sind auch Funktionen als Daten gespeichert. In C gibt es nun die Möglichkeit, Funktionen als Parameter an andere Funktionen zu übergeben. Leider ist die Typnotation für Funktionen dabei etwas kompliziert. Betrachten wir das Konzept an einem Beispiel.

Wir können in C eine Funktion schreiben, die eine andere Funktion als Parameter erhält. Wozu sollte das gut sein. Es ist z.B. vorstellbar, dass wir die übergebene Funktion zweimal auf einen Parameter anwenden wollen. Hierzu können wir die Funktion `twice` definieren. Sie soll als

ersten Parameter eine Funktion `f` bekommen und als zweiten Parameter eine Zahl `x`. Das Ergebnis soll so berechnet werden, dass zunächst die Funktion `f` auf die Zahl `x` angewendet wird. Anschließend wird nocheinmal die Funktion `f` auf das Ergebnis der ersten Berechnung angewendet. Mathematisch läßt sich die Funktion `twice` durch folgende Gleichung spezifizieren:

$$\text{twice}(f, x) = f(f(x))$$

Tatsächlich läßt sich obige Spezifikation ziemlich genau eins-zu-eins in C umsetzen. Hierzu ist lediglich zu wissen, wie der Typ des Parameters eines Funktionstyps zu deklarieren ist:

Soll ein Paramter eine Funktion darstellen, so ist vor dem Parameternamen der Rückgabetyp zu schreiben, danach in runde Klammern eingeschlossen die Parametertypen. Damit ist z.B. ein Parameter, der die Übergabe einer Funktion erwartet, die einen `int` Parameter erwartet und auch ein `int` als Rückgabe hat, zu deklarieren als `int f(int)`. Der Parametername ist hierbei `f`. Wie man sieht, schreibt man einfach die Signatur der erwarteten Funktion, die übergeben werden soll.

So läßt sich die Funktion `twice` direkt ausdrücken als:

Twice.c

```
1  #include <stdio.h>
2  int twice(int f(int),int x){
3    return f(f(x));
4  }
```

Um diese Auszutesten, brauchen wir erst einmal eine Funktion, die wir für den Parameter `f` übergeben können. Hierzu sei einfach die Funktion, die den Parameter quadriert definiert:

Twice.c

```
5  int quadrat(int x){return x*x;}
```

Wendet man diese Funktion zweimal hintereinander an, so wird hoch vier gerechnet. Das läßt sich jetzt direkt mit der Funktion `twice` ausdrücken:

Twice.c

```
6  int hoch4(int x){return twice(quadrat,x);}
```

Und nun ist endlich Zeit, das alles einmal in einer Hauptfunktion auszuprobieren:

Twice.c

```
7  int add5(int x){return x+5;}
8
9  int main(){
10   printf("twice(add5,32): %i\n",twice(add5,32));
11   printf("hoch4(2): %i\n",hoch4(2));
12   return 0;
13 }
```

Tatsächlich kompiliert das Programm nicht nur, sondern errechnet auch die erwarteten Ausgaben.

```
sep@pc305-3:~/fh/c/student> bin/Twice
twice(add5,32): 42
hoch4(2): 16
sep@pc305-3:~/fh/c/student>
```

Alle Arten von Funktionen können als Parameter übergeben werden. So auch Prozeduren und Funktionen ohne Parameter. Wollen wir z.B. ausdrücken, dass etwas endlich oft wiederholt werden soll, so läßt sich hierzu eine Funktion `repeat` schreiben:

Repeat.c

```
1 #include <stdio.h>
2
3 void repeat(int n, void f()){
4   if (n>0) {
5     f();
6     repeat(n-1,f);
7   }
8 }
```

Nun läßt sich über den Aufruf von `repeat` mehrfach dieselbe Prozedur aufrufen.

Repeat.c

```
1 void printHallo(){
2   printf("Hallo\n");
3 }
4
5 int main(){
6   repeat(10,printHallo);
7   return 0;
8 }
```

Was noch ein wenig esoterisch anmuten mag, Funktionen als Parameter zu übergeben, ist in der Programmierung graphischer Benutzeroberflächen in C Gang und Gebe. Dabei werden graphischen Komponenten, wie z.B. Knöpfen auf der Oberfläche Funktionen übergeben, die diese Komponenten in bestimmten Situationen ausführen sollen (z.B. wenn der Knopf gedrückt wird). Solche Funktionen in der Programmierung graphischer Benutzeroberflächen in C werden als *callback function* bezeichnet.

Allgemein werden Funktion, die wiederum Funktionen als Parameter haben, als Funktionen höherer Ordnung bezeichnet. Die Funktionen `twice` und `repeat` in den obigen Beispielen sind Funktionen höherer Ordnung.

4.5.1 typedef und Funktionstypen

Genauso wie für Strukturen und für Aufzählungen läßt sich das `typedef`-Konstrukt nutzen, um einfachere oder sprechendere Namen für Funktionstypen einzuführen. So läßt sich das letzte Beispiel umformulieren, indem Prozeduren ohne Parameter auch als solche bezeichnet werden:

Repeat2.c

```
1 #include <stdio.h>
2 typedef void prozedur();
```

```
3
4  void repeat(int n,prozedur f){
5    if (n>0) {
6      f();
7      repeat(n-1,f);
8    }
9  }
10
11 void printHallo(){
12   printf("Hallo\n");
13 }
14
15 int main(){
16   repeat(10,printHallo);
17   return 0;
18 }
```

Aufgabe 33 Nehmen Sie Ihre Implementierung aus der letzten Aufgabe und schreiben Sie jetzt eine Funktion höherer Ordnung:
`Complex mapComplex(Complex c,double f(double x))`.
Sie soll die komplexe Zahl zurückgeben, die man erhält, wenn man die übergebene Funktion sowohl auf Real- als auch auf den Imaginärteil anwendet. Testen Sie auch die Funktion an einigen Beispielen.

4.6 Dynamische Daten

4.6.1 Adresstypen

Alle Daten stehen irgendwo im Hauptspeicher. Nicht nur Daten sondern, wie wir gesehen haben, auch Funktionen, weshalb wir sie auch als Daten benutzen können. Da der Speicher unseres Komputers in einzelne Speicherzellen eingeteilt ist, und diese in Adressen durchnummeriert sind, liegt jedes Datenobjekt im Speicher in einer Speicherzelle mit einer bestimmten Adresse.

Adressoperator

C ermöglicht es über den `&`-Operator auf eben diese Adresse zuzugreifen. Die Adressen sind dabei eigentlich nur Zahlen, die wir ausgeben können.

GetAddress.c

```
1  #include <stdio.h>
2  int f(int x){return 2*x;}
3
4  typedef struct {int x;int y;} Point;
5
6  int main(){
7    int x = 42;
8    int y = 17;
```

```
9   Point p ={x,y};
10  double d=0.0;
11  printf("&x: %u\n",&x);
12  printf("&y: %u\n",&y);
13  printf("&p: %u\n",&p);
14  printf("&d: %u\n",&d);
15  printf("&f: %u\n",&f);
16  return 0;
17 }
```

Starten wir das Programm, so gibt uns C ein wenig von seinem Innenleben bekannt. Für die einzelnen Variablen wird die Adresse ausgegeben, an deren Stelle im Speicher die einzelnen Variablen stehen.

```
sep@pc305-3:~/fh/c/student> bin/GetAddress
&x: 3221219380
&y: 3221219376
&p: 3221219368
&d: 3221219360
&f: 134513548
sep@pc305-3:~/fh/c/student>
```

Wie man sieht, gibt es für fast alle Daten Adressen: für einfache Typen wie `int` und `double`, für Strukturen aber auch für Funktionen. Wie man der Ausgabe entnehmen kann liegen die lokalen Variablen recht nah beieinander im Hauptspeicher. Die Funktion hingegen an einer komplett anderen Stelle. Wo im Speicher die einzelnen Daten gespeichert werden, darauf haben wir allerdings keinen Einfluss, das bestimmt der Compiler automatisch.

Adressen gibt es allerdings nur für Daten, die in Variablen gespeichert sind. Konstanten und beliebige Ausdrücke sind davon ausgeschlossen. Versucht man es trotzdem, die Adresse zu erfragen, so kommt es zu einem Übersetzungsfehler:

GetAddressError.c

```
1 #include <stdio.h>
2 int main(){
3   printf("&42: %u\n",&42);
4   printf("&(17+4): %u\n",&(17+4));
5 }
```

Der Compiler gibt folgende Fehlermeldung:

```
sep@pc305-3:~/fh/c/student> gcc src/GetAddressError.c
src/GetAddressError.c: In function ‘main’:
src/GetAddressError.c:3: error: invalid lvalue in unary ‘&’
src/GetAddressError.c:4: error: invalid lvalue in unary ‘&’
sep@pc305-3:~/fh/c/student>
```

Adresstype

Mit dem `&`-Operator läßt sich für jede Variable die Speicheradresse geben. Doch was für einen Typ hat dieser Operator als Rückgabe? Im oberen Testprogramm haben wir die Rückgabe

einfach als Zahl interpretiert zurückgegeben. Wer das Programm aber mit allen Warnungen übersetzt, ahnt bereits, dass da mehr hinter steckt. Tatsächlich gibt es zu jedem Typ in C einen passenden Adresstyp, auch als Zeigertyp oder Pointertyp bezeichnet. Er wird mit einem Sternsymbol nach dem Typnamen notiert. So gibt es also zum Typ `int` den Zeigertyp `int*`. Der `&`-Operator liefert zu einer Variablen vom Typ `x` die Adresse des Typs `x*`, also für Variablen des Typs `int` eine Adresse vom Typ `int*`.

FirstPointer.c

```
1  #include <stdio.h>
2  typedef struct {int x;int y;} Point;
3
4  int main(){
5    int x = 42;
6    int* xp = &x;
7    int** xpp = &xp;
8    Point p = {17,4};
9    Point* pp = &p;
10   return 0;
11 }
```

Wie man an der Variablen `xpp` sieht kann man das Spielchen tatsächlich mehrfach spielen. Man kann auch von einer Adressvariablen sich wiederum die Adresse geben lassen.

Betrachten wir das Programm noch einmal im Detail. In Zeile 5 wird eine Variable deklariert. Hierzu bedarf es einem Speicherplatz im Hauptspeicher um eine Zahl zu speichern. Dieser Speicherplatz hat eine Adresse. In Zeile 6 wird mit `&x` die Adresse des Speicherplatz für die Variable `x` erfragt. Diese Adresse hat den Typ `int*`. Auch sie soll in einer Variablen gespeichert werden. Deshalb wird die Variable `xp` vom Typ `int*` deklariert. Auch die Variable `xp` wird im Speicher in einer Speicherzelle mit einer eigenen Adresse gespeichert. Diese Adresse läßt sich mit `&xp` erfragen. Sie ist vom Typ `int**` und wird in Zeile 7 in der Variablen `xpp` gespeichert.

Stern-Operator

Mit dem `&`-Operator läßt sich die Speicheradresse einer Variablen in Form eines Zeigertyps erfragen. Als inversen Operator gibt es in C den Sternoperator `*`. Wird er auf einen Zeigertyp angewendet, dann werden die Daten angesprochen, die an der entsprechenden Adresse gespeichert sind. Er kehrt quasi die Operation des `&`-Operators wieder um. Während der `&`-Operator für ein `int` eine Adresse auf ein `int` berechnet, also ein `int*`, berechnet der *-Operator für ein `int*` wieder ein `int`.

Jetzt können wir das obige Programm erweitern, und mit Hilfe des Sternoperators für die Zeigervariablen wieder die Werte erfragen, die in den Speicherzellen gespeichert sind, auf den die Zeigervariablen verweisen.

StarOperator.c

```
1  #include <stdio.h>
2  typedef struct {int x;int y;} Point;
3
4  int main(){
5    int x = 42;
6    int* xp = &x;
```

```
7   int y = *xp;
8   printf("y = %i\n",y);
9
10  int** xpp = &xp;
11  int z = **xpp;
12  printf("z = %i\n",z);
13  printf("**xpp = %i\n",**xpp);
14
15  *xp = 17;
16  printf("x = %i\n",x);
17
18  Point p = {17,4};
19  Point* pp = &p;
20  printf("(*pp).x = %i\n",(*pp).x);
21  return 0;
22 }
```

```
sep@pc305-3:~/fh/c/student> bin/StarOperator
z = 42
**xpp = 42
x = 17
(*pp).x = 17
sep@pc305-3:~/fh/c/student>
```

Pfeiloperator

Beim Zugriff auf die Felder einer Struktur, die über einen Zeiger zugegriffen wird, ist zunächst mit dem Sternoperator die Referenz aufzulösen, um dann mit dem Punktoperator auf das Feld der Struktur zuzugreifen. Das gibt dann Ausdrücke der Form `(*pp).x`, wie bereits im letzten Beispiel gesehen. C bietet hierfür eine speziellen Operator an, der dieses etwas kürzer notieren läßt, der Pfeiloperator `->`. `(*pp).x` kann mit ihm als `pp->x` ausgedrückt werden.

ArrowOperator.c

```
1  #include <stdio.h>
2  typedef struct {int x;int y;} Point;
3
4  int main(){
5    Point p = {17,4};
6    Point* pp = &p;
7    printf("(*pp).x = %i\n",(*pp).x);
8    printf("pp->x = %i\n",pp->x);
9    return 0;
10 }
```

Referenzübergabe

Was haben wir mit den Zeigern gewonnen (außer dass es etwas komplizierter wird Programme zu verstehen)? Man erinnere sich daran, dass in C die Parameter an eine Funktion immer per Wert übergeben werden. Das bedeutet, dass die Parameterwerte kopiert werden. Man erinnere

sich dazu an das Beispielprogramm `NoRefparam`. Mit Zeigertypen gibt es nun eine Möglichkeit, nicht den Wert, sondern die Speicheradresse einer Variablen an eine Funktion zu übergeben. Durch minimale Änderungen am Programm `NoRefparam` erhalten wir ein Programm, in dem die Funktion tatsächlich die übergebene Variable verändern kann.

RefParam.c

```
1  #include <stdio.h>
2  int f1(int* x){
3    *x = 2* (*x);
4    return *x;
5  }
6  int main(){
7    int y = 42;
8    int z = f1(&y);
9    printf("y = %i\n",y);
10   printf("z = %i\n",z);
11   return 0;
12 }
```

Wie man an der Ausgabe sehen kann, ändert der Aufruf von `f1` jetzt tatsächlich den Wert der Variablen `y`.

```
sep@pc305-3:~/fh/c> ./student/bin/RefParam
y = 84
z = 84
sep@pc305-3:~/fh/c>
```

Parameter per Zeiger zu übergeben statt per Wert kann insbesondere sinnvoll sein, wenn es sich bei den Parametern um Strukturen handelt. Strukturobjekte können mitunter sehr groß sein und viel Speicherplatz beanspruchen. Auch will man gerade recht oft Strukturobjekte durch Prozeduraufrufe verändern.

Betrachten wir hierzu ein weiteres Mal die Struktur `Punkt`. Wir haben bereits festgestellt, dass bei einer Wertübergabe die Funktion `verschiebe` das ursprünglichen Punktobjekt gar nicht verändert. Jetzt per Übergabe des Punktobjets als Zeiger tritt der gewünschte Effekt ein:

Punkt4.c

```
1  #include <stdio.h>
2  typedef struct {
3    int x;
4    int y;
5  } Punkt;
6
7  void printPunkt(Punkt p){
8    printf("(%i,%i)\n",p.x,p.y);
9  }
10
11 void verschiebe(Punkt* p){
12   p->x=p->x+1;
13   p->y=p->y+2;
14 }
```

```
15
16 int main(){
17   Punkt p={17,4};
18   printPunkt(p);
19   verschiebe(&p);
20   printPunkt(p);
21   return 0;
22 }
```

Zeiger auf lokale Variablen

Bisher vertrauen wir auf die automatische Speicherverwaltung lokaler Variablen in C. Für eine lokale Variable einer Funktion ebenso wie für einen Funktionsparameter, wird beim Aufruf einer Funktion Platz im Speicher angelegt. An dieser Stelle werden die Daten der Variablen gespeichert. Sobald die Funktion fertig ausgewertet ist und ihr Ergebnis berechnet hat, wird dieser Speicherplatz nicht mehr benötigt. Die Speicherstelle wird wieder freigegeben, um sie später für eventuelle andere Variablen zu benutzen.

Da wir uns explizit mit dem `&`-Operator die Adresse einer Speicherzelle geben lassen können, so können wir eine solche Adresse auch als Funktionsergebnis zurückgeben. Es läßt sich folgende Funktion definieren:

DeadVariable.c

```
1 #include <stdio.h>
2
3 int* f(int y){
4   int x= y;
5   return &x;
6 }
```

Bei einem Aufruf dieser Funktion erhalten wir einen Zeiger auf eine Speicherzelle, in der während des Aufrufs eine lokale Variable gespeichert war. Auf die Variable selbst läßt sich nicht mehr zugreifen. Sie existiert quasi nicht mehr. Wir haben also einen Zeiger in einen Speicherbereich, der einmal für eine lokale Variable genutzt wurde. Diese Variable gibt es aber gar nicht mehr.

Der Compiler macht uns auch tatsächlich darauf aufmerksam, dass hier etwas merkwürdig ist:

```
sep@pc305-3:~/fh/c> gcc DeadVariable.c
DeadVariable.c: In function ‘f’:
DeadVariable.c:5: warning: function returns address of local variable
sep@pc305-3:~/fh/c>
```

Da es sich aber nur um eine Warnung handelt, vielleicht wissen wir ja was wir tun, können wir das Programm übersetzen. Wir vervollständigen es um eine zweite Funktion und um eine Hauptfunktion.

DeadVariable.c

```
7 void f2(int y){
8   int z= y;
9 }
10
```

```
11 int main(){
12   int* xp = f(42);
13   f2(17);
14   printf("%i\n",*xp);
15   return 0;
16 }
```

Das Programm kann zu folgender Ausgabe führen:

```
sep@pc305-3:~/fh/c> ./DeadVariable
17
sep@pc305-3:~/fh/c>
```

Hier läßt sich etwas über die interne Speicherverwaltung unseres Compilers lernen. Wir lassen uns die Speicherzelle der lokalen Variablen `x` aus der Funktion `f` zurückgeben. Anschließend rufen wir die Funktion `f2` auf. Diese hat ihrerseits eine lokale Variable. Nach Aufruf der Funktion `f2` befindet sich ein Wert in der Speicherzelle die einstmals die lokale Variable `x` speicherte, der zuletzt in der lokalen Variablen `z` der Funktion `f2` stand.

Tatsächlich ist es ziemlicher Zufall, was sich am Speicherplatz einer nicht mehr existierenden lokalen Variable befindet.

der NULL-Zeiger

Wir haben uns bemüht Variablen immer gleich bei der Deklaration auch einen initialen Wert zuzuweisen. Wird einer Variablen kein Wert zugewiesen, so hängt es vom Zufall ab, was für einen Wert man bekommt, wenn auf eine solche noch nicht initialisierte Variable zugegriffen wird. Dasselbe gilt auch für Zeigervariablen. Es gibt aber Situationen, in denen man eine Zeigervariable deklariert, ohne dass schon der zuzuweisende Adresswert vorhanden ist. Oder umgekehrt: für eine Zeigervariable ist im Laufe des Programms keine gültige Adresse mehr vorhanden. Um in einer Zeigervariablen nicht einen zufälligen oder ungültigen Adresswert speichern zu müssen, gibt es in C die Möglichkeit eines ausgezeichneten null-Zeigers. Er wird durch das Wort `NULL` bezeichnet. Man kann mit der Gleichheit testen, ob ein Zeiger mit `NULL` belegt ist.

NullVar.c

```
17 #include <stdlib.h>
18 #include <stdio.h>
19 int* f(int* y){
20   static int* x=NULL;
21   if (NULL!=x) x=y;
22   else *y=*x;
23   return x;
24 }
25
26 int main(){
27   int x= 42;
28   printf("*f(&x)%i\n",*f(&x));
29   x=17;
30   printf("*f(&x)%i\n",*f(&x));
31   printf("x%i\n",x);
```

```
32   return 0;
33 }
```

Um den `NULL`-Zeiger verwenden zu können, ist es notwendig die Bibliothek `stdlib.h` zu inkludieren.

4.6.2 der void-Zeiger

Wir haben gesehen, dass es für jeden Typ, einen entsprechenden Zeigertyp gibt. So gibt es zu `int` den Zeigertyp `int*` und für eine Struktur `struct Punkt` den Zeigertyp `struct Punkt*`. Technisch gesehen sind alle diese Zeigertypen eine Zahl, die eine Adresse in unserem Speicherbereich bezeichnet. Also sind eigentlich alle Zeigertypen Daten von derselben Art. Die Programmiersprache C bietet einen besonderen allgemeinen Zeigertyp an, der ausdrückt, dass es sich um irgendeinen Zeiger handelt, der aber nicht angibt, was in der Speicheradresse gespeichert ist, auf die der Zeiger verweist. Dieser allgemeine Zeigertyp wird mit `void*` bezeichnet. Das Wort `void` sollte hier so gelesen werden, dass nicht bekannt ist, was für Daten an der Speicherstelle stehen, auf die verwiesen wird.

Zunächst definieren wir als Beispiel eine kleine Struktur:

VoidPointer.c

```
1  #include <stdio.h>
2
3  typedef struct {
4    int x;
5    int y;
6  } Punkt;
```

Wir wollen einmal einen `void`-Zeiger benutzen:

VoidPointer.c

```
7  int main(){
8    void* object;
```

In diesem `void`-Zeiger läßt sich jetzt z.B. die Adresse einer `int`-Vatiablen speichern.

VoidPointer.c

```
9    int x = 17;
10   object = &x;
```

Soll die Adresse über den `void`-Zeiger angesprochen werden und mit der darin gespeicherte Zahl gerechnet werden, so müssen wir explizit den Zeiger zu einem Zeiger auf eine `int`-Zahl umwandeln. Hierzu ist die Notation in runden Klammern zu benutzen, wie wir sie schon vom Rechnen mit primitiven Zahlen gewohnt sind.

VoidPointer.c

```
11   *(int*)object += 4;
12   *(int*)object *= 2;
13   printf("x = %i\n",x);
```

Wir können aber auf die gleiche Weise versuchen, in der `void`-Zeigervariablen ein Objekt unserer Punktstruktur zu speichern. Um wieder auf das Objekt zugreifen zu können, ist dann der Zeiger wieder entsprechend umzuwandeln.

VoidPointer.c

```
14    Punkt p = {2,6};
15    object = &p;
16
17    printf("p.x: %i\n",p.x);
18    printf("((Punkt*)object)->x: %i\n",((Punkt*)object)->x);
19
20    return 0;
21  }
```

Im obigen Beispiel ist mit dem Namen `object` schon angedeutet, wofür man den `void`-Zeiger verwenden kann: für die Möglichkeit, Zeiger auf Daten belieber Art zu speiechern. In der objektorientierten Programmierung wird dieses in dem Konzept eines allgemeinen Objektes münden. Am Ende dieses Kapitels werden wir ein Beispiel sehen, in dem wir `void`-Pointer sinnvoll nutzen.

4.6.3 Rechnen mit Zeigern

Zeiger sind technisch nur die Adressen bestimmter Speicherzellen, also nichts weiteres als Zahlen. Und mit Zahlen können wir rechnen. So können tatsächlich arithmetische Operationen auf Zeigern durchgeführt werden.

Beispiel 4.6.1 *Wir erzeugen mehrere Zeiger auf ganze Zahlen und betrachten die Nachbarschaft eines dieser Zeiger.*

ZeigerArith.c

```
1   #include <stdio.h>
2
3   int main(){
4     int i = 5;
5     int j = 42;
6     int *pI = &i;
7     int *pJ = &j;
8
9     int *pK;
10    printf
11     ("i:%i,pI:%u,pI+1:%u,*(pI+1):%i,pI-1:%u,*(pI-1):%i,pJ:%u,*pJ:%i\n"
12      ,i   ,pI   ,pI+1   ,*(pI+1)   ,pI-1   ,*(pI-1)   ,pJ   ,*pJ );
13    *pK = 12;
14    printf("pK: %i, *pK: %i\n", pK,*pK);
15    return 0;
16  }
```

Dieses Programm erzeugt z.B. folgende interessante Ausgabe:

```
sep@pc305-3:~/fh/c/student> bin/ZeigerArith
i:5,pI:3221219380,pI+1:-1073747912,*(pI+1):-1073747816,pI-1:3221219376,*(pI-1):42,pJ:3221219376,*pJ:42
pK: 1075174352, *pK: 12
sep@pc305-3:~/fh/c/student>
```

Das Ergebnis einer Zeigerarithmetik hängt sehr davon ab, wie die einzelnen Variablen im Speicher abgelegt werden. In der Regel wird man die Finger von ihr lassen und nur ganz bewußt auf sie zurückgreifen, wenn man genau weiß, was man tut, um eventuell Optimierungen durchführen zu können.

Erstaunlicher Weise werden wir aber ein C-Konstrukt kennenlernen, das eines der gebräuchlichsten Konzepte in C überhaupt ist, und tatsächlich auf der Zeigerarithmetik basiert.

4.6.4 Speicherverwaltung

Dieses Kapitel trägt den Titel dynamische Daten. Hiervon haben wir vorerst aber noch nicht wirklich etwas gesehen. Bisher haben wir nur geschaut, in welchen Speicherzellen die Daten unserer Variablen gespeichert sind. Damit ist in einem Programm relativ statisch durch die Anzahl der Variablen festgelegt, wieviel Daten das Programm speichern kann. Normaler Weise wünscht man sich aber, dass ein Programm während des Programmdurchlaufs dynamisch neuen Speicher nach Bedarf für weitere Daten anlegen kann. Hierzu bietet C die Möglichkeit an über, Speicherallokationsfunktionen neuen Speicher zu reservieren.

Allocation

In der C-Standardbibliothek gibt es zwei ein Funktionen, mit der Speicher beliebiger Größe vom Compiler angefordert werden kann.

malloc Die Funktion `malloc` hat einen Parameter. In diesem wird angegeben, wieviel Speicherplatz bereitgestellt werden soll. Diese Angabe ist die Anzahl der Byte. Das Ergebnis ist ein Zeiger auf diesen neu bereitgestellten Speicher. Da die Funktion `malloc` nicht wissen kann, was für Daten einmal in diesem Speicherbereich gespeichert werden sollen, ist ihr Rückgabetyp der allgemeine Zeiger `void*`.

Malloc1.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(){
5    void* vp = malloc(7);
6    int* ip=(int*)vp;
7    *ip=42;
8    printf("%i\n",*ip);
9    return 0;
10 }
```

Im obigen Beispiel haben wir willkürlich sieben Byte Speicher reservieren lassen, um sie dann zur Speicherung von einer `int`-Zahl zu benutzen. Man sollte natürlich am Besten immer genau soviel Speicher anfordern, wie für die geplanten Zwecke gebraucht wird. Wir kennen bereits die

Funktion `sizeof`, die Angaben darüber macht, wieviel Speicher ein bestimmter Typ benötigt. Mit dieser läßt sich genau der erforderlicher Speicherplatz anfordern:

Malloc2.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(){
5    void* vp = malloc(sizeof(int));
6    int* ip=(int*)vp;
7    *ip=42;
8    printf("%i\n",*ip);
9    return 0;
10 }
```

calloc Es gibt in C eine Variante der Funktion `malloc`: die Funktion `calloc`. Nur, dass es bei der Funktion calloc() nicht einen, sondern zwei Parameter gibt. Im Gegensatz zu `malloc` können wir mit `calloc` noch die Anzahl von Speicherobjekten angeben, die reserviert werden soll. Wird z.B. für 10 Zahlen vom Typ `int` Speicherplatz benötigt, so kann dies mit `calloc` erledigt werden. Auf die mehreren Elemente dieses Speicherbereichs kann dann mit der Zeigerarithmetik zugegriffen werden:

Calloc.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(){
5    int* xs = (int*)calloc(10,sizeof(int));
6    int i=0;
7    for(i=0;i<10;i++)  *(xs+i) = 2*i;
8    for(i=0;i<10;i++)  printf("%i, ",*(xs+i));
9    printf("\n");
10   return 0;
11 }
```

Wir haben somit ein kleines Programm geschrieben, in dem wir Speicher für 10 Zahlen angefordert haben und mit diesen 10 Speicherplätzen auch gearbeitet haben:

```
sep@pc305-3:~/fh/c/student> bin/Calloc
0, 2, 4, 6, 8, 10, 12, 14, 16, 18,
sep@pc305-3:~/fh/c/student>
```

Zusätzlich werden mit der Funktion `calloc` alle Werte des angeforderten Speichers automatisch mit binären 0-Werten initialisiert. Bei `malloc` hat der reservierte Speicherplatz zunächst einen undefinierten Wert. Braucht man die zusätzliche Initialisierung des Speichers nicht, so kann natürlich auch mit `malloc` Speicher für mehrere Elemente angefordert werden. Der Aufruf
`calloc(10,sizeof(int));`
ist hierzu zu Ersetzen durch:
`malloc(10*sizeof(int));`

realloc to be done

Speicher für ganze Datenreihungen

Wir haben oben schon gesehen, wie sich Speicher für ganze Reihen von Daten eines Typs bequem allokieren läßt. Über die Zeigerarithmetik läßt sich dann auf die einzelnen Elemente in einer solchen Reihung zugreifen.

Zeichenketten Bisher hatten wir noch keine Möglichkeit mit Zeichenketten, also Wörtern und Sätzen zu arbeiten. Einzelne Buchstaben ließen sich durch Daten des Typs `char` adequat[4] darstellen. Ein Wort oder längere Texte können nun als ein zusammen allokierten Speicherbereich von `char`-Werten aufgefasst werden. Damit kann ein Zeiger auf diesen Speicherbereich als Zeiger auf einen ganzen Text verstanden werden. So können wir Speicher anfordern und darin Wörter speichern:

Word1.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(){
5    char* hallo = malloc(5*sizeof(char));
6    *(hallo+0) = 'H';
7    *(hallo+1) = 'A';
8    *(hallo+2) = 'L';
9    *(hallo+3) = 'L';
10   *(hallo+4) = 'O';
11
12   printf("%c\n",*(hallo+3));
13   return 0;
14 }
```

Wir haben ein Problem: wir können der Variablen `hallo` vom Typ `char*` im obigen Beispiel nich ansehen, für wieviel Buchstaben dort Platz ist. Diese Information müssten wir seperat in einer Variablen speichern. Also die eigentliche Wortlänge. Mit dieser Information läßt sich dann auch ein Wort komplett ausdrucken:

Word2.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void printWord(unsigned int l, char* word){
5    unsigned int i;
6    for (i=0;i<l;i++)
7      printf("%c",*(word+i));
8
9    printf("\n");
10 }
11
```

[4] Nunja, zumindest wenn man sich auf die 26 Buchstaben des lateinischen Alpabets beschränkt.

```
12 int main(){
13   unsigned int laenge = 5;
14
15   char* hallo = malloc(laenge*sizeof(char));
16   *(hallo+0) = 'H';
17   *(hallo+1) = 'A';
18   *(hallo+2) = 'L';
19   *(hallo+3) = 'L';
20   *(hallo+4) = 'O';
21
22   printWord(laenge,hallo);
23
24   return 0;
25 }
```

Das ist etwas umständlich, wenn man immer die Länge als Zweitinformation mit sich herumschleppen muß. Deshalb gibt es einen Standardtrick in C, wenn man mit Reihungen von Zeichen arbeitet. Als letztes Zeichen schreibt man noch den Nullbuchstaben, der als `'\0'` notiert wird. Dann kann man auf eine vorgegebene Länge verzichten und bei der Ausgabe abbrechen, wenn das Nullzeichen erreicht wurde.

Word3.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void printWord( char* word){
5    int i;
6    for (i=0;*(word+i)!='\0';i++)
7      printf("%c",*(word+i));
8
9    printf("\n");
10 }
11
12 int main(){
13   char* hallo = malloc(6*sizeof(char));
14   *(hallo+0) = 'H';
15   *(hallo+1) = 'A';
16   *(hallo+2) = 'L';
17   *(hallo+3) = 'L';
18   *(hallo+4) = 'O';
19   *(hallo+5) = '\0';
20   printWord(hallo);
21
22   return 0;
23 }
```

Die oben dargestellte Form Texte zu verarbeiten ist der Standardweg in C. Dieses ist in die Sprache eingebaut. Wenn ein Text in doppelten Anführungszeichen steht, wie wir es ja schon oft in den Funktionsaufrufen von `printf` gesehen haben, dann wird der Text auf eben diese

Weise in einen über den Typ `char*` beschriebenen Speicherbereich wie oben von Hand durchgeführt gespeichert. Somit läßt sich die Erzeugung des obigen Textes im Speicher viel einfacher schreiben:

Word4.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void printWord( char* word){
5    int i;
6    for (i=0;*(word+i)!='\0';i++)
7      printf("%c",*(word+i));
8
9    printf("\n");
10 }
11
12 int main(){
13   char* hallo = "HALLO";
14   printWord(hallo);
15
16   return 0;
17 }
```

Ein weiterer Luxus ist, dass die Funktion `printf` auch darauf vorbereitet ist, derartige Zeichenketten auszugeben, so dass wir auf ein eigenes `printWord` verzichten können. Hierzu ist der Funktion `printf` als Formatierungsanweisung `%s` mitzugeben:

Word5.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(){
5    char* hallo = "HALLO";
6    printf("%s\n",hallo);
7
8    return 0;
9  }
```

Aufgabe 34 Schreiben Sie eine Funktion `int length(char* text)`, die angibt, wieviel Zeichen in dem Speicherbereich von `text` gespeichert sind.

Lösung

B6A1.c

```
1  #include <stdio.h>
2  int length(char* text){
3    int i;
4    for (i=0;*(text+i)!='\0';i++){}
5
6    return i;
```

```
7  }
8
9  int main(){
10   printf("%i\n",length(""));
11   printf("%i\n",length("A"));
12   printf("%i\n",length("Hallo"));
13   printf("%i\n",length("ottos mops kotzt"));
14   return 0;
15 }
```

Aufgabe 35 Schreiben Sie eine Funktion `bool isPallindrom(char* text)`, die testet, ob der übergebene Text vorwärts und rückwärts gelesen identisch ist.

Lösung

B6A2.c

```
1  #include <stdio.h>
2  #include "Bool.h"
3  int length(char* text){
4    int i;
5    for (i=0;*(text+i)!='\0';i++){}
6    return i;
7  }
8
9  bool isPallindrom(char* text){
10   int l = length(text);
11   int i;
12   for (i=0;i<l/2;i++)
13     if ((*(text+i)) != (*(text+l-1-i))) return false;
14   return true;
15 }
16 int main(){
17   printBool(isPallindrom("otto"));
18   printBool(isPallindrom("hans"));
19   printBool(isPallindrom("koelbleok"));
20   printBool(isPallindrom("regallager"));
21
22   return 0;
23 }
```

Aufgabe 36 Schreiben Sie eine Funktion `int leseAlsZahl(char* text)`, die Zeichenkette als Zahl interpretiert. Für die Zeichenkette `"42"` soll also die Zahl `42` als Ergebnis zurückgegeben werden:

Lösung

B6A3.c

```
1  #include <stdio.h>
2  int leseAlsZahl(char* text){
```

```
3   int result=0;
4   int i;
5   for (i=0;*(text+i)!='\0';i++)
6     result=result*10+(*(text+i)-'0');
7   return result;
8 }
9 int main(){
10  printf("%i\n",leseAlsZahl("42"));
11  printf("%i\n",leseAlsZahl("007"));
12  printf("%i\n",leseAlsZahl("a"));
13  printf("%i\n",leseAlsZahl("12"));
14  return 0;
15 }
```

Aufgabe 37 Schreiben Sie eine Funktion `char* concat(char* text1,char* text2)`, die eine neue Zeichenkette erzeugt, die aus den beiden Zeichenketten der Parameter aneinandergehängt besteht.

Lösung

B6A4.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int length(char* text){
5    int i;
6    for (i=0;*(text+i)!='\0';i++){}
7    return i;
8  }
9  char* concat(char* text1,char* text2){
10   char* result=malloc(sizeof(char)* (length(text1)+length(text2)+1));
11   int i;
12   for (i=0;*(text1+i)!='\0';i++){
13     *(result+i)=*(text1+i);
14   }
15   int j;
16   for (j=0;*(text2+j)!='\0';(i++,j++)){
17     *(result+i)=*(text2+j);
18   }
19   *(result+i)='\0';
20   return result;
21 }
22 int main(){
23   printf("%s\n",concat("otto","reber"));
24   printf("%s\n",concat("007","008"));
25   printf("%s\n",concat ("john","paul"));
26   printf("%s\n",concat ("george","ringo"));
27   return 0;
```

```
28 }
```

Speicherfreigabe

Speicher anzufordern ist ein feine Sache, aber wenn wir auch viel davon in heutugen Rechner haben, so ist der Speicher trotzdem endlich. Wenn wir sorglos immer wieder neuen Speicher anfordern bekommen wir ein Müllproblem. Wie sich ein solches auswirkt, läßt sich leicht ausprobieren. Man starte einmal das folgende Progamm.

NoFree.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main(){
4    int i=0;
5    int* ip;
6    for (;i<10000000;i++){
7      ip=(int*)malloc(sizeof(int));
8      *ip=i;
9      printf("%i ",*ip);
10   }
11   return 0;
12 }
```

Mit dem Betriebssystembefehl `top` läßt sich schön beobachten, wie dieses Programm immer neuen Speicher benötigt und schließlich über 100 Megabyte Speicher reserviert hat (und damit den Rechner schon arg belastet). Man spricht in einem solchen Fall von einem Speicherleck. Eine der großen Schwierigkeiten der Programmiersprache C ist es, dass dynamisch allokierter Speicher explizit wieder frei gegeben werden muß. Hierzu gibt es die Funktion `free`. Sie erwartet einen Zeiger und gibt alles mit diesem Zeiger allokierten Speicherbereich wieder frei zur neuen Benutzung. Rufen wir nun im obigen Programm am Ende des Schleifenrumpfs die Funktion `free` auf, so können wir beobachten, dass das Programm mit konstanten Speicherbelegung läuft.

Free.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main(){
4    int i=0;
5    int* ip;
6    for (;i<10000000;i++){
7      ip=(int*)malloc(sizeof(int));
8      *ip=i;
9      printf("%i ",*ip);
10     free(ip);
11   }
12   return 0;
13 }
```

Schon seit Anfang der 60er Jahre gibt es mit *Lisp* eine Programmiersprache, in der die dynamische Speicherverwaltung automatisch vom Compiler durchgeführt wird. Dieses macht die sogenannte *garbage collection*. Es ist erstaunlich und schwer erklärlich, dass sich eine Programmiersprache ohne eine vollkommen automatische Speicherverwaltung wie C derart durchsetzen und hartnäckig halten konnte. Erst mitte der 90er Jahre setzte sich mit *Java* eine Mainstream Programmiersprache durch, die eine automatische Speicherverwaltung per *garbage collection* hat.

Reflexartig wiederholt für alle unangenehmen Eigenschaften von C wird stets ein Performancevorteil. Allerdings spielt dieser wahrscheinlich für 99% der Anwendungen keine signifikante Rolle.

Warum hat man aber nicht irgendwann eine automatische Speicherverwaltung in C integriert. Tatsächlich gibt es C-Compiler die dieses zumindest teilweise anbieten; allerdings verhindern einige Konzepte von C, insbesondere die Speicherarithmetik, dass eine *garbage collection* allgemein in C eingeführt werden könnte.

Doch genug des C *bashings*. Für uns ist sicherlich C eine wunderbare Sprache, um etwas über den Speicher zu lernen und konfontiert damit zu werden, dass angeforderter Speicher irgendwie auch wieder frei gesetzt werden muss. Eine Erfahrung, die wir in Java nie gemacht hätten.

Ganz allein läßt uns C allerdings auch nicht mit der Speicherverwaltung. Zum einen wird der Speicher lokaler Variablen nach Ablauf einer Funktionsberechnung wieder frei gegeben, zum anderen müssen wir dem Compiler in der Funktion `free` nicht mitteilen, wieviel Speicher denn wieder frei gegeben werden soll. Die C-Laufzeit merkt sich für jede dynamisch angeforderte Speicherzelle, wieviel Speicher angefordert wurde und gibt genau diese Speichergröße dann auch wieder frei.

objektorientiertes Arbeiten mit Strukturzeigern

Es ist Vorteilhaft auch in C schon einen weitgehendst objektorientierten Programmierstil zu kultivieren. Objektarten werden durch Strukturen definiert. Objekte dieser Strukturen sollen nur über einen Zeiger benutzt werden. Wir können eine weitgehendst objektorientiert angelegte Struktur gerne auch schon als *Klasse* bezeichnen.

Hierzu nocheinmal das Beispiel mit den Punkten im zweidimensionaloen Raum:

Punkt5.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4
5  typedef struct {
6    int x;
7    int y;
8  } PunktStrukt;
9
10 typedef PunktStrukt* Punkt;
```

Es ist sinnvoll eine Funktion vorzusehen, die aus gegebenen Werte ein Objekt einer Struktur erzeugt. Hierzu ist der entsprechende Speicher zu allokieren und die einzelnen Felder des Strukturobjekts mit den Werten der Parameter zu übergeben. Eine solche Funktion wird als

Konstruktor bezeichnet. Eine sinnvolle Namensgebung ist den Konstruktornamen mit `new` beginnen zu lassen, gefolgt von dem Namen der Struktur:

Punkt5.c

```
11 Punkt newPunkt(int x,int y){
12   Punkt this = (Punkt)malloc(sizeof(PunktStrukt));
13   this->x = x;
14   this->y = y;
15   return this;
16 }
```

Ebenso sollte man eine Funktion vorsehen, die es erlaubt ein Objekt wieder aus dem Speicher zu entfernen. Man spricht dabei von einen Destruktor. Meistens reicht es aus, hier einfach die Funktion `free` aufzurufen, wir werden aber bald auch ein Beispiel sehen, in dem dies nicht ausreicht. Für die Punkt-Klasse erhalten wir den einfachen Destruktor:

Punkt5.c

```
17 void deletePunkt(Punkt this){
18   free(this);
19 }
```

Nun können wir Punktobjekte erzeugen und löschen. Jetzt wollen wir mit diesen Objekten auch gerne etwas vornehmen. Dazu schreiben wir Funktionen, die als ersten Parameter ein entsprechendes Objekt erhalten. In Hinblick auf objektorientierte Sprachen ist es sinnvoll diesen ersten Parameter dann mit den Namen `this` zu bezeichnen. Funktionen, die als ersten Parameter Objekte einer bestimmten Klasse erhalten, sollen als *Objektmethode* kurz auch nur *Mehtode*dieser Klasse bezeichnet werden.

So können wir eine einfache Methode schreiben, die ein Punktobjekt auf der Kommandozeile ausgibt:

Punkt5.c

```
20 void printPunkt(Punkt this){
21   printf("(%i,%i)\n",this->x,this->y);
22 }
```

Mehtoden können auch Rückgaben haben. Die folgende Mehtode errechnet als den betrag eines Punktes seine Entfernung zum Ursprung. Hierzu wird die Formel von Pythagoras verwendet:

Punkt5.c

```
23 double betrag(Punkt this){
24   return sqrt(this->x*this->x+this->y*this->y);
25 }
```

Methoden können nach dem ersten Parameter noch weitere Parameter enthalten. Die folgende Methode kann ein Punktobjekt um die als weitere Werte angegebenen Distanzen verschieben.

Punkt5.c

```
26 void verschiebe(Punkt this,int deltaX,int deltaY){
27   this->x += deltaX;
28   this->y += deltaY;
29 }
```

Nun können wir tatsächlich schon fast wie in einer objektorientierten Sprache mit der Punkt-Klasse arbeiten:

Punkt5.c

```
30 int main(){
31   Punkt p= newPunkt(17,4);
32   printPunkt(p);
33   printf("betrag: %f\n",betrag(p));
34   verschiebe(p,2,4);
35   printPunkt(p);
36   deletePunkt(p);
37   return 0;
38 }
```

Das Programm führt zu folgender Ausgabe:

```
sep@pc305-3:~/fh/c/student> bin/Punkt5
(17,4)
betrag: 17.464249
(19,8)
sep@pc305-3:~/fh/c/student>
```

4.7 Reihungen (Arrays)

Eines der komplexesten Konstrukte der Programmiersprache C ist gleichzeitig auch das wahrscheinlich am häufigsten benutzte. Es handelt sich dabei um Arrays, auf Deutsch oft als Datenfelder bezeichnet. Innerhalb dieses Skripts werden wir sie mit dem ebenfalls gebräuchlichen Ausdruck *Reihungen* bezeichnen, da wir als Feld auch schon die Attribute von Strukturen bezeichnen. Bei Reihungen geht es darum, mehrere Variablen eines uniformen Datentyps in einer Variablen zu speichern.

Ähnlich wie schon bei Funktionstypen wird der Typ einer Reihung vor und hinter dem Variablennamen bezeichnet. Vor dem Variablennamen steht der Typ, den die einzelnen Elemente der Reihung haben und nach dem Variablennamen steht ein eckiges Klammernpaar, um zu bezeichenen, dass es sich hier im eine Reihungsvariablen handelt. So bezeichnet `int xs []` eine Reihung von ganzen Zahlen. Eine Reihung kann, ebenso wie wir es von Strukturen kennen, durch die Aufzählung der Werte in geschweiften Klammern intialisiert werden.

Zeit unseren ersten Array zu deklarieren. Statt vier einzelner Variablen (noch einmal als Beispiel drüber) wird eine einzige Variable mit vier Werten deklariert:

FirstArray.c

```
1 #include <stdio.h>
2
3 int main(){
4   int x0 = 1;
5   int x1 = 2;
6   int x2 = 3;
7   int x3 = 4;
8   int x [] = {1,2,3,4};
```

Normale Variablen können wir direkt benutzen, um an ihre Werte zu kommen:

FirstArray.c
```
1   printf("x0  =%i, x1  =%i, x2  =%i, x3 =%i\n",x0,x1,x2,x3);
```

In Arrayvariablen verstecken sich mehrere einzelne Werte. Um auf diese einzeln zugreifen zu können wird ein Index benutzt. Der Index ist in eckigen Klammern der Variablen anzuhängen. Das erste Element wird über den Index 0 angesprochen. Die vier Werte unserer ersten Reihung sind also wie folgt anzusprechen:

FirstArray.c
```
1   printf("x[0]=%i, x[1]=%i, x[2]=%i, x[3]=%i\n",x[0],x[1],x[2],x[3]);
2   return 0;
3 }
```

Bei der Deklaration einer Reihungsvariablen muß die Anzahl der Elemente bekannt sein. Im ersten Beispiel wurde die Anzahl aus der Elementanzahl in der Initialisierung spezifiziert. Wird eine Reihung nicht direkt initialisiert, so beschwert sich der Compiler darüber, dass er nicht bestimmen kann, wieviel Elemente die Reihung benötigt:

UnknownSize.c
```
1 int main(){
2   int xs [];
3   return 0;
4 }
```

Die Übersetzung dieses Programms führt zu einem Fehler:

```
sep@pc305-3:~/fh/c/student> gcc src/UnknownSize.c
src/UnknownSize.c: In function ‘main’:
src/UnknownSize.c:2: error: array size missing in ‘xs’
sep@pc305-3:~/fh/c/student>
```

Der C-Übersetzer muß immer in der Lage sein, die Anzahl der Elemente einer Reihung abzuleiten. Der Grund dafür ist, daß sobald eine Variable deklariert wird, der C Compiler den Speicherplatz für die entsprechende Variable anfordern muß. In C bedeutet das bei einer Reihung, daß der Speicher für eine komplette Reihung angefordert wird. Hierfür muß die Anzahl der Elemente bekannt sein.

Man kann die Anzahl der Elemente in den eckigen Klammern bei einer Arraydeklaration angeben. Im folgenden Beispiel wird ein Array mit 15 Elementen angelegt.

KnownSize.c
```
1 #include <stdio.h>
2
3 int main(){
4   int xs [15];
5   printf("%i\n",xs[2]);
6   return 0;
7 }
```

Die Elementanzahl für eine Reihung muß nicht zur Übersetzungszeit bekannt sein, sondern kann dynamisch errechnet werden und z.B. als Parameter einer Funktion übergeben werden:

LocalArray.c

```
#include <stdio.h>

void f(int l){
  int xs [l];
  int i;
  for (i= 0; i<l;i=i+1){
    xs[i]=i;
    printf("%i,", xs[i]);
  }
  printf("\n");
}

int main(){
  f(7);
  f(2);
  return 0;
}
```

Wie wir uns in der Ausgabe überzeugen können, werden in diesem Programm in der Funktion `f` nacheinander zwei Reihungen unterschiedlicher Länge erzeugt.[5]

4.7.1 Die Größe einer Reihung

Einer Reihung kann man im allgemeinen nicht ansehen, wie viele Elemente sie enthält. Das ist recht unangenehm, denn somit kann es z.B. passieren, das versucht wird bei einer Reihung mit 10 Elementen, auf das 100. Element zuzugreifen.

WrongIndex.c

```
#include <stdio.h>
int main(){
  int xs [10];
  printf("%i\n",xs[100]);
  return 0;
}
```

Es gibt niemanden, der uns in diesem Programm auf die Finger haut. Wir können es comilieren und laufen lassen. Es läuft sogar ohne Fehlermeldung und Programmabbruch:

```
sep@pc305-3:~/fh/c/student> bin/WrongIndex
-1073744294
sep@pc305-3:~/fh/c/student>
```

Allerdings wird hier irgendwo in den Speicher gegriffen und der zufällig dort gespeicherte Wert gelesen.

[5] Angeblich sollen bestimmte Compiler derartige Programme nicht übersetzen.

4.7.2 Reihungen sind nur Zeiger

Wir können Funktionen schreiben, die Reihungen als Parameter übergeben bekommen. Diese Reihungen lassen sich im Funktionsrumpf ganz normal als Reihungen behandeln:

AddInts.c

```
1  #include <stdio.h>
2
3  int add(int xs [],int argc){
4    int result=0;
5    int i=0;
6    for (;i<argc;i++){
7      result=result+xs[i];
8    }
9    return result;
10 }
11
12 int main(){
13   int xs [] = {1,2,3,4};
14   printf("%i\n", add(xs,4));
15   return 0;
16 }
```

Wir können jetzt einmal in diesem Programm einen Fehler einbauen, um zu sehen, als was für einen Typ der Compiler unseren Reihungsparameter betrachtet:

WrongArrayParameter.c

```
1  #include <stdio.h>
2
3  int add(int xs [],int argc){
4    int result=0;
5    int i=0
6    for (;i<argc;i++){
7      result=result+xs[i];
8    }
9    return result;
10 }
11
12 int main(){
13   int xs [] = {1,2,3,4};
14   printf("%i\n", add(xs,xs));
15   return 0;
16 }
```

Auf diese Weise erzwingen wir eine Fehlermeldung des Compilers, in der die erwarteten Parametertypen der Funktion `add` angegeben werden. Der Kompilierversuch führt zu folgender Fehlermeldung:

```
sep@pc305-3:~/fh/c/student> gcc -Wall src/WrongArrayParameter.c
src/WrongArrayParameter.c: In function ‘main’:
src/WrongArrayParameter.c:14: warning: passing arg 2 of ‘add’ makes integer from pointer without a cast
sep@pc305-3:~/fh/c/student>
```

Der Übersetzer ist der Meinung, der zweite Parameter, den wir der Funktion `add` übergeben, ist ein Zeiger. Der Übersetzer macht aus einem Reihungsparameter implizit einen Zeiger. Und zwar einen Zeiger auf das erste Element des Reihungsparameters. Ein Array `int xs []` ist also nichts weiter als ein Zeiger des Typs `int* xs`.

Die Schreibweise der eckigen Klammern für Reihungen für den Elementzugriff, läßt sich somit als eine versteckte Zeigerarithmetik interpretieren. `xs[i]` bedeutet dann: `*(xs+i)`.

Tatsächlich können wir ohne die Syntax der eckigen Klammern mit Reihungen arbeiten. Obiges Programm läßt sich auch wie folgt formulieren:

NoBrackets.c

```
1  #include <stdio.h>
2
3  int add(int xs [],int argc){
4    int result=0;
5    int i=0;
6    for (;i<argc;i++){
7      result=result + *(xs+i);
8    }
9    return result;
10 }
11
12 int main(){
13   int xs [] = {1,2,3,4};
14   printf("%i\n", add(xs,4));
15   return 0;
16 }
```

Auch diese Version hat als Ergebnis die 10.

Aufgabe 38 Schreiben sie eine Funktion
`void reverse(int* xs,int length)`,
die die Reihenfolge der Elemente des Arrays gerade einmal umdreht.

4.7.3 Reihungen als Rückgabewert

Im letzem Abschnitt haben wir bereits gesehen, dass bei der Parameterübergabe von Reihungen diese implizit als Zeiger auf das erste Element behandelt werden. Gleiches gilt auch für die Rückgabewerte von Funktionen. Dadurch lassen sich einige Probleme einhandeln. Versuchen wir einmal eine einfache Funktion zu schreiben, die eine neue Reihungsvariable anlegt und diese als Ergebnis zurückgibt:

WrongArrayReturn.c

```
1  #include <stdio.h>
2
3  int*  newArray(int length){
4    int result [length];
5    int i=0;
6    for (;i<5;i++)result[i]=i;
```

```
7   return result;
8 }
9
10 int main(){
11    int* xs = newArray(5);
12    int i=0;
13    for (;i<5;i++){
14      printf("%i ",xs[i]);
15    }
16    printf("\n");
17    return 0;
18 }
```

Die Übersetzung dieser Klasse glückt zwar, aber der Übersetzer gibt uns eine ernst zu nehmende Warnung:

```
sep@pc305-3:~/fh/c/student> gcc -std=c99 src/WrongArrayReturn.c
src/WrongArrayReturn.c: In function ‘newArray’:
src/WrongArrayReturn.c:7: warning: function returns address of local variable
sep@pc305-3:~/fh/c/student> .
```

Die Warnung ist insofern ernst zu nehmen, als daß wir tatsächlich ein undefiniertes Laufzeitverhalten bekommen:

```
sep@pc305-3:~/fh/c/student> ./a.out
0 134518380 808989496 134513155 1074069176
sep@pc305-3:~/fh/c/student>
```

Den Grund hierfür kennen wir bereits aus unseren Kapitel über Zeiger. Implizit behandelt der Übersetzer die Zeile `return result` als eine Rückgabe des Zeigers auf die lokale Variable `result`. Eine lokale Variable innerhalb einer Funktion, wird im Speicher nach Beendigung der Funktion wieder freigegeben. Zeigen wir noch auf diese Variable, so werden wir an ihrer Speicherzelle undefinierte Werte sehen.

Die einzige vernünftigen Möglichkeiten der Rückgabe von Reihungen in C sind die Rückgabe eines als Parameter überreichten Zeiger auf eine Reihung, oder aber der Zeiger auf eine dynamisch neu erzeugte Reihung:

ReturnArray.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int* newArray(int length){
5    int* result =  (int*)malloc(sizeof(int[length]));
6    int i=0;
7    for (;i<5;i++)result[i]=i;
8    return result;
9  }
10
11 int main(){
12   int* xs = newArray(5);
```

```
13    int i=0;
14    for (;i<5;i++){
15      printf("%i ",xs[i]);
16    }
17    printf("\n");
18    return 0;
19  }
```

Dieses Programm übersetzt ohne Warnung und liefert die erwartete Ausgabe.

```
sep@pc305-3:~/fh/c/student> ./bin/ReturnArray
0 1 2 3 4
sep@pc305-3:~/fh/c/student>
```

Tatsächlich findet man es relativ selten in C Programmen, daß eine Funktion eine Reihung als Rückgabewert hat. Zumeist bekommen Funktionen Reihungen als Parameter übergeben und manipulieren diese.

4.7.4 Funktionen höherer Ordnung für Reihungen

Wir haben bereits gesehen, daß Funktionen über Funktionszeigern an andere Funktionen als Parameter übergeben werden können. Im Zusammenhang mit Reihungen eröffnet dieses ein mächtiges Programmierprinzip. Fast jede Funktion, die sich mit Reihungen beschäftigt, wird einmal jedes Element der Reihung durchgehen und dazu eine `for`-Schleife benutzen. Wie wäre es denn, wenn man dieses Prinzip verallgemeinert und eine allgemeine Funktion schreibt, die einmal jedes Element einer Reihung betrachtet und etwas mit ihm macht. Was mit jedem Element zu tun ist, wird dieser Funktion als Funktionszeiger übergeben.

Funktionen, die als Parameter Funktionen übergeben bekommen, werden auch als Funktionen höherer Ordnung bezeichnet.

Beispiel 4.7.1 *Wir schreiben für Reihungen ganzer Zahlen die Funktion* `map`*, die jedes Element einer Reihung mit einer übergebenen Funktion umwandelt:*

IntMap.c

```
1  #include <stdio.h>
2
3  void map(int xs [],int l,int (f) (int)){
4    int i=0;
5    for (;i<l;i++)  xs[i]=f(xs[i]);
6  }
```

Wir stellen drei Funktionen, die eine ganze Zahl als Parameter und Ergebnis haben zur Verfügung:

IntMap.c

```
7  int add5(int x){return x+5;}
8  int square(int x){return x*x;}
9  int doubleX(int x){return 2*x;}
```

Jetzt können wir die Funktion `map` *verwenden, um für jedes Element einer Reihung, eine derartige Funktion anzuwenden. Zur Ausgabe von Reihungen schreiben wir eine kleine Hilfsfunktion:*

IntMap.c

```
10 void printArray(int xs [],int l){
11   printf("[");
12   int i=0;
13   for (;i<l;i++){
14     printf("%i",xs[i]);
15     if (i!=l-1) printf(", ");
16   }
17   printf("]\n");
18 }
19
20 int main(){
21   int xs [] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
22   map(xs,15,add5);
23   printArray(xs,15);
24   map(xs,15,square);
25   printArray(xs,15);
26   map(xs,15,doubleX);
27   printArray(xs,15);
28   return 0;
29 }
```

Und hier die Ausgabe des Programms:

```
sep@pc305-3:~/fh/c/student> bin/IntMap
[6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
[36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400]
[72, 98, 128, 162, 200, 242, 288, 338, 392, 450, 512, 578, 648, 722, 800]
sep@pc305-3:~/fh/c/student>
```

Zunächst wird auf jedes Element der Reihung 5 aufaddiert. Dann werden die Elemente quadriert und schließlich verdoppelt.

Aufgabe 39 Schreiben Sie eine Funktion `fold` mit folgender Signatur:

Fold.h

```
1 int fold(int xs [],int length,int op(int,int),int startV);
```

Die Spezifikation der Funktion sei durch folgende Gleichung gegeben:

$$\text{fold(xs,l,op,st)} = \text{op}(\ldots\text{op}(\text{op}(\text{op}(\text{st,xs[0]}),\text{xs[1]}),\text{xs[2]})\ldots,\text{xs[l-1]})$$

Oder bei Infixschreibweise der Funktion `op` gleichbedeutend über folgende Gleichung:

$$\text{fold}(\{x_0, x_1,,\ldots,x_{l-1}\},\text{l,op,st}) = st \text{ op } x_0 \text{ op } x_1 \text{ op } x_2 \text{ op } \ldots \text{op } x_{l-1}$$

Testen Sie Ihre Methode `fold` mit folgenden Programm:

FoldTest.c

```
1  #include <stdio.h>
2  #include "Fold.h"
3
4  int add(int x,int y){return x+y;}
5  int mult(int x,int y){return x*y;}
6
7  int fold(int xs [],int length,int (*foldOp)(int,int),int startV);
8
9  int main(){
10   int xs [] = {1,2,3,4,5};
11   printf("%i\n",fold(xs,5,add,0));
12   printf("%i\n",fold(xs,5,mult,1));
13   return 0;
14 }
```

4.7.5 Sortieren von Reihungen mit bubblesort

Auf Reihungen läßt sich einer der populärsten Algorithmen der Infomatik implementieren. Es der sogenannte *bubble sort* zum Sortieren der Elemente einer Sammlung, in unserem Fall einer Reihung, nach einer bestimmten Ordnung.

Der Name *bubble sort* leitet sich davon ab, daß Elemente wie die Luftblasen in einem Mineralwasserglass innerhalb der Reihung aufsteigen, wenn sie laut der Ordnung an einen höheren Platz gehören. Ein vielleicht phonetisch auch ähnlicher klingender deutscher Name wäre *Blubbersortierung*. Dieser Name ist jedoch nicht in der deutschen Terminologie etabliert und es wird in der Regel der englische Name genommen.

Die Idee des *bubble sort* ist, jeweils nur zwei benachbarte Elemente einer Reihung zu betrachten und diese gegebenenfalls in ihrer Reihenfolge zu vertauschen. Eine Reihung wird also von vorne bis hinten durchlaufen, immer zwei benachbarte Elemente betrachtet und diese, falls das vordere nicht kleiner ist als das hintere, getauscht. Wenn die Liste in dieser Weise einmal durchgegangen wurde, ist sie entweder fertig sortiert oder muß in gleicher Weise nocheinmal durchgegangen werden, solange, bis keine Vertauschungen mehr vorzunehmen sind. Allerdings braucht man bei einem weiteren Durchgang, das letzte Element nicht mehr mit dem vorletzten zu vergleichen, da ein Blubberdurchgang sicherstellt, dass das größte Element an die oberste Stelle kommt.

Betrachten wir ein Beispiel:
Die Reihung `("z","b","c","a")` wird durch den *bubble sort*-Algorithmus in folgender Weise sortiert:

1. Bubble-Durchlauf

```
("z","b","c","a")
("b","z","c","a")
("b","c","z","a")
("b","c","a","z")
```

Das Element `"z"` ist in diesem Durchlauf an das Ende der Reihung geblubbert.

2. Bubble-Durchlauf

```
("b","c","a","z")
("b","a","c","z")
```

In diesem Durchlauf ist das Element `"c"` um einen Platz nach hinten geblubbert.

3. Bubble-Durchlauf
`("b","a","c","z")`
`("a","b","c","z")`

Im letzten Schritt ist das Element `"b"` auf seine endgültige Stelle geblubbert.

Implementierung

Blubbersortierung läßt sich relativ leicht für Reihungen implementieren. Eine Reihung ist dabei nur mehrfach zu durchlaufen und nebeneinanderstehende Elemente sind eventuell in der Reihenfolge zu vertauschen. Dieses wird so lange gemacht, bis keine Vertauschung mehr stattfindet, maximal aber um eines weniger, als die Reihung Elemente hat.

BubbleSort1.c

```
1  #include <stdio.h>
2  typedef enum {false,true} boolean;
3
4  void bubble(int* xs, int i){
5    boolean changed = true ;
6    for (;changed && i>1;i--){
7      changed=false;
8      int j=0;
9      for (;j<(i-1);j++){
10       if (xs[j]>xs[j+1]){
11         int tmp=xs[j];
12         xs[j]=xs[j+1];
13         xs[j+1]=tmp;
14         changed=true;
15       }
16     }
17   }
18 }
19
20 int main(){
21   int xs [] = {321,43,43,5345,54,543,543,543,43,1,4,2};
22   bubble(xs,12);
23   int i=0;
24   for (;i<12;i++)printf("%i,",xs[i]);
25   printf("\n");
26   return 0;
27 }
```

Implementierung mit Hilfsfunktionen

Wem die obige Implementierung zu komplex innerhalb einer Funktion ist, der kann sie sinnvoller Weise auf mehrere Funktionen splitten. Drei Aufgabe sind bei diesem Sotrierverfahren zu berwerkstelligen:

- jeweils zwei Elemente eines Arrays sind zu vertauschen
- ein Array ist einmal vom Anfang bis zu einem vorgegebenen Index zu durchlaufen. Sind zwei benachbarte Elemente nicht in der richtigen Reihenfolge, so sind diese zu vertauschen.
- es ist so oft durch einen Array immer wieder zu durchlaufen, bis er sortiert ist.

Entsprchend läßt sich die Sortierung mit drei kleinen Teilfunktionen implementieren:

Zunächst die kleine Hilfsmethode zum Vertauschen zweier Elemente eines Arrays. Die Indizes dieser Elemente werden als weitere Parameter übergeben:

BubbleSort2.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  typedef enum {false,true} boolean;
4
5  void swap(int* xs,int i,int j){
6    int tmp=xs[i];
7    xs[i]=xs[j];
8    xs[j]=tmp;
9  }
```

Als zweite Funktion kann das eigentliche *bubblen* realsiert werden:

BubbleSort2.c

```
10 boolean bubble(int* xs,int i){
11   boolean result = false;
12   int j=0;
13   for (;j<(i-1);j++){
14     if (!xs[j]<xs[j+1]){
15       swap(xs,j,j+1);
16       result=true;
17     }
18   }
19   return result;
20 }
```

Dieses wird nun so oft auf einem Array angewendet, bis dieser sortiert ist.

BubbleSort2.c

```
21 void bubbleSort(int* xs, int length){
22   boolean changed = true;
23   for (;changed && length>1;length--)
24     changed=bubble(xs,length);
25 }
```

Als kleine Demonstration noch einmal die Testfunktion:

BubbleSort2.c

```
26 int main(){
27   int xs [] = {321,43,43,5345,54,543,543,543,43,1,4,2};
28   bubbleSort(xs,12);
29   int i=0;
30   for (;i<12;i++)printf("%i,",xs[i]);
31   printf("\n");
32   return 0;
33 }
```

Sortierrelation übergeben

Ein Sortieralgorithmus ist vollkommen unabhängig davon, wonach sortiert werden soll. Die obige Implementierung kann nur sortieren, dass erst die kleinen und dann die großen Zahlen kommen. Wenn wir umgekehrt von groß nach klein sortieren wollen, muß der ganze Algorithmus komplett neu implementiert werden, nur um ein einziges Zeichen zu ändern, nämlich das Kleinerzeichen ins Größerzeichen. Das würde zu einer immensen Codeverdoppelung führen. Daher wäre es doch schön, den *bubblesort* so zu implementieren, dass flexibel gehalten ist, nach welcher Sotrierrelation sortiert werden soll. Da wir in C Funktionen als Parameter übergeben können, können wir der Sotrierfunktion als zusätzlichen Parameter eine Funktion übergeben, die für zwei Elemente *true* zurückgibt, wenn das erste kleiner als das zweite ist.

So implementieren wir *bubblesort* als Funktion höherer Ordnung. Die Funktion `swap` ist davon nicht betroffen:

BubbleSort3.c

```
1 #include <stdio.h>
2 typedef enum {false,true} boolean;
3
4 void swap(int* xs,int i,int j){
5   int tmp=xs[i];
6   xs[i]=xs[j];
7   xs[j]=tmp;
8 }
```

Die Funktion `bubble` bekommt einen zusätzliche Parameter. Eine Funktion, die für zwei Zahlen entscheidet, ob die erste kleiner in einer Relation ist. Diese wird dann verwendet, beim Vergleich der benachbarten zwei Arrayelemente:

BubbleSort3.c

```
9  boolean bubble(int* xs,int i,boolean smaller(int,int)){
10   boolean result = false;
11   int j=0;
12   for (;j<(i-1);j++)
13     if (!smaller(xs[j],xs[j+1])){
14       swap(xs,j,j+1);
15       result=true;
16     }
17   return result;
18 }
```

Ebenso braucht die Funktion `bubbleSort` diesen Funktionsparameter, um ihn an die Funktion `bubble` weiterzureichen:

BubbleSort3.c

```
19 void bubbleSort(int* xs, int length,boolean smaller(int,int)){
20   boolean changed = true;
21   for (;changed && length>1;length--)
22     changed=bubble(xs,length,smaller);
23 }
```

Soweit der Bubblesort mit der Sortierrelation als Parameter. Jetzt seien einmal drei verschiedene Sortierrelationen zu schreiben. Einmal sollen kleine vor großen Zahlen kommen, einmal umgekehrt, und in der dritten Sortierrelation seien ungerade Zahlen prinzipiell kleiner als gerade Zahlen:

BubbleSort3.c

```
24 boolean le(int x,int y){return x<y;}
25 boolean ge(int x,int y){return x>y;}
26 boolean oddFirst(int x,int y){return x%2>y%2 || (x%2==y%2 && x<y);}
```

Für die nun folgenden Test, sei wieder eine kleine Ausgabefunktion für Reihungen geschrieben:

BubbleSort3.c

```
27
28 void printArray(int* xs,int length){
29   int i=0;
30   printf("[");
31   for (;i<length;i++){
32     printf("%i",xs[i]);
33     if (i<length-1) printf(", ");
34   }
35   printf("]\n");
36 }
```

Und jetzt können wir ein und dieselbe Sortierfunktion benutzen, um nach ganz unterschiedlichen Relationen zu sortieren:

BubbleSort3.c

```
37 int main(){
38   int xs [] = {321,43,43,5345,54,543,543,543,43,1,4,2};
39   bubbleSort(xs,12,le);
40   printArray(xs,12);
41   bubbleSort(xs,12,ge);
42   printArray(xs,12);
43   bubbleSort(xs,12,oddFirst);
44   printArray(xs,12);
45   return 0;
46 }
```

Und tatsächlich wird die Reihung dreifach unterschiedlich sortiert:

```
sep@pc305-3:~/fh/c/student> bin/BubbleSort3
[1, 2, 4, 43, 43, 43, 54, 321, 543, 543, 543, 5345]
[5345, 543, 543, 543, 321, 54, 43, 43, 43, 4, 2, 1]
[1, 43, 43, 43, 321, 543, 543, 543, 5345, 2, 4, 54]
sep@pc305-3:~/fh/c/student>
```

Beliebige Zeigerelemente sortieren

Im letzten Abschnitt haben wir die Sortierrelation, über die sortietr werden variabel gehalten und als Parameter übergeben. Wir haben aber stets nur Reihungen, die ganze Zahlen als Elemente enthalten haben, sortiert. Im Prinzip ist einem Sortieralgorithmus auch gleichgültig, um was für Elemente es sich in einer Reihung handelt, solange eine Sotrierrelation auf diesen Elementen bekannt ist.

So soll jetzt der Bubblesort so modifiziert werden, dass er beliebige Datenobjekte sortieren kann. Die einziege Möglichkeit, die es in C gibt über beliebige Daten zu sprechen, ist über den `void`-Zeiger. Um unser vorgehen noch ein wenig plastischer zu machen, sei der Typ `void*` als `Object` bezeichnet:

BubbleSort4.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef void* Object;
```

Die Funktion `swap` ändert sich im Prinzip nicht. Wir müssen nur berücksichtigen, dass nun nicht in einer Reihung mit `int`-Zahlen zwei Elemente vertauscht werden, sondern in einer Reihung von Objekten:

BubbleSort4.c

```
5  typedef enum {false,true} boolean;
6
7  void swap(Object* xs,int i,int j){
8    Object tmp=xs[i];
9    xs[i]=xs[j];
10   xs[j]=tmp;
11 }
```

Tatsächlich ändert sich auch nichts in der Funktion `bubble`, bis auf dass der Typ `int` durch den Typ `Object` ersetzt wurde:

BubbleSort4.c

```
12 boolean bubble(Object* xs,int i,boolean smaller(Object,Object)){
13   boolean result = false;
14   int j=0;
15   for (;j<(i-1);j++)
16     if (!smaller(xs[j],xs[j+1])){
17       swap(xs,j,j+1);
18       result=true;
19     }
20   return result;
21 }
```

Das gleiche gilt für die Funktion `bubbleSort`:

BubbleSort4.c

```
22 void bubbleSort(Object* xs, int length,boolean smaller(Object,Object)){
23   boolean changed = true;
24   for (;changed && length>1;length--)
25     changed=bubble(xs,length,smaller);
26 }
```

In der Funktion `printArray` bekommen wir ein kleines Problem. Wir wissen nicht, um was für Objekte es sich in der Reihung handelt, somit können wir auch nicht wissen, wie diese Objekte auszugeben sind. Daher benötigen wir noch eine weitere Funktion als Parameter übergeben. Diese soll ein Objekt auf der KOmmandozeile ausgeben:

BubbleSort4.c

```
27 void printArray(Object* xs,int length,void printer(Object)){
28   int i=0;
29   printf("[");
30   for (;i<length;i++){
31     printer(xs[i]);
32     if (i<length-1) printf(", ");
33   }
34   printf("]\n");
35 }
```

Soweit die Implementierung der allgemeinen Sortierung. Wir haben nun zum einen die Art der Daten, die sortiert werden sollen, zum anderen die Sortierrelation offen gelassen. Um die Sortierung zu testen, seien ein paar Klassen definiert. Zunächst einmal mehr die schon bekannte Punktklasse für Punkte im zweidimensionalen Raum:

BubbleSort4.c

```
36 typedef struct {
37   int x;
38   int y;
39 } PunktStrukt;
40
41 typedef PunktStrukt* Punkt;
```

Es seien wieder entsprechend Konstruktor und Destruktor definiert, sowie eine einfache Funktion für das Quadrat der Entfernung zum Nullpunkt:

BubbleSort4.c

```
42 Punkt newPunkt(int x,int y){
43   Punkt this = (Punkt)malloc(sizeof(PunktStrukt));
44   this->x=x;
45   this->y=y;
46   return this;
47 }
48
49 void deletePunkt(Punkt this){
```

```
50   free(this);
51 }
52
53 int square(int x){return x*x;}
54 int betragQuadrat(Punkt p){return square(p->x)+square(p->y);}
55
```

Um Punktobjekte zu sortieren, brauchen wir auf Punktobjekten eine Sortieerrelation. Punkte, die näher am Nullpunkt liegen, seien kleiner als weiter entfernte in dieser Relation:

BubbleSort4.c

```
56 boolean lePunkt(Object o1,Object o2){
57   Punkt p1=(Punkt)o1;
58   Punkt p2=(Punkt)o2;
59   return betragQuadrat(p1)<betragQuadrat(p2);
60 }
```

Wir müssen auch eine Methode zur Ausgabe eines Punktobjektes definieren:

BubbleSort4.c

```
61 void printPunkt(Object p){
62   printf("(%i,%i)",((Punkt)p)->x,((Punkt)p)->y);
63 }
```

Als zweites wollen wir auch wieder ganze Zahlen sortieren, allerdings können wir nur verzeigerte Objekte sortieren. Somit sei aich eine Klasse die `int`-Zahlen repräsentiert, definiert. Auch diese Klasse sei gleich mit Konstruktor, destruktor, der `print`-Methode und schließlich einer Kleinerrelation definiert:

BubbleSort4.c

```
64 typedef int* Integer;
65
66 Integer newInt(int x){
67   Integer result =(void*)malloc(sizeof(int));
68   *result=x;
69   return result;
70 }
71
72 void deleteInt(Integer this){free(this);}
73
74 void printInt(Object p){printf("%i",*((Integer)p));}
75
76 boolean leInt(Object o1,Object o2){
77   Integer i1=(Integer)o1;
78   Integer i2=(Integer)o2;
79   return *i1<*i2;
80 }
```

Als dritte Klasse von Elemente, die sortiert werden sollen, sei nun noch eine Klasse für Wahrheitswerte definiert. Auch diese wieder mit Konstruktor, Destruktor, `print`-Methode und schließlich einer Kleinerrelation:

BubbleSort4.c

```
81  typedef boolean* Boolean;
82
83  Boolean newBoolean(boolean x){
84    Boolean result =(Boolean)malloc(sizeof(boolean));
85    *result = x;
86    return result;
87  }
88
89  void printBoolean(Object p){
90    if (*((Boolean)p)) printf("true");
91    else printf("false");
92  }
93
94  boolean leBoolean(Object o1,Object o2){return *(Boolean)o1;}
```

Nun kann es losgehen. Zunächst sei eine Reihung von Punkten definiert:

BubbleSort4.c

```
95  int main(){
96    Punkt xs [] = {newPunkt(321,43)
97                  ,newPunkt(43,5345)
98                  ,newPunkt(54,543)
99                  ,newPunkt(543,543)
100                 ,newPunkt(43,1)
101                 ,newPunkt(4,2)};
102   bubbleSort((Object*)xs,6,lePunkt);
103   printArray((Object*)xs,6,printPunkt);
104   int i=0;
105   for (;i<6;i++) deletePunkt(xs[i]);
```

Jetzt können wir dieselbe Sortierfunktion benutzen, um eine Reihung von Zahlen zu sortieren:

BubbleSort4.c

```
106   Object ys [6];
107   for (i=0;i<6;i++)ys[i]=newInt(42-i);
108
109   bubbleSort(ys,6,leInt);
110   printArray(ys,6,printInt);
111   for (i=0;i<6;i++) deleteInt(ys[i]);
```

Und auf die gleiche Weise können wir auch eine Reihung von Wahrheitswerten sortieren:

BubbleSort4.c

```
112   for (i=0;i<6;i++) ys[i]=((Object)newBoolean(i%3==0));
113   bubbleSort(ys,6,leBoolean);
114   printArray(ys,6,printBoolean);
115
116   return 0;
117 }
```

Und tatsächlich werden die drei Reihungen mit ihren ganz unterschiedlichen Daten alle korrekt sortiert:

```
sep@pc305-3:~/fh/c/student> bin/BubbleSort4
[(4,2), (43,1), (321,43), (54,543), (543,543), (43,5345)]
[37, 38, 39, 40, 41, 42]
[true, true, false, false, false, false]
sep@pc305-3:~/fh/c/student>
```

4.8 Arbeiten mit Texten

Wahrscheinlich die am häufigsten benötigten Daten sind Texte. Texte unbestimmter Länge, die Wachsen und Schrumpfen können. Texte die editiert werden sollen und modifiziert. Unglücklicher Weise stellt C für Texte keinen eigenen Typ zur Verfügung. Damit steht C als Programmiersprache ziemlich allein dar, denn fast alle neueren Programmiersprachen bieten komfortable Mittel an, um Texte in Form von Zeichenketten zu bearbeiten.

Dabei haben wir bereits gesehen, dass auch C nicht gänzlich blind für Zeichenketten ist, denn es wird ja ein entsprechendes Literal angeboten. Zeichenketten können in dopppelten Hochkommas eingeschlossen als Literale in C ausgedrückt werden. Es fragt sich, wenn es nicht wie in anderen Programmiersprachen einen Typ `string` für Zeichenketten gibt, als was für einen Typ werden die `string`-Literale dann gespeichert?

Die Antwort liegt in den geraden kennengelernten Reihungen. Ein `string` ist eine Reichung vom einzelnen Werten des Typs `char`, und da es sich bei Reihungen lediglich um einen Zeiger auf das erste Reihungselement handelt, sind tatsächlich Zeichenketten als Werte des Typs `char*` gespeichert.

Intern hat jeder C-String ein Element mehr als Zeichen. In diesem letzten Element ist mit einer `0` markiert, daß der String jetzt zu Ende ist. Zeichenketten werden also intern durch ein Nullelement beendet. Dieses ist unabhängig von der eigentlichen technischen Länge der Reihung.

Initialisiert man eine Reihung von Zeichen mit der Mengenklammeraufzählung ist der beendende Nullwert unbedingt zu berücksichtigen.

CString2.c

```
1 #include <stdio.h>
2
3 int main(){
4   char hello[] =
5    { 'H', 'e', 'l', 'l', 'o', '\0' };
6   printf("%s\n",hello);
7   return 0;
8 }
```

Wird die Endemarkierung bei der Initialisierung vergessen, so blickt C unkontrolliert weiter in den Speicher und betrachtet alles nachfolgende als Zeichen der Zeichenkette, bis zufällig ein Nullwert kommt. Das folgende Programm erzeugt auch eine indeterministische Ausgabe:

CString3.c

```
1 #include <stdio.h>
2
```

```
3  int main(){
4    char hello[] =
5     { 'H', 'e', 'l', 'l', 'o'};
6    printf("%s\n",hello);
7    return 0;
8  }
```

```
sep@pc305-3:~/fh/c/student> ./bin/CString3
Hello¿½@@OGï¿½@
        sep@pc305-3:~/fh/c/student>
```

Für C ist eine Zeichenkette immer genau dann beendet, wenn der abschließende Nullwert auftritt. Dieses gilt auch, wenn er inmitten der Reihung auftritt.

Im folgenden Programm ist nach der Zuweisung des Nullwerts für das Element am Index 6 für C der String nach dem Wort `hello` beendet, obwohl die Reihung länger ist und noch weitere sinnvolle Zeichen in der Reihung gespeichert sind.

CString4.c

```
1  #include <stdio.h>
2
3  int main(){
4    char hello [] = "hello world!";
5    hello[5] ='\0';
6
7    printf("%s\n",hello);
8    return 0;
9  }
```

```
 sep@pc305-3:~/fh/c/student> ./bin/CString4
hello
sep@pc305-3:~/fh/c/student>
```

In traditioneller C-Programmierung gibt es typischer Weise Reihungen einer maximalen Länge, in denen unterschiedlich lange Strings gespeichert sind. Für das Programm bedeutet das, daß die Stringwerte an diesen Stellen eine bestimmte Maximallänge nicht überschreiten dürfen. Daher kommt in vielen älteren Programmen wahrscheinlich die Restriktion, daß bestimmte Namen, Pfade oder Optionen nur eine bestimmte Länge haben dürfen.

StringLength.h

```
1  #ifndef STRING_LENGTH__H_
2  #define STRING_LENGTH__H_
3
4  unsigned int stringLaenge(char* xs);
5  #endif
```

StringLength.c

```
1  #include "StringLength.h"
2  unsigned int stringLaenge(char* xs){
3    int result=0;
```

```
4   int i = 0;
5   for (;xs[i]!='\0';i++){
6     result++;
7   }
8   return result;
9 }
```

TestStringLength.c

```
1 #include "StringLength.h"
2 #include <stdio.h>
3 int main(){
4   char* str = "hallo";
5   printf("der String \"%s\" hat %i Zeichen\n",str,stringLaenge(str));
6   return 0;
7 }
```

StringToUpper.h

```
1 #ifndef STRING_TO_UPPER__H_
2 #define STRING_TO_UPPER__H_
3
4 void toUpper(char* source,char* target);
5 #endif
```

StringToUpper.c

```
1  #include "StringToUpper.h"
2  #define LOWER_TO_UPPER  'A'-'a';
3
4  char charToUpper(char c){
5    if (c>='a' && c <= 'z'){
6      return c+LOWER_TO_UPPER;
7    }
8    return c;
9  }
10
11 void toUpper(char* source,char* target){
12   int i=0;
13   for (;source[i]!='\0';i++){
14     target[i] = charToUpper(source[i]);
15   }
16   target[i]='\0';
17 }
```

TestToUpper.c

```
1 #include "StringToUpper.h"
2 #include "StringLength.h"
3 #include <stdio.h>
4 int main(){
5   char* str = "hallo Du da!";
```

```
6   char erg [stringLaenge(str)+1];
7   toUpper(str,erg);
8   printf("der String \"%s\" in Grossbuchstaben: \"%s\"\n",str,erg);
9   return 0;
10 }
```

Kommandozeilenparameter

Auch für die Hauptmethode hat es Auswirkungen, daß C Reihungen ihre Elementanzahl nicht kennen. Unsere bisherigen Hauptmethoden hatten keine Argumente. Man kann zur Kommandozeilenübergabe in C aber auch eine Hauptmethode schreiben, der in einer Reihung die Kommandozeilenoptionen übergeben werden. Hierzu braucht man dann aber auch ein weiteres Argument, das die Länge der übergebenen Reihung mit angibt.

CommandLineOption.c

```
1  #include <stdio.h>
2
3  int main(int argc, char* argv []){
4    int i;
5    for (i=0;i<argc;i++){
6      char* arg = argv[i];
7      printf("%s\n",arg);
8    }
9    return 0;
10 }
```

Wie man an dem folgenden Testlauf sieht, wird als erstes Reihungselement der Name des Programms übergeben.

```
sep@pc305-3:~/fh/c/student> ./bin/CommandLineOption hallo C -v gruss
./bin/CommandLineOption
hallo
C
-v
gruss
sep@pc305-3:~/fh/c/student>
```

4.9 Typsumme mit Unions

Als Summentyp wird in der Informatik ein Typ bezeichnet, der die Vereinigung einer oder mehrerer Typen darstellt. Auch hierfür stellt C ein eigenes Konstrukt zur Verfügung: die Vereinigung von Typen, die `union`.

Syntaktisch sieht eine Typvereinigung einer Struktur sehr ähnlich. Lediglich das Schlüsselwort `struct` ist durch das Schlüsselwort `union` zu ersetzen. Somit läßt sich eine Vereinigung der Typen `int` und `char*` definieren als:
`union stringOrInt {int i;char *s;};`.

Daten des Typs `union stringOrInt` sind nun entweder eine Zahl, die über das Attribut `i` angesprochen werden kann, oder der Zeiger auf das erstes Element einer Zeichenkette, der über das Attribut `s` angesprochen werden kann.

Beispiel 4.9.1 *Im folgenden kleinem Programm wird eine Typsumme definiert und beispielhaft in einer Hauptmethode benutzt.*

Union1.c

```
1  #include <stdio.h>
2
3  union stringOrInt {int i;char *s;};
4
5  int main(){
6    union stringOrInt sOrI;
7    sOrI.s="hallo da draussen";
8    printf("%s\n", sOrI.s);
9
10   sOrI.i=42;
11   printf("%i\n", sOrI.i);
12   return 0;
13 }
```

Wie man der Ausgabe entnehmen kann, wird in der Variablen des Typs `stringOrInt` *entweder ein* `int` *oder eine Zeichenkettenreferenz gespeichert:*

```
sep@linux:~/fh/prog3/examples/src> gcc -o Union1 Union1.c
sep@linux:~/fh/prog3/examples/src> ./Union1
hallo da draussen
42
sep@linux:~/fh/prog3/examples/src>
```

Im letzten Beispiel haben wir die Typsumme gutartig benutzt. Die Typsumme drückt aus, daß die Daten entweder von einem oder vom anderen Typ sind. Im Beispiel haben wir uns gemerkt, welchen Typ wir als letztes in der entsprechenden Variablen gespeichert haben und nur auf diesen Typ zugegriffen. C bewahrt uns aber nicht davor, dieses immer korrekt zu tun.

Beispiel 4.9.2 *In diesem Beispiel benutzen wir den selben Summentypen wie im vorhergehenden Beispiel. Jetzt interpretieren wir aber jeweils die Daten auch als die andere Typalternative.*

Union2.c

```
1  #include <stdio.h>
2
3  union stringOrInt {int i;char *s;};
4
5  int main(){
6    union stringOrInt sOrI;
7    sOrI.s="hallo da draussen";
8    printf("%s\n", sOrI.s);
9    printf("%i\n", sOrI.i);
10
11   sOrI.i=42;
12   printf("%s\n", sOrI.s);
13   printf("%i\n", sOrI.i);
14   return 0;
15 }
```

Wenn wir jetzt einen Adresse auf einen Zeichenkette in der Variablen `sOrI` *gespeichert haben und auf die Variable mit ihrem* `int`*-Attribut* `i` *zugreifen, so bekommen wir die gespeicherte Adresse als Zahl geliefert.*

Wenn wir umgekehrt eine Zahl gespeichert haben und diese nun als Adresse einer Zeichenkette lesen wollen, so kommt es in der Regel zu einem Fehler, weil wir höchstwahrscheinlich versuchen eine Adresse im Speicher zu lesen, auf die wir nicht zugreifen können.

```
sep@linux:~/fh/prog3/examples/src> ./Union2
hallo da draussen
134514920
42
Speicherzugriffsfehler
sep@linux:~/fh/prog3/examples/src>
```

Um Fehler, wie im letzem Beispiel zu vermeiden, ist es ratsam Strukturen, Aufzählungen und Typsummen geschachtelt zu verwenden. Hierzu kapselt man die Typsumme in eine Struktur. In dieser Struktur gibt ein zweites Attribut an, um welchen Typ es sich gerade in der Typsumme handelt. Für das Attribut, das den Typ kennzeichnet, bietet sich an, einen Aufzählungstypen zu verwenden.

Beispiel 4.9.3 *In diesem Beispiel definieren wir eine Typsumme, die in einer Struktur eingebettet ist. Dazu definieren wir drei Typen: eine Aufzählung, eine Typsumme und eine Struktur.*

Union3.c

```
1 #include <stdio.h>
2
3 typedef enum   {INT,STRING} TypeMarker;
4 typedef union  {int i;char * s;} stringOrInt;
5
6 typedef struct {TypeMarker type;stringOrInt v;} strOrInt;
```

Jetzt können wir uns zwei Funktionen schreiben, die jeweils eine der beiden Varianten unseres Summentypens konstruieren:

Union3.c

```
7  strOrInt forInt(int i){
8    strOrInt result;
9    result.type=INT;
10   result.v.i=i;
11   return result;
12 }
13
14 strOrInt forString(char * s){
15   strOrInt result;
16   result.type=STRING;
17   result.v.s=s;
18   return result;
19 }
```

Schließlich ein Beispiel für eine Funktion, die den neuen Typen benutzt. Hierzu ist zunächst zu fragen, von was für einen Typ denn die gespeichert sind, um dann entsprechend mit den Daten zu verfahren:

Union3.c

```
20 char * toString(strOrInt si){
21   char* result;
22   if (si.type==INT){
23     sprintf(result,"%d",si.v.i);
24   }else if (si.type==STRING){
25     result=si.v.s;
26   }
27   return result;
28 }
```

Abschließend ein kleiner Test in einer Hauptmethode:

Union3.c

```
29 int main(){
30   strOrInt test = forString("hallo da draussen");
31   printf("%s\n", toString(test));
32   test = forInt(42);
33   printf("%s\n", toString(test));
34   return 0;
35 }
```

In der Ausgabe können wir uns davon überzeugen, daß tatsächlich immer die Daten der Typsumme korrekt interpretiert werden:

```
sep@linux:~/fh/prog3/examples/src> gcc -o Union3 Union3.c
sep@linux:~/fh/prog3/examples/src> ./Union3
hallo da draussen
42
sep@linux:~/fh/prog3/examples/src>
```

Schließlich ist noch interessant, wieviel Speicher C für eine Typsumme reserviert. Auch dieses können wir experimentel erfragen:

Union4.cpp

```
1  #include <iostream>
2
3  typedef union {char a1 [15]; char a2 [200];}  U1;
4  typedef union {int a1; char a2 [16];}  U2;
5
6  int main(){
7    U1 u1;
8    U2 u2;
9    std::cout << sizeof(u1) << std::endl;
10   std::cout << sizeof(u2) << std::endl;
11  }
```

An der Ausgabe ist zu erkennen, daß der maximal notwendige Speicher für einen der Typen in der Summe angefordert wird.

```
sep@pc305-3:~/fh/c/student> bin/Union4
200
16
sep@pc305-3:~/fh/c/student>
```

4.10 Rekursive Daten

Als ein fundamentales Grundkonzept der Informatik haben wir die Rekursion kennengelernt. Wir haben rekursive Funktionen definiert. Eine rekursive Funktion ruft sich selbst in ihrem Rumpf wieder auf. Nicht nur Funktionen, auch Daten können rekursiv definiert werden. Rekursive Daten sind Daten, die sich selbst wieder enthalten.

4.10.1 Listen

Eine der häufigsten Datenstrukturen in der Programmierung sind Sammlungstypen. In fast jedem nichttrivialen Programm wird es Punkte geben, an denen eine Sammlung mehrerer Daten gleichen Typs anzulegen sind. Eine der einfachsten Strukturen, um Sammlungen anzulegen, sind Listen. Das Wesen einer Liste ist, dass ihre maximale Länge nicht in vornherein bekannt ist. Es kannt also nicht statisch zuvor ein Array im Speicher angelegt werden, in dem nach einander die Elemente gespeichert werden. Eine Liste soll die Möglichkeit haben, ohne vorgesetztes Limit beliebig wachsen zu können.

Formale Spezifikation

Wir werden Listen als abstrakten Datentyp formal spezifizieren. Ein abstrakter Datentyp (ADT) wird spezifiziert über eine endliche Menge von Funktionen. Hierzu wird spezifiziert, auf welche Weise Daten eines ADT konstruiert werden können. Dazu werden entsprechende Konstruktorfunktionen spezifiziert. Dann wird eine Menge von Funktionen definiert, die wieder Teile aus den konstruierten Daten selektieren können. Schließlich werden noch Testfunktionen spezifiziert, die angeben, mit welchem Konstruktor ein Datum erzeugt wurde.

Der Zusammenhang zwischen Konstruktoren und Selektoren sowie zwischen den Konstruktoren und den Testfunktionen wird in Form von Gleichungen spezifiziert.

Der Trick, der angewendet wird, um abstrakte Datentypen wie Listen zu spezifizieren, ist die Rekursion. Das Hinzufügen eines weiteren Elements zu einer Liste wird dabei als das Konstruieren einer neuen Liste aus der Ursprungsliste und einem weiteren Element betrachtet. Mit dieser Betrachtungsweise haben Listen eine rekursive Struktur: eine Liste besteht aus dem zuletzt vorne angehängten neuen Element, dem sogenannten Kopf der Liste, und aus der alten Teilliste, an die dieses Element angehängt wurde, dem sogenannten Schwanz der Liste. Wie bei jeder rekursiven Struktur bedarf es eines Anfangs der Definition. Im Falle von Listen wird dieses durch die Konstruktion einer leeren Liste spezifiziert.[6]

Konstruktoren Abstrakte Datentypen wie Listen lassen sich durch ihre Konstruktoren spezifizieren. Die Konstruktoren geben an, wie Daten des entsprechenden Typs konstruiert werden können. In dem Fall von Listen bedarf es nach den obigen Überlegungen zweier Konstruktoren:

[6] Man vergleiche es mit der Definition der natürlichen Zahlen: die 0 entspricht der leeren Liste, der Schritt von n nach $n + 1$ dem Hinzufügen eines neuen Elements zu einer Liste.

- einem Konstruktor für neue Listen, die noch leer sind.
- einem Konstruktor, der aus einem Element und einer bereits bestehenden Liste eine neue Liste konstruiert, indem an die Ursprungsliste das Element vorne angehängt wird.

Wir benutzen in der Spezifikation eine mathematische Notation der Typen von Konstruktoren.[7] Dem Namen des Konstruktors folgt dabei mit einem Doppelpunkt abgetrennt der Typ. Der Ergebnistyp wird von den Parametertypen mit einem Pfeil getrennt.

Somit lassen sich die Typen der zwei Konstruktoren für Listen wie folgt spezifizieren:

- Empty: () →**List**
- Cons: (**int**,**List**) →**List**

Der Konstruktor für leere Listen hat keinen Parameter. Der Konstruktor **Cons** hat ein neues Listenelement und eine bestehende Liste als Parameter. Hier versteckt sich die Rekursion. Um eine neue Liste mit **Cons** zu erzeugen, bedient man sich einer bereits bestehenden Liste.

Selektoren Die Selektoren können wieder auf die einzelnen Bestandteile der Konstruktion zurückgreifen. Der Konstruktor **Cons** hat zwei Parameter. Für **Cons**-Listen werden zwei Selektoren spezifiziert, die jeweils einen dieser beiden Parameter wieder aus der Liste selektieren. Die Namen dieser beiden Selektoren sind traditioneller Weise *head* und *tail*.

- head: **(List)→int**
- tail: **(List)→List**

Der funktionale Zusammenhang von Selektoren und Konstruktoren läßt sich durch folgende Gleichungen spezifizieren:

$$\begin{aligned} head(Cons(x, xs)) &= x \\ tail(Cons(x, xs)) &= xs \end{aligned}$$

Die erste Gleichung ist zu lesen:
wenn nach dem *head* einer Liste gefragt wird, und diese durch den **Cons**-Konstruktor mit einem neuen Element x und einer bestehenden Liste xs erzeugt wurde, dann ist das Ergebnis das Element x.

Analog ist die zweite Gleichung zu lesen:
wenn nach dem *tail* einer Liste gefragt wird, und diese durch den **Cons**-Konstruktor mit einem neuen Element x und einer bestehenden Liste xs erzeugt wurde, dann ist das Ergebnis die Restliste xs.

Wie man sieht gibt es keine Spezifikation von *head* und *tail* für den Fall, dass die fragliche Liste eine leere Liste ist, also eine Liste, die durch **Empty** erzeugt wurde.

[7] Entgegen der Notation in C, in der der Rückgabetyp kurioser Weise vor den Namen der Funktionen geschrieben wird.

Testmethoden Um für Listen Algorithmen umzusetzen, ist es notwendig, unterscheiden zu können, welche Art der beiden Listen vorliegt: die leere Liste oder eine **Cons**-Liste. Hierzu bedarf es noch einer Testfunktion, die mit einem bool'schen Wert als Ergebnis angibt, ob es sich bei der Eingabeliste um die leere Liste handelte oder nicht. Wir wollen diese Testfunktion *isEmpty* nennen. Sie hat folgenden Typ:

- isEmpty: **List→boolean**

Das funktionale Verhalten der Testfunktion läßt sich durch folgende zwei Gleichungen spezifizieren:

$$\begin{aligned} isEmpty(Empty()) &= true \\ isEmpty(Cons(x, xs)) &= false \end{aligned}$$

Somit ist alles spezifiziert, was eine Listenstruktur ausmacht. Listen können konstruiert werden, die Bestandteile einer Liste wieder einzeln selektiert und Listen können nach der Art ihrer Konstruktion unterschieden werden.

Listenalgorithmen Allein diese fünf Funktionen beschreiben den ADT der Listen. Wir können aufgrund dieser Spezifikation Algorithmen für Listen schreiben.

Länge Es läßt sich durch zwei Gleichungen spezifizieren, was die Länge einer Liste ist:

$$\begin{aligned} length(Empty()) &= 0 \\ length(Cons(x, xs)) &= 1 + length(xs) \end{aligned}$$

Mit Hilfe dieser Gleichungen läßt sich jetzt schrittweise die Berechnung einer Listenlänge auf Listen durchführen. Hierzu benutzen wir die Gleichungen als Ersetzungsregeln. Wenn ein Unterausdruck in der Form der linken Seite einer Gleichung gefunden wird, so kann diese durch die entsprechende rechte Seite ersetzt werden. Man spricht bei so einem Ersetzungsschritt von einem Reduktionsschritt.

Beispiel 4.10.1 *Wir errechnen in diesem Beispiel die Länge einer Liste, indem wir die obigen Gleichungen zum Reduzieren auf die Liste anwenden:*

$$\begin{aligned} & length(\mathbf{Cons}(a, \mathbf{Cons}(b, \mathbf{Cons}(c, \mathbf{Empty}())))) \\ \rightarrow\ & 1 + length(\mathbf{Cons}(b, \mathbf{Cons}(c, \mathbf{Empty}()))) \\ \rightarrow\ & 1 + (1 + length(\mathbf{Cons}(c, \mathbf{Empty}()))) \\ \rightarrow\ & 1 + (1 + (1 + length(\mathbf{Empty}()))) \\ \rightarrow\ & 1 + (1 + (1 + 0)) \\ \rightarrow\ & 1 + (1 + 1) \\ \rightarrow\ & 1 + 2 \\ \rightarrow\ & 3 \end{aligned}$$

Letztes Listenelement Wir können mit einfachen Gleichungen spezifizieren, was wir unter dem letzten Element einer Liste verstehen.

$$\begin{aligned} last(Cons(x, Empty())) &= x \\ last(Cons(x, xs))) &= last(xs) \end{aligned}$$

Auch die Funktion `last` können wir von Hand auf einer Beispielliste einmal per Reduktion ausprobieren:

$$\begin{aligned} & last(\mathbf{Cons}(a, \mathbf{Cons}(b, \mathbf{Cons}(c, \mathbf{Empty}())))) \\ \rightarrow\ & last(\mathbf{Cons}(b, \mathbf{Cons}(c, \mathbf{Empty}()))) \\ \rightarrow\ & last(\mathbf{Cons}(c, \mathbf{Empty}())) \\ \rightarrow\ & c \end{aligned}$$

Listenkonkatenation Die folgenden Gleichungen spezifizieren, wie zwei Listen aneinandergehängt werden:

$$\begin{aligned} concat(Empty(), ys) &= ys \\ concat(Cons(x, xs), ys) &= Cons(x, concat(xs, ys)) \end{aligned}$$

Auch diese Funktion läßt sich beispielhaft mit der Reduktion einmal durchrechnen:

$$\begin{aligned} & concat(\mathbf{Cons}(i, \mathbf{Cons}(j, \mathbf{Empty}())), \mathbf{Cons}(a, \mathbf{Cons}(b, \mathbf{Cons}(c, \mathbf{Empty}())))) \\ \rightarrow\ & \mathbf{Cons}(i, concat(\mathbf{Cons}(j, \mathbf{Empty}()), \mathbf{Cons}(a, \mathbf{Cons}(b, \mathbf{Cons}(c, \mathbf{Empty}()))))) \\ \rightarrow\ & \mathbf{Cons}(i, \mathbf{Cons}(j, concat(\mathbf{Empty}(), \mathbf{Cons}(a, \mathbf{Cons}(b, \mathbf{Cons}(c, \mathbf{Empty}())))))) \\ \rightarrow\ & \mathbf{Cons}(i, \mathbf{Cons}(j, \mathbf{Cons}(a, \mathbf{Cons}(b, \mathbf{Cons}(c, \mathbf{Empty}()))))) \end{aligned}$$

Schachtel- und Zeiger-Darstellung Listen lassen sich auch sehr schön graphisch visualisieren. Hierzu wird jede Liste durch eine Schachtel mit zwei Feldern dargestellt. Von diesen beiden Feldern gehen Pfeile aus. Der erste Pfeil zeigt auf das erste Element der Liste, dem `head`, der zweite Pfeil zeigt auf die Schachtel, die für den Restliste steht dem *tail*. Wenn eine Liste leer ist, so gehen keine Pfeile von der Schachtel aus, die sie repräsentiert.

Die Liste *Cons(a,Cons(b,Cons(c,Empty())))* hat somit die Schachtel- und Zeiger- Darstellung aus Abbildung 4.1.

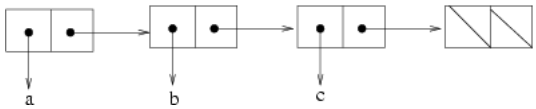


Abbildung 4.1: Schachtel Zeiger Darstellung einer dreielementigen Liste.

In der Schachtel- und Zeiger-Darstellung läßt sich sehr gut nachverfolgen, wie bestimmte Algorithmen auf Listen dynamisch arbeiten. Wir können die schrittweise Reduktion der Methode `concat` in der Schachtel- und Zeiger-Darstellung gut nachvollziehen:

Abbildung 4.2 zeigt die Ausgangssituation. Zwei Listen sind dargestellt. Von einer Schachtel, die wir als die Schachtel der Funktionsanwendung von `concat` markiert haben, gehen zwei Zeiger

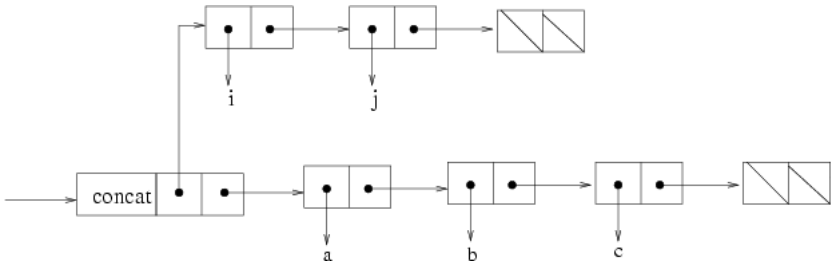


Abbildung 4.2: Schachtel Zeiger Darstellung der Funktionsanwendung von concat auf zwei Listen.

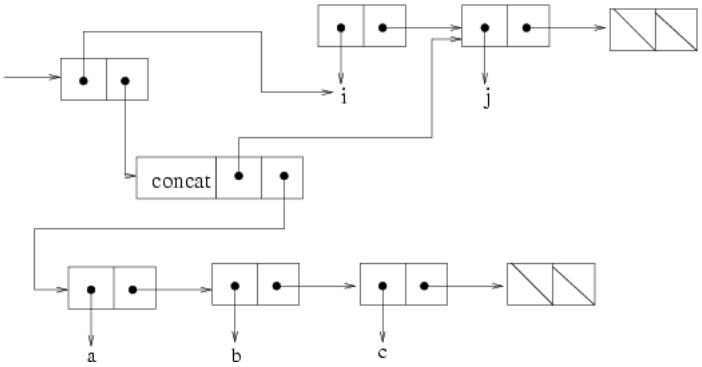


Abbildung 4.3: Schachtel-Zeiger-Darstellung nach dem zweiten Reduktionsschritt.

aus. Der erste auf das erste Argument, der zweite auf das zweite Argument der Funktionsanwendung.

Abbildung 4.3 zeigt die Situation, nachdem die Funktion `concat` einmal reduziert wurde. Ein neuer Listenknoten wurde erzeugt. Dieser zeigt auf das erste Element der ursprünglich ersten Argumentliste. Der zweite zeigt auf den rekursiven Aufruf der Funktion `concat`, diesmal mit der Schwanzliste des ursprünglich ersten Arguments.

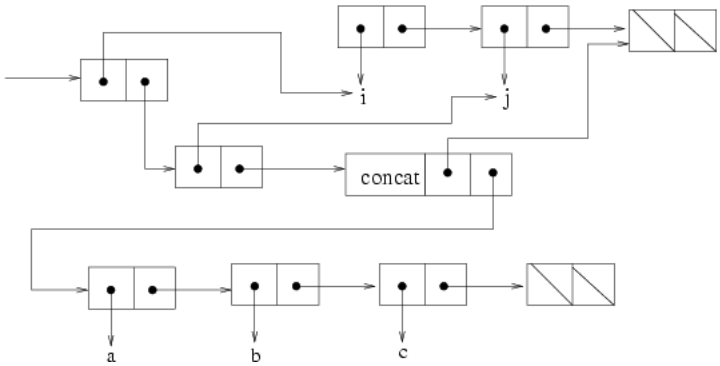


Abbildung 4.4: Schachtel-Zeiger-Darstellung nach dem zweiten Reduktionsschritt.

Abbildung 4.4 zeigt die Situation nach dem zweiten Reduktionsschritt. Ein weiterer neuer Listenknoten ist entstanden und ein neuer Knoten für den rekursiven Aufruf ist entstanden.

Abbildung 4.5 zeigt die endgültige Situation. Der letzte rekursive Aufruf von `concat` hatte als

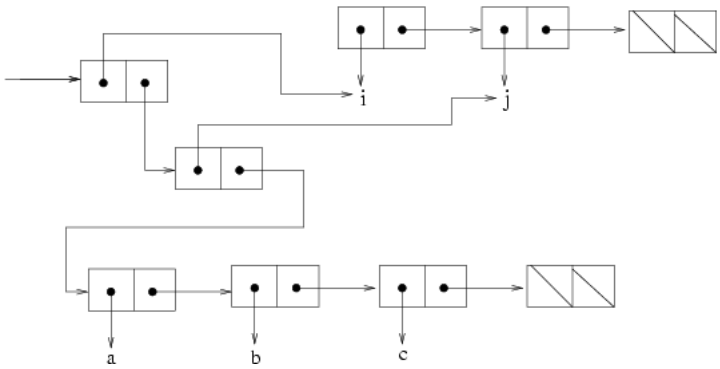


Abbildung 4.5: Schachtel Zeiger Darstellung des Ergebnisses nach der Reduktion.

erstes Argument eine leere Liste. Deshalb wurde kein neuer Listenknoten erzeugt, sondern lediglich der Knoten für die Funktionsanwendung gelöscht. Man beachte, daß die beiden ursprünglichen Listen noch vollständig erhalten sind. Sie wurden nicht gelöscht. Die erste Argumentliste wurde quasi kopiert. Die zweite Argumentliste teilen sich gewisser Maßen die neue Ergebnisliste der Funktionsanwendung und die zweite ursprüngliche Argumentliste.

Implementierung

Es ist nun endlich an der Zeit die Spezifikation der Liste auch in C-Code zu gießen. Tatsächlich ist diese Recht einfach und die Gleichungen lassen sich ziemlich eins-zu-eins in entsprechende Zeile C-Code umsetzen.

Wir werden zwei Implementierungen vornehmen. Zunächst werden wir nur Listen implementieren, in denen als Elemente `int`-Zahlen gespeichert werden können. Anschließend werden wir daraus eine generische Implementierung ableiten, in der beliebige Elementtypen gespeichert werden können.

Beginnen wir damit, in der Kopfdatei zu definieren, als welchen Datentypen unsere liste gespeichert werden sollen.

Zunächst definieren wir uns einen Typ für Wahrheitswerte:

List.h

```
1 #ifndef LIST_
2 #define LIST_
3
4 typedef enum {false,true} bool;
```

Für Listen wird eine Struktur definiert. Nach unserer Spezifikation muß eine Liste zwei Bestandteile speichern können.

Zusätzlich sehen wir noch ein Attribut vor, das kennzeichnet, ob die Liste eine leere Liste ist.

List.h

```
5 struct Liste{
6   int head;
7   struct Liste* tail;
8 };
```

Um ein wenig lesbarer über Listen sprechen zu können, führen wir ein Typsynonym für die Struktur ein. Damit umgehen wir, dass wir immer das Wort `struct` vor dem Typnamen schreiben müssen:

List.h

```
9  typedef struct Liste* List;
```

Die Selektoren der Listen haben wir bereits durch die Felder der Struktur `Liste` ausgedrückt. Ebenso dient das Feld `isEmpty` als Test.

Wir müssen nach unserer Spezifikation noch zwei Konstruktorfunktionen vorsehen. Hierzu definieren wir die Funktionen `cons` und `nil`, wobei `nil` für die Konstruktion leerer Liste stehen soll.

List.h

```
10 List nil();
11 List cons(int x,List xs);
12
13 bool isEmpty(List xs);
14
```

Wir hantieren eifrig mit dynamischen daten. Daher ist es notwendig eine Funktion zum Löschen der Listen aus dem Speicher vorzusehen. Diese tauchte in der Spezifikation nicht auf. Sie hat nichts mit der Funktionalität von Listen zu tun, sondern ist eine technisches Funktion für die explizite Speicherfreigabe.

List.h

```
15 void delete(List xs);
```

Soweit die Kopfdatei.

List.h

```
16 #endif /*LIST_*/
17
```

Zur Implementierung der Kopfdatei sind drei Funktionen umzusetzen. zunächst die beiden Konstruktoren. Beide müssen zunächst Speicher allokieren, in dem das neue Listenobjekt gespeichert werden kann. Daher sei zunächst eine eigene kleine Funktion definiert, die diesen Speicherplatz allokiert:

List.c

```
1  #include "List.h"
2  #include <stdlib.h>
3  #include <stdio.h>
4
5  List newList(){return (List)malloc(sizeof (struct Liste));}
```

Im Konstruktor für leere Listen ist ein neues Listenobjekt zu erzeugen, und dann darin zu vermerken, dass es sich um eine leere Liste handelt:

List.c
```
List nil(){
  List result=newList();
  result->tail=NULL;
  return result;
};
```

Der Konstruktor `cons` hat zwei Parameter, die in den entsprechenden Feldern des neu erzeugten Listenobjekts abzuspeichern sind:

List.c
```
List cons(int x, List xs){
  List result=newList();
  result->head=x;
  result->tail=xs;
  return result;
};
```

List.c
```
bool isEmpty(List xs){
  return xs->tail==NULL;
};
```

Verbleibt nur noch das Löschen einer Liste. Hierzu ist nicht nur das aktuelle Listenobjekt zu löschen, sondern, sofern es einen `tail` gibt, also für nichtleere Liste, ist zunächst der `tail` im Speicher wieder freizugeben. Erst anschließend kann das Listenobjekt selbst im Speicher wieder freigegeben werden. Es ergibt sich folgende rekursive Funktion.

List.c
```
void delete(List xs){
  if (!isEmpty(xs)) delete(xs->tail);
  free(xs);
}

```

Zeit, ein paar erste Tests mit unserer listenimplementierung durchzuführen. Hierzu sei eine Liste aus vier Zahlen erzeugt und drei dieser Zahlen aus der Liste wieder zugegriffen:

TestList.c
```
#include "List.h"
#include <stdio.h>

int main(){
  List xs=cons(1,cons(2,cons(3,cons(4,nil()))));
  printf("Das erste Element: %i\n",xs->head);
  printf("Das zweite Element: %i\n",xs->tail->head);
  printf("Das dritte Element: %i\n",xs->tail->tail->head);
  delete(xs);
  return 0;
}

```

Nun können wir Algorithmen für Listen implementieren. Es sollen vorerst sechs Algorithmen implementiert werden, darunter die beiden Funktionen, die wir im vorangegangenen Abschnitt bereits implementiert haben: der Algorithmus zur Berechnung der Länge einer Liste und der zum Aneinanderhängen zweier Liste[8].

Hier nun die Kopfdatei, in der die Signaturen der sechs zu implementierenden Algorithmen definiert sind:

ListUtil.h

```
1  #ifndef LIST_UTIL_
2  #define LIST_UTIL_
3  #include "List.h"
4
5  int length(List xs);
6  int sum(List xs);
7  double avg(List xs);
8
9  int last(List xs);
10
11 List reverse(List xs);
12
13 List copy(List xs);
14 List append(List xs,List ys);
15
16 List map(int f(int),List xs);
17
18 #endif /*LIST_UTIL_*/
19
```

Beginnen wir mit der Implementierung der Längenfunktion. Sie war durch zwei Gleichungen spezifiziert. Leere Listen haben die Länge 0, nichtleere Listen sind um eins Länger als der Listentail. Dieses mündet direkt in folgende Implementierung:

ListUtil.c

```
1  #include "ListUtil.h"
2
3  int length(List xs){
4    if (isEmpty(xs)) return 0;
5    return 1+length(xs->tail);
6  }
```

Die Implementierung bedient sich der Rekursion. Da unsere Listen eine rekursiv definierte Datenstruktur sind, lassen sich die meisten Algorithmen sehr schön rekursiv implementieren. Allerdings sind iterative Lösungen zumindest leichter zu effizienten Code kompilierbar.

ListUtil.c

```
1  int lengthIteration(List xs){
2    int result=0;
3    while (!isEmpty(xs)){
```

[8] Die in der Spezifikation `concat` hieß, hier aber unter den Namen `append` implementiert ist.

```
4     result=result+1;
5     xs=xs->tail;
6   }
7   return result;
8 }
```

Da wir bisher nur Zahlen als Listenelemente Speichern können wir diese natürlich auch auch aufsummieren. Auch die Summenfunktion läßt sich in zwei Gleichungen spezifizieren:

$$\begin{aligned} sum(Empty()) &= 0 \\ sum(Cons(x, xs)) &= x + sum(xs) \end{aligned}$$

Diese Gleichungen münden wieder direkt in die entsprechende Implementierung, in der für jede der beiden Gleichungen genau eine Zeile benötigt wird:

ListUtil.c

```
1 int sumRecursion(List xs){
2   if (isEmpty(xs)) return 0;
3   return xs->head+length(xs->tail);
4 }
```

Oder aber wir können die Spezifikation auch iterativ implementieren.

ListUtil.c

```
5 int sum(List xs){
6   int result=0;
7   while (!isEmpty(xs)){
8     result=result+xs->head;
9     xs=xs->tail;
10  }
11  return result;
12 }
```

Wenn die Summe der einzelnen Elemente bekannt ist und die Länge der Liste, dann läßt sich aus beiden direkt der Durchschnittswert der Elemente berechnen.

ListUtil.c

```
13 double avg(List xs){
14   if (isEmpty(xs)) return 0;
15   double su = sum(xs);
16   double le = length(xs);
17   return su/le;
18 }
```

Auch die Funktion zur Selektion des letzten Elements einer Liste haben wir bereits in zwei Gleichungen spezifiziert. Auch diese zwei Gleichungen lassen sich direkt in entsprechenden Code umsetzen. Allerdings fügen wir als dritten Fall hinzu, dass für leere Listen die Zahl 0 als letztes Element zurcükgegeben wird.

ListUtil.c

```
19 int last(List xs){
20   if (isEmpty(xs)) return 0;
21   if (isEmpty(xs->tail)) return xs->head;
22   return last(xs->tail);
23 }
```

Eine naheliegende Funktion für Listen, ist ihre Reihenfolge umzudrehen. Hierzu wird das erste Element zum letzen gemacht. Auch diese Funktion läßt sich in zwei Gleichungen spezifizieren. In der Spezifikation wird die bereits spezifizierte Funktion `concat`.

$$\begin{aligned} reverse(Empty()) &= Empty() \\ reverse(Cons(x, xs)) &= concat(reverse(xs), Cons(x, Empty())) \end{aligned}$$

Wenn wir einmal davon ausgehen, dass die Funktion `concat` unter den Namen `append` weiter unten implementiert wird, so lassen sich auch diese Gleichungen direkt in zwei Zeilen umsetzen:

ListUtil.c

```
24 List reverseRecursive(List xs){
25   if (isEmpty(xs)) return nil();
26   return append(reverse(xs->tail),cons(xs->head,nil()));
27 }
```

Diese Implementierung ist allerdings hoffnungslos ineffizient. Im jeden rekursiven Aufruf wirds die Funktion `append` aufgerufen, die ihrerseits wieder rekursiv ist. Damit haben wir eine ineinander geschachtelte Rekursion. Der Aufwand hierfür wird quadratisch.

Die Funktion `reverse` läßt sich zum vergleich recht einfach und effizient iterative implementieren. Es wird eine Ergebnisliste in einer Schleife stückweise zusammengesetzt, indem immer das erste Element der durchlaufenden Liste vorn an die Ergebnisliste angehängt wird.

ListUtil.c

```
28 List reverse(List xs){
29   List result=nil();
30   while (!isEmpty(xs)){
31     result=cons(xs->head,result);
32     xs=xs->tail;
33   }
34   return result;
35 }
```

Diese Implementierung ist wesentlich effizienter. Sie durchläuft genau einmal die Liste, die umzudrehen ist. Nur ist streng genommen noch formal zu beweisen, dass die Funktion `reverse` das gleiche Ergebnis hat, wie die Funktion `reverseRecursive`. Ein solcher Beweis ist allerdings formal sehr schwierig. In der Praxis beschränkt man sich daher zumeist darauf, sich durch ein paar Testbeispiele zu versichern, dass die Implementierung das gewünschte leistet.

Als nächstes sei eine Funktion implementiert, die eigentlich keine wirkliche neue Berechnung vornimmt, sondern eine eins-zu-eins Kopie einer Liste erstellt. Hierzu wird für eine leere Liste eine neue `nil`-Liste erzeugt und für nichtleere Liste eine neue `cons`-Liste.

ListUtil.c
```
36 List copy(List xs){
37   if (isEmpty(xs)) return nil();
38   return cons(xs->head,copy(xs->tail));
39 }
```

Nun endlich soll die Implementierung der Funktion folgen, die aus zwei Listen eine neue Liste erzeugt. Die neue Liste besteht erst aus den Elementen des ersten Parameters, dann aus den Elementen des zweiten Parameters:

ListUtil.c
```
40 List append(List xs,List ys){
41   if (isEmpty(xs)) return copy(ys);
42   return cons(xs->head,append(xs->tail,ys));
43 }
```

Im Falle einer leeren Liste als ersten Parameter begnügen wir uns nicht damit, die Liste des zweiten Parameters zurückzugeben, sondern erzeugen eine Kopie von diesem. Die Entscheidung hierfür liegt in der Tatsache, dass es sonst sehr schwer wird darüber Buch zu führen, welche Listen noch Bestandteil von Teillisten sind, und welche im Speicher nach Gebrauch wieder frei gegeben werden können.

Zu guter letzt soll noch eine Funktion höherer Ordnung folgen. Die Funktion `map` soll eine neue liste erzeugen, in dem auf jedes Listenelement eine Funktion übergeben wird. Auch die Funktion `map` ist schnell durch zwei Gleichungen spezifiziert:

$$\begin{aligned} map(f, Empty()) &= Empty() \\ map(f, Cons(x, xs)) &= Cons(f(x), map(f, xs)) \end{aligned}$$

Und diese zwei Gleichungen lassen sich wieder direkt in entsprechenden Coe umsetzen:

ListUtil.c
```
44 List map(int f (int),List xs){
45   if (isEmpty(xs)) return nil();
46   return cons(f(xs->head),map(f,xs->tail));
47 }
48
```

Nun sollen die Funktionen endlich auch einmal ausgetestet werden:

TestListUtil.c
```
1 #include "ListUtil.h"
2 #include <stdio.h>
3
4 int addFive(int x){return x+5;}
5
6 int main(){
7   List xs=cons(1,cons(2,cons(3,cons(4,nil()))));
8   printf("Das letzte Element von xs ist: %i\n",last(xs));
9   printf("Die Laenge von xs ist: %i\n",length(xs));
```

```
10    printf("Die Summe von xs ist: %i\n",sum(xs));
11    printf("Der Durchschnitt von xs ist: %f\n",avg(xs));
12
13    List xsxs=append(xs,xs);
14    printf("Die Laenge von xsxs ist: %i\n",length(xsxs));
15    printf("Die Summe von xsxs ist: %i\n",sum(xsxs));
16    printf("Der Durchschnitt von xsxs ist: %f\n",avg(xsxs));
17
18    delete(xsxs);
19
20    List sx=reverse(xs);
21    printf("Das erste Element von sx: %i\n",sx->head);
22    delete(sx);
23
24    List xs5=map(addFive,xs);
25    printf("Die Summe von xs5 ist: %i\n",sum(xs5));
26
27    delete(xs5);
28    delete(xs);
29
30    return 0;
31  }
32
```

generischer Zeigerliste

Die Listen im letzen Abschnitt haben als Listenelemente ganze Zahlen enthalten. Wollen wir andere Daten in Listen speichern, so wird eine neue Listenimplementierung notwendig. Das ist etwas schade, denn die meisten Algorithmen für listen sind unabhängig von den Elementtypen der Listen. Zur Berechnung der Länge einer Liste, ist ziemlich egal, was für Elemente in den einzelnen Listenzellen im Feld `head` gespeichert sind. Es wäre schön, wenn wir eine Listenstruktur implementieren könnten, die in der Lage ist unterschiedlichste Daten als Elemente anzuspeichern. Hierzu haben wir schon früher einen Trick kennengelernt: die Void-Zeiger. Statt jetzt eine Liste von ganzen Zahlen zu definieren, können wir eine Liste von Void-Zeigern definieren. Ein Void-Zeiger kann dann auf beliebige Arten von daten zeigen.

Wir können die Headerdatei für unsere Listen von ganzen Zahlen weitgehendst übernehmen, indem wir einfach überall wo der Typ `int` eingesetzt war, den Typ `void*` schreiben. Da wir den Typ `void*` benutzen wollen um beliebige Objekte darin speichern zu können, führen wir wie bereits früher dazu das Typsynonym `Object` ein.

Damit erhalten wir die folgende Header-Datei:

PList.h

```
1  #ifndef PLIST_
2  #define PLIST_
3
4  typedef enum {false,true} bool;
5  typedef void* Object;
6
7  struct PListe{
```

```
8   Object head;
9   struct PListe* tail;
10 };
11
12 typedef struct PListe* PList;
13
14 PList nil();
15 PList cons(Object x,PList xs);
16
17 bool isEmpty(PList xs);
18
19 void delete(PList xs);
20 #endif /*PLIST_*/
21
```

Es stellt sich heraus, dass wir die Implementierung der Listen von ganzen Zahlen fast wörtlich übernehmen können, um unsere allgemeinere Listenimplementierung zu bekommen:

PList.c

```
1  #include "PList.h"
2  #include <stdlib.h>
3  #include <stdio.h>
4
5  PList nil(){
6    PList result=malloc (sizeof (struct PListe));
7    result->tail=NULL;
8    result->head=NULL;
9    return result;
10 };
11
12 PList cons(Object x, PList xs){
13   PList result=malloc (sizeof (struct PListe));
14   result->head=x;
15   result->tail=xs;
16   return result;
17 };
18
19 bool isEmpty(PList xs){
20   return xs->tail==NULL;
21 };
22
23 void delete(PList xs){
24   if (!isEmpty(xs)) delete(xs->tail);
25   free(xs);
26 }
27
```

Bevor wir jetzt die einzelnen Algorithmen für Listen implementieren, soll ein kleiner Test uns davon, dass wir mit ihr tatsächlich in der Lage sind, beliebige Objekttypen zu speichern.

Es sollen zunächst Objekte eines Punkttypens gespeichert werden. Hierzu sei die entsprechende Struktur zur Darstellung von Punkten im zweidimensionalen Raum definiert. Mit einer kleinen Hilfsfunktion sollen solche Punktobjekte auf der Kummandozeile ausgegeben werden:

Punkt.h

```
1  #ifndef PUNKT__H_
2  #define PUNKT__H_
3  typedef struct {
4    int x;
5    int y;
6  } Punkt;
7
8  void printPunkt(Punkt* p);
9  #endif
```

Punkt.c

```
1  #include "Punkt.h"
2  #include <stdio.h>
3  void printPunkt(Punkt* p){
4    printf("(%i,%i)",p->x,p->y);
5  }
```

In der Testfunktion legen wir ein paar Punkte an, und bauen aus diesen eine Liste.

TestPList.c

```
6  #include "PList.h"
7  #include "Punkt.h"
8
9  #include <stdio.h>
10
11 int main(){
12   Punkt p1 = {0,0};
13   Punkt p2 = {0,2};
14   Punkt p3 = {2,2};
15   Punkt p4 = {2,0};
16
17   PList polygon=cons(&p1,cons(&p2,cons(&p3,cons(&p4,nil()))));
```

In gewohnter Weise lässt sich auf diese Liste mit `head` und `tail` zugreifen:

TestPList.c

```
18   printPunkt((Punkt*)polygon->tail->tail->head);
19
20   delete(polygon);
21
```

Nun wollen wir die Listenimplementierung benutzen, um auch weiterhin ganze Zahlen zu speichern. Jetzt allerdings können wir nur Zeiger auf ganze Zahlen speichern.

TestPList.c

```
22
23
24    int x1=1;
25    int x2=2;
26    int x3=3;
27    int x4=4;
28    PList xs=cons(&x1,cons(&x2,cons(&x3,cons(&x4,nil()))));
29
30    printf("Das erste Element: %i\n",*(int*)xs->head);
31    printf("Das zweite Element: %i\n",*(int*)xs->tail->head);
32    printf("Das dritte Element: %i\n",*(int*)xs->tail->tail->head);
33    delete(xs);
34    return 0;
35  }
36
```

Damit ließen sich also mit einer Listenimplementierung Listen ganz unterschiedlichen Inhalts erzeugen.

Nun wollen wir auch die unterschiedlichen Algorithmen auf Listen implementieren:

PListUtil.h

```
1   #ifndef PLIST_UTIL_
2   #define PLIST_UTIL_
3   #include "PList.h"
4
5   int length(PList xs);
6   PList copy(PList xs);
7   PList append(PList xs,PList ys);
8   PList reverse(PList xs);
9   PList map(Object f(Object),PList xs);
10  void forEach(void f(Object),PList xs);
11
12  #endif /*LIST_UTIL_*/
13
```

Tatsächlich ist für die meisten dieser Funktionen gegenüber der bisherigen Implementierung nichts zu ändern.

PListUtil.c

```
1   #include "PListUtil.h"
2
3   int length(PList xs){
4     if (isEmpty(xs)) return 0;
5     return 1+length(xs->tail);
6   }
7
8   PList copy(PList xs){
9     if (isEmpty(xs)) return nil();
```

```
10    return cons(xs->head,copy(xs->tail));
11  }
12
13  PList append(PList xs,PList ys){
14    if (isEmpty(xs)) return copy(ys);
15    return cons(xs->head,append(xs->tail,ys));
16  }
17
18  PList reverse(PList xs){
19    PList result=nil();
20    while (!isEmpty(xs)){
21      result=cons(xs->head,result);
22      xs=xs->tail;
23    }
24    return result;
25  }
26
27  PList map(Object f (Object),PList xs){
28    if (isEmpty(xs)) return nil();
29    return cons(f(xs->head),map(f,xs->tail));
30  }
31
32  void forEach(void f (Object),PList xs){
33    for (;!isEmpty(xs);xs=xs->tail) f(xs->head);
34  }
35
```

Wir bedienen uns den Punkten im zweidimensionalen Raum, um ein paar Tests mit diesen Funktionen durchzuführen

TestPListUtil.c

```
1   #include "PListUtil.h"
2   #include "Punkt.h"
3
4   #include <stdio.h>
5
6   void move(Punkt* this){
7     this->x=2*this->x;
8     this->y=2*this->y;
9   }
10
11  int main(){
12    Punkt p1 = {0,0};
13    Punkt p2 = {0,2};
14    Punkt p3 = {2,2};
15    Punkt p4 = {2,0};
16
17    PList polygon=cons(&p1,cons(&p2,cons(&p3,cons(&p4,nil()))));
18
19    printf("length(polygon)=$i\n",length(polygon));
```

```
20    printf("polygon: [");
21    forEach(printPunkt,polygon);
22    printf("]\n");
23
24    PList pol2 = append(polygon,polygon);
25    printf("append(polygon,polygon): [");
26    forEach(printPunkt,pol2);
27    printf("]\n");
28
29    PList pol3 = reverse(pol2);
30    printf("reverse(pol2): [");
31    forEach(printPunkt,pol3);
32    printf("]\n");
33
34    map(move,pol3);
35    printf("map(move,pol3): [");
36    forEach(printPunkt,pol3);
37    printf("]\n");
38
39    delete(polygon);
40    delete(pol2);
41    delete(pol3);
42
43    return 0;
44  }
45
46
```

Aufgabe 40 (3 Punkte) Erweitern Sie das Bibliothek `PList` um folgende weitere Funktionen und testen Sie Ihre Implementierungen an ein paar Beispielen aus:

a) `Object last(PList this );`
Die Funktion sei spezifiziert durch folgende Gleichung:

$$\begin{aligned} last(Cons(x, Nil())) &= x \\ last(Cons(x, xs))) &= last(xs) \end{aligned}$$

b) `Object get(PList this,unsigned int i );`
Die Funktion sei spezifiziert durch folgende Gleichung:

$$\begin{aligned} get(Cons(x, xs), 0) &= x \\ get(Cons(x, xs)), i) &= get(xs, i-1) \end{aligned}$$

c) `PList take(PList this,unsigned int i);`
Die Funktion sei spezifiziert durch folgende Gleichung:

$$\begin{aligned} take(xs, 0) &= Nil() \\ take(Nil(), i) &= Nil() \\ take(Cons(x, xs), i)) &= Cons(x, take(xs, i-1)) \end{aligned}$$

d) `PList drop(PList this,unsigned int i);`
Die Funktion sei spezifiziert durch folgende Gleichung:

$$\begin{aligned} drop(Nil(), i) &= Nil() \\ drop(xs, 0) &= xs \\ drop(Cons(x, xs), i) &= drop(xs, i-1) \end{aligned}$$

e) `PList oddElements(PList this);`
Die Funktion sei spezifiziert durch folgende Gleichung:

$$\begin{aligned} oddElements(Nil()) &= Nil() \\ oddElements(Cons(x, Nil())) &= Cons(x, Nil()) \\ oddElements(Cons(x, Cons(y, ys))) &= Cons(x, oddElements(ys)) \end{aligned}$$

f) `PList filter(PList this,bool predicate(Object));`
Die Funktion sei spezifiziert durch folgende Gleichung:

$$\begin{aligned} filter(Nil(), pred) &= Nil() \\ filter(Cons(x, xs), pred) &= \begin{cases} Cons(x, filter(xs, p)) & \text{für pred}(x) \\ filter(xs, p) & \text{für } \neg\text{pred}(x) \end{cases} \end{aligned}$$

Lösung

PListUtil2.h

```
#ifndef PLIST_UTIL2_
#define PLIST_UTIL2_
#include "PList.h"

Object last(PList this );
Object get(PList this,unsigned int i);
PList take(PList this,unsigned int i);
PList drop(PList this,unsigned int i);
PList oddElements(PList this);
PList filter(PList this,bool predicate(Object));

#endif /*LIST_UTIL2_*/

```

PListUtil2.c

```
#include "PList.h"
#include "PListUtil.h"
#include "PListUtil2.h"

#include <stdio.h>

Object last(PList this ){
 if (isEmpty(this->tail)) return this->head;
 return last(this->tail);
}
```

```
11
12 Object get(PList this,unsigned int i){
13   if (i==0) return this->head;
14   return get(this->tail,i-1);
15 }
16
17 PList take(PList this,unsigned int i){
18   if (isEmpty(this)||i==0) return nil();
19   return cons(this->head,take(this->tail,i-1));
20 }
21
22 PList drop(PList this,unsigned int i){
23   if (isEmpty(this)) return copy(this);
24   if (i==0) return copy(this);
25   return drop(this->tail,i-1);
26 }
27
28 PList oddElements(PList this){
29   if (isEmpty(this)) return nil();
30   if (isEmpty(this->tail)) return cons(this->head,nil());
31   return cons(this->head,oddElements(this->tail->tail));
32 }
33
34 PList filter(PList this,bool predicate(Object)){
35   if (isEmpty(this)) return nil();
36   if (predicate(this->head))
37      return cons(this->head,filter(this->tail,predicate));
38   return filter(this->tail,predicate);
39 }
40
```

PListUtil2Test.c

```
1 #include "PList.h"
2 #include "Punkt.h"
3 #include "PListUtil.h"
4 #include "PListUtil2.h"
5
6 #include <stdio.h>
7
8 bool nichtUrsprung(Punkt* p){
9   return p-> x != 0 || p ->y !=0;
10 }
11
12 int main(){
13   Punkt p1 = {0,0};
14   Punkt p2 = {0,2};
15   Punkt p3 = {2,2};
16   Punkt p4 = {2,0};
17
```

```
18    PList polygon=cons(&p1,cons(&p2,cons(&p3,cons(&p4,nil()))));
19
20    printf("polygon: [");
21    forEach(printPunkt,polygon);
22    printf("]\n");
23
24    printf("last(polygon)=");
25    printPunkt((Punkt*) last(polygon));
26    printf("\n");
27    printf("get(polygon,2)=");
28    printPunkt((Punkt*)get(polygon,2));
29    printf("\n");
30    PList xs = drop(polygon,2);
31
32    printf("drop(polygon,2): [");
33    forEach(printPunkt,xs);
34    printf("]\n");
35
36    delete(xs);
37
38    xs = take(polygon,2);
39
40    printf("take(polygon,2): [");
41    forEach(printPunkt,xs);
42    printf("]\n");
43
44    delete(xs);
45
46    xs = oddElements(polygon);
47
48    printf("oddElements(polygon): [");
49    forEach(printPunkt,xs);
50    printf("]\n");
51
52    delete(xs);
53
54    xs = filter(polygon,nichtUrsprung);
55
56    printf("filter(polygon,nichtUrsprung): [");
57    forEach(printPunkt,xs);
58    printf("]\n");
59    delete(xs);
60    delete(polygon);
61
62    return 0;
63  }
64
```

4.11 Binäre Suchbäume

Im letzten Abschnitt haben wir als erste rekursive Datenstruktur Listen ausgiebig behandelt. Exemplarisch soll in diesem Abschnitt eine weitere häufig gebrauchte Datenstruktur aus der Informatik implementiert werden. Es sollen Bäume realisiert werden. Dabei wollen wir uns auf eine bestimmte Art von Bäumen beschränken: den Binärbäumen. Allgemein sind Bäume so aufgebaut, dass ein Baumknoten Teilbäume als Kinder hat. Bei Binärbäumen hat jeder Baumknoten maximal 2 Kinder: einen linken und einen rechten Teilbaum als Kind.

Die Knoten eines Baumes sollen mit Daten markiert sein. In unserem Fall wollen wir uns auf Strings als Knotenmarkierungen beschränken. Mit diesen Vorgaben lassen sich Bäume kurz und knapp in C umsetzen:

BinTree.c

```
1  #include <string.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  struct B {
6    char* name;
7    struct B* links;
8    struct B* rechts;
9  };
10
11 typedef struct B* Baum;
```

Ein genauer Blick auf diese Datenstruktur zeigt, dass sie in ihrer Art unserer Listenstruktur sehr ähnelt. Das Feld, dass hier den Namen `name` trägt, entspricht den Feld `head` der Listen, das Feld mit dem Name `links` entspricht dem Feld `tail` unserer Listen. Zusätzlich gibt es ein weiteres Feld, das Feld `rechts`, das man quasi als zweiten `tail` verstehen kann. Im Grunde genommen waren Listen auch eine Art von Bäumen, man könnte sie als Unärbäume bezeichnen, weil jeder Listenknoten maximal ein Kind hat, seinen `tail`.

Als nächstes soll eine Konstruktorfunktion geschrieben werden, die die einfachste Form binärer Bäume konstruiert, nämlich Blätter. Blätter sind Bäume, die kein Kind haben, was in unserer Implementierung bedeutet, dass rechtes und linkes Kind beide mit `NULL` belegt werden.

BinTree.c

```
12 Baum newBlatt(char* n){
13   Baum this = (Baum)malloc(sizeof(struct B));
14   this->name=n;
15   this->links=NULL;
16   this->rechts=NULL;
17   return this;
18 }
```

Binärbäume können dazu genutzt werden, um Objekte effizient gemäß einer Ordnung zu speichern und effizient nach ihnen unter Benutzung dieser Ordnung wieder zu suchen. Die Knotenmarkierungen unserer Bäume sind Strings, für die sich die lexikographische Ordnung anbietet.

Binäre Bäume heißen binäre Suchbäume, wenn für jeden Baumknoten gilt:

- die Knotenmarkierung seines linken Teilbaums ist kleiner oder gleich als die eigene Knotenmarkierung.
- die Knotenmarkierung des rechten Teilbaums ist größer als die eigene Knotenmarkierung.

Um einen binären Suchbaum zu erhalten, müssen Bäume mit mehr als einen Knoten kontrolliert konstruiert werden. Hierzu ist es sinnvoll eine Funktion anzubieten, die eine neue Markierung in einen bestehenden Baum einfügt. Diese Einfügefunktion muß sicherstellen, dass, wenn der ursprüngliche Baum ein binärer Suhbaum war, dann der geänderte Baum auch wieder ein binärer Suchbaum ist.

Algorithmisch kann das wie folgt erreicht werden:

Gegeben seien ein Baum `t` und ein Element `o`.

- Ist `o` kleiner oder gleich der Markierung von `t`, so muß `o` ins linke Kind von `t` eingefügt werden.
- Ansonsten muß `o` ins rechte Kind von `t` eingefügt werden.

Hat `t` kein linkes (bzw. rechtes) Kind, in das eingefügt weden kann, so ist eine neues Blatt mit `o` zu erzeugen, welches das linke (bzw. rechte) Kind von `t` wird.

Es bietet sich an, diesen Algorithmus in zwei C Funktionen umzusetzen, die sich gegenseitig verschränkt rekursiv aufrufen. Eine Funktion `insert`, die ein neues Element in einen Baum einfügt, und eine Funktion `insertOrNewChild`, die eine Referenz auf ein Kind hat. Sollte es dieses Kind nicht geben, die Referenz also auf `NULL` zeigt, so ist ein neues Blatt auf dieser Referenz einzutragen, ansonsten ist das Element in dem referenzierten Baum einzufügen.

Zunächst die zwei Sinaturen der benötigten Funktionen.

BinTree.c

```
19 void insertOrNewChild(Baum* insertHere,char* n);
20 void insert(Baum this,char* n);
```

Die Umsetzung beider Funktionen ist erschreckend einfach. Beginnen wir mit der eigentlichen Einfügefunktion `insert`. Es wird mit der Standardfunktion `strcmp` der einzufügende String mit der Wurzelmarkierung verglichen. Je nach dem ist links oder rechts das Element einzufügen.

BinTree.c

```
21 void insert(Baum this,char* n){
22   if (strcmp(n,this->name)>0) insertOrNewChild(&this->rechts,n);
23   else insertOrNewChild(&this->links,n);
24 }
```

Ebenso nur mit zwei Zeilen Funktionsrumpf lässt sich die zweite Funktion umsetzen. Hier wird unterschieden, ob es noch einen Baum gibt, in dem einzufügen ist, oder ob ein Blatt erzeugt und eingehängt werden soll:

BinTree.c

```
25 void insertOrNewChild(Baum* insertHere,char* n){
26   if (*insertHere==NULL) *insertHere=newBlatt(n);
27   else insert(*insertHere,n);
28 }
```

Das war es schon. Zum Testen brauchen wir Funktionen, die einen Baum ausgeben können, so dass man sich ein Bild von seiner Struktur machen kann. Hierzu die folgende Funktion, die einen Binärbaum ausgibt.

BinTree.c

```
29 void nl(int i){printf("\n");for (;i>0;i--) printf(" ");}
30
31 int isLeaf(Baum this){return this->links==NULL&&this->rechts==NULL;}
32
33 void printTree(Baum this,int indent){
34   if (this!=NULL){
35     printf("%s",this->name);
36     if (!isLeaf(this)){
37       nl(indent);
38       printf("[");
39       printTree(this->links,indent+1);
40       nl(indent);
41       printf(",");
42       printTree(this->rechts,indent+1);
43       nl(indent);
44       printf("]");
45     }
46   }else
47     printf("NULL");
48 }
```

Eine der interessanten Eigenschaften eines binären Suchbaums ist es, dass wenn zunächst die Elemente des linken Kindes, dann die Wurzelmakierung und schließlich die Elemente des rechten Kindes ausgegeben werden, die Elemente in sortierter Reihenfolge ausgegeben werden.

BinTree.c

```
49 void printSorted(Baum this){
50   if (this==NULL) return;
51   printSorted(this->links);
52   printf(" %s ",this->name);
53   printSorted(this->rechts);
54 }
```

Ein minimaler test soll uns von der Arbeitsweise obiger Funktionen überzeugen:

BinTree.c

```
55 int main(){
56   Baum b = newBlatt("Sven Eric");
57   insert(b,"Sebastian");
58   insert(b,"Tristan");
59   insert(b,"Jan");
60   insert(b,"Andreas");
61   insert(b,"Michael");
62   insert(b,"Thorsten");
63   insert(b,"Robert");
```

```
64    insert(b,"Thobias");
65    insert(b,"Daniel");
66    printTree(b,1);
67    printSorted(b);
68    return 0;
69  }
```

das Programm erzeugt folgende kleine Ausgabe:

```
sep@pc305-3:~/fh/c/student> bin/BinTree
Sven Eric
 [Sebastian
  [Jan
   [Andreas
    [NULL
    ,Daniel
    ]
   ,Michael
    [NULL
    ,Robert
    ]
   ]
  ,NULL
  ]
 ,Tristan
  [Thorsten
   [Thobias
   ,NULL
   ]
  ,NULL
  ]
 ] Andreas  Daniel  Jan  Michael  Robert  Sebastian  Sven Eric  Thobias  Thorsten  Tristan sep@pc305-3:~/fh/c/s
```

4.12 Praktikumsaufgabe

In dieser Projektaufgabe sollen Programme geschrieben werden, die farbige Bilddateien erzeugen. Als einfachstes Bildformat benutzen wir das Format `bmp`, das Bilder als einfache Bitmapdateien speichert.

Hierzu sei die folgende Header-Datei einer kleinen Bibliothek zum Erzeugen von Bitmapdateien gegeben. Darin werden zwei einfache Strukturen definiert. Eine um Farben zu speichern:

BMP.h

```
1  #ifndef BMP__H_
2  #define BMP__H_
3
4  #include <stdlib.h>
5  #include <stdio.h>
6
7  typedef struct {
8    unsigned char red;
9    unsigned char green;
10   unsigned char blue;
```

```
11 } Color;
12
13 const static Color BLACK = {0,0,0};
14 const static Color WHITE = {255,255,255};
15 const static Color RED   = {255,0,0};
16 const static Color GREEN = {0,255,0};
17 const static Color BLUE  = {0,0,255};
18 const static Color YELLOW= {255,255,0};
```

Und eine für die eigentlichen Bitmapdaten.

BMP.h

```
19 typedef struct {
20   int width;
21   int height;
22   Color* imagedata ;
23 } Bmp;
```

Die Bibliothek sieht vier Funktionen vor:

- `newBmp`: zum Erzeugen eines neuen Bitmapobjektes im Speicher.
- `deleteBmp`: um ein Bitmapobjekt wieder aus dem Speicher zu löschen.
- `writeBmpToFile`: um ein Bitmapobjekt in eine Datei zu schreiben.
- `setBmpPoint`: um einen Punkt mit einer Farbe zu beschreiben.

BMP.h

```
24 Bmp* newBmp(int width,int  height);
25 void deleteBmp(Bmp* this);
26 void writeBmpToFile(Bmp* this,char* fileName);
27 void setBmpPoint(Bmp* b,Color c,int x, int y);
28
29 #endif
30
```

Es folgt die Implementierung dieser Bibliothek:

BMP.c

```
1  #include "BMP.h"
2  #include <stdlib.h>
3  #include <stdio.h>
4
5  Bmp* newBmp(int width,int height){
6    Bmp* result = (Bmp*)malloc(sizeof(Bmp));
7    result->imagedata =(Color*)malloc(sizeof(Color)*width*height);
8    result->width=width;
9    result->height=height;
10   return result;
```

```
11 }
12 void deleteBmp(Bmp* this){
13   free(this->imagedata);
14   free(this);
15 }
16
17 void setBmpPoint(Bmp* this,Color c,int x, int y){
18   if (!(x<0||x>=this->width||y<0||y>=this->height)){
19     this->imagedata[y*this->width+x]=c;
20   }
21 }
22
23 void putByte(FILE* out,unsigned char ch) {fputc(ch,out);}
24
25 void putShort(FILE* out, unsigned short w) {
26   putByte(out, w & 0xff);
27   putByte(out, (w>>8) & 0xff);
28 }
29 void putLong(FILE* out, unsigned int l) {
30   putByte(out, l & 0xff);
31   putByte(out, (l>>8) & 0xff);
32   putByte(out, (l>>16) & 0xff);
33   putByte(out, (l>>24) & 0xff);
34 }
35 void putColor(FILE* out, Color c) {
36   putByte(out,c.blue);
37   putByte(out,c.green);
38   putByte(out,c.red);
39 }
40 int getBytesPerLines(Bmp* this){
41   long bytesPerLine = this->width * 3l;  /* (for 24 bit images) */
42   bytesPerLine=bytesPerLine+(4-bytesPerLine%4)%4;
43   return bytesPerLine;
44 }
45 void writeBmpToFile(Bmp* this,char* fileName) {
46   FILE* out = fopen(fileName, "wb");
47   if (out == NULL){
48     printf("Error opening output file\n");
49     exit(1);
50   }
51   putByte(out,'B'); // "BM"-ID schreiben
52   putByte(out,'M');
53
54   int headersize = 54L;
55   int bytesPerLine = getBytesPerLines(this);
56   int filesize  = headersize+bytesPerLine*this->height;
57   putLong(out,filesize);
58
59   putShort(out,0);
60   putShort(out,0);
```

```
61    putLong(out, headersize);
62    putLong(out, 0x28L);       //infoSize
63    putLong(out,this-> width);
64    putLong(out,this-> height);
65    putShort(out, 1);          //biPlanes
66    putShort(out, 24);         //bits
67    putLong(out,0);            //(no compression)
68    putLong(out,0);
69    putLong(out,0);
70    putLong(out,0);
71    putLong(out,0);
72    putLong(out,0);
73
74    int line, colum;
75    for (line=0;line < this->height;line++){
76      for (colum=0;colum < this->width;colum++){
77        putColor(out,this->imagedata[line*this->width+colum]);
78      }
79      int missingBytes=bytesPerLine-3*this->width;
80      for (;missingBytes>0;missingBytes--)
81        putByte(out,0);
82    }
83    fclose(out);
84  }
```

Es folgt eine kleine Beispielanwendung, in der eine erste Bilddatei erzeugt wird.

FirstBitmap.c

```
1  #include "BMP.h"
2  int main(){
3    Bmp* bh = newBmp(850,400);
4    int y,x;
5    for (y=0;y<bh->height;y++){
6      for (x=0;x<bh->width;x++){
7        Color c={0,x,0};
8        setBmpPoint(bh,c,x,y);
9      }
10   }
11   writeBmpToFile(bh,"test.bmp");
12   deleteBmp(bh);
13   return 0;
14 }
```

Nun sind Sie dran, Programme zu schreiben, mit denen Bilder generiert werden können.

Aufgabe 41 Schreiben Sie eine kleine Bibliothek, mit deren Hilfe Sie geometrische Figuren in Bitmapdateien scheiben können.

a) Schreiben Sie eine Prozedur
`void background(Bmp* this,Color c)`
Sie soll das ganze Bild mit einer Hintergrundfarbe ausfüllen.

b) Schreiben Sie eine Prozedur
`void fillRect(Bmp* this,Color c,int x,int y,int w,int h)`
Sie soll ein farbiges Rechteck mit der linken unteren Ecke `(x,y)`, der Weite `w` und der Höhe `h` in die Bitmapdatei zeichnen.

c) Schreiben Sie eine Prozedur
`void drawLine(Bmp* this,Color c,int fromX,int fromY,int toX,int toY)`
Sie soll eine farbiges Linie mit dem Startpunkt `(fromX,fromY)` und dem Endpunkt `(toX,toY)` in die Bitmapdatei zeichnen.

d) Schreiben Sie eine Prozedur
`void drawRect(Bmp* this,Color c,int x,int y,int w,int h)`
Sie soll die farbige Umrandung eines Rechtecks mit der linken unteren Ecke `(x,y)`, der Weite `w` und der Höhe `h` in die Bitmapdatei zeichnen.

e) Schreiben Sie eine Prozedur
`void drawCircle(Bmp* this,Color c,int x,int y,int radius)`
Sie soll die farbige Umrandung eines Kreises mit dem Mittelpunkt `(x,y)` und dem Radius `radius` in die Bitmapdatei zeichnen.

f) Schreiben Sie eine Prozedur
`void fillCircle(Bmp* this,Color c,int x,int y,int radius)`
Sie soll einen farbigen Kreises mit dem Mittelpunkt `(x,y)` und dem Radius `radius` in die Bitmapdatei zeichnen.

g) Schreiben Sie ein Programm, das Ihre Bibliothek nutzt und verschiedene interessante Bilder erzeugt.

Lösung

geo.h

```
1  #ifndef GEO__H_H
2  #define GEO__H_H
3
4  #include "BMP.h"
5  void background(Bmp* this,Color c);
6  void fillRect(Bmp* this,Color c,int x,int y,int w,int h);
7  void drawLine(Bmp* this,Color c,int fromX,int fromY,int toX,int toY);
8  void drawRect(Bmp* this,Color c,int x,int y,int w,int h);
9  void drawCircle(Bmp* this,Color c,int x,int y,int radius);
10 void fillCircle(Bmp* this,Color c,int x,int y, int radius);
11 #endif
12
```

geo.c

```
1  #include "BMP.h"
2  #include <math.h>
3
```

```
4  #ifndef M_PI
5  #define M_PI  3.14159265358979323846
6  #endif
7
8  void background(Bmp* this,Color c){
9    for (int y=0;y<this->height;y++){
10     for (int x=0;x<this->width;x++)
11       setBmpPoint(this,c,x,y);
12   }
13 }
14
15 void fillRect(Bmp* this,Color c,int x,int y,int w,int h){
16   for (int i=y;i<y+h;i++){
17     for (int j=x;j<x+w;j++)
18       setBmpPoint(this,c,j,i);
19   }
20 }
21
22 void swapValues(int* x,int* y){
23   int z= *x;
24   *x= *y;
25   *y = z;
26 }
27
28 void drawLine(Bmp* this,Color c,int fromX,int fromY,int toX,int toY){
29   int xDiv = abs(toX-fromX);
30   int yDiv = abs(toY-fromY);
31
32   if (fromX==toX && fromY==toY){
33     setBmpPoint(this,c,fromX,fromY);
34   }else if (xDiv>yDiv){
35     if (fromX>toX){
36       swapValues(&fromY,&toY);
37       swapValues(&fromX,&toX);
38     }
39     double steigung= ((double)toY-fromY)/(toX-fromX);
40     for(int x=fromX;x<=toX;x++){
41       setBmpPoint(this,c,x,fromY+(int)(x-fromX)*steigung);
42     }
43   }else{
44     if (fromY>toY){
45       swapValues(&fromY,&toY);
46       swapValues(&fromX,&toX);
47     }
48     double steigung= ((double)toX-fromX)/(toY-fromY);
49     for(int y=fromY;y<=toY;y++){
50       setBmpPoint(this,c,fromX+(int)(y-fromY)*steigung,y);
51     }
52   }
53 }
```

```
54
55 void drawRect(Bmp* this,Color c,int x,int y,int w,int h){
56   drawLine(this,c,x,y,x+w,y);
57   drawLine(this,c,x+w,y,x+w,y+h);
58   drawLine(this,c,x+w,y+h,x,y+h);
59   drawLine(this,c,x,y+h,x,y);
60 }
61
62 void drawCircle(Bmp* this,Color c,int x,int y,int radius){
63   if (radius<0) radius= -radius;
64   const int maxAlpha=360*(1+radius/100);
65   for (int alpha=0;alpha<maxAlpha;alpha++){
66     setBmpPoint(this,c
67                 ,x+(int)(radius*sin(2*M_PI*alpha/maxAlpha))
68                 ,y+(int)(radius*cos(2*M_PI*alpha/maxAlpha)));
69   }
70 }
71
72 void fillCircle(Bmp* this,Color c,int x,int y, int radius){
73   if (radius<0) radius= -radius;
74   for (int i=y-radius;i<y+radius;i++)
75     for (int j=x-radius;j<x+radius;j++)
76       if ((i-y)*(i-y)+(j-x)*(j-x)<radius*radius)
77           setBmpPoint(this,c,j,i);
78 }
```

testGeo.c

```
1  #include "BMP.h"
2  #include <math.h>
3  #include "geo.h"
4
5  int main(){
6    Bmp* bmp=newBmp(200,100);
7    background(bmp, BLACK);
8    fillRect(bmp,RED,20,20,80,30);
9    fillRect(bmp,BLUE,80,40,80,50);
10   writeBmpToFile(bmp,"b1.bmp");
11   deleteBmp(bmp);
12
13   bmp=newBmp(200,200);
14   background(bmp, RED);
15   drawRect(bmp,BLUE,80,40,80,50);
16   drawLine(bmp,GREEN,10,10,50,90);
17   drawLine(bmp,GREEN,10,10,50,190);
18   drawLine(bmp,GREEN,10,10,90,50);
19   drawLine(bmp,GREEN,10,10,190,50);
20   drawLine(bmp,GREEN,10,10,10,150);
21   drawLine(bmp,GREEN,10,10,150,10);
22
```

```
23
24    drawLine(bmp,GREEN,50,90,10,10);
25    drawLine(bmp,GREEN,50,190,10,10);
26    drawLine(bmp,GREEN,90,50,10,10);
27    drawLine(bmp,GREEN,190,50,10,10);
28    drawLine(bmp,GREEN,10,150,10,10);
29    drawLine(bmp,GREEN,150,10,10,10);
30
31    drawCircle(bmp,BLACK,100,100,100);
32    drawCircle(bmp,BLACK,100,100,90);
33    drawCircle(bmp,BLACK,100,100,80);
34    drawCircle(bmp,BLACK,100,100,70);
35    drawCircle(bmp,BLACK,100,100,60);
36    drawCircle(bmp,BLACK,100,100,50);
37    drawCircle(bmp,BLACK,-900,-900,1500);
38
39    writeBmpToFile(bmp,"b2.bmp");
40    deleteBmp(bmp);
41
42    bmp=newBmp(200,200);
43    background(bmp, BLUE);
44    fillCircle(bmp,YELLOW,100,100,80);
45    fillCircle(bmp,GREEN,1,1,70);
46    writeBmpToFile(bmp,"b3.bmp");
47
48    deleteBmp(bmp);
49    return 0;
50  }
51
```

Aufgabe 42 In dieser Aufgabe sollen Sie mit Polynomen rechnen. Polynome bestehen aus der Summe von Monomen. Ein Monom ist der Form $c * x^e$, wobei c der Koeffizient eine reelle Zahl ist und e der Exponent, eine natürliche Zahl inklusive 0 ist.

a) Definieren Sie eine Datenstruktur, mit der Sie Polynome adequat darstellen können. **Tipp**: Betrachten Sie hierzu ein Polynom als eine Art Liste von Monomen. Definieren Sie sich also zunächst ein `struct`, mit dem Sie Monome darstellen können. Dann definieren Sie sich ein `struct`, mit dem Sie eine Verkettung von Monomen darstellen können.

b) Schreiben Sie eine Funktion:
`double eval(Polynomial this,double x)`
die für einen Eingabewert das Polynom ausrechnet.

c) Schreiben Sie eine Funktion:
`Polynomial derivation(Polynomial this);`
die für ein Polynom die erste Ableitung erzeugt.

d) Schreiben Sie eine Prozedur:
`void drawPolynomial(Bmp* this,Color c,Polynomial poly`

`,int minX,int maxX, int minY);` Sie soll den Graphen des Polynoms in einer Bitmapdatei zeichnen. `minX` und `maxX` geben den gewünschten Wertebereich auf der x-Achse an. `minY` den y-Wert in der untersten Bildzeile an.

Lösung

poly.h

```
#ifndef POLY__H_H
#define POLY__H_H

#include "BMP.h"

typedef struct{
  double coefficient;
  int exponent ;
} Monomial;

struct PolyStruct{
  Monomial mono;
  struct PolyStruct* next;
};

typedef  struct PolyStruct* Polynomial;

Polynomial newMonoPolynomial(Monomial m);
Polynomial newPolynomial(Monomial m,Polynomial p);
void deletePolynomial(Polynomial this);

double eval(Polynomial this,double x);
Polynomial derivation(Polynomial this);
void drawPolynomial(Bmp* this,Color c,Polynomial poly
                   ,int minX,int maxX, int minY);

#endif
```

poly.c

```
#include "poly.h"
#include "geo.h"
#include <stdlib.h>
#include <math.h>

Polynomial newMonoPolynomial(Monomial m){
  Polynomial this=(Polynomial)malloc(sizeof(struct PolyStruct));
  this->mono=m;
  this->next=NULL;
  return this;
}
Polynomial newPolynomial(Monomial m,Polynomial p){
  Polynomial this=(Polynomial)malloc(sizeof(struct PolyStruct));
  this->mono=m;
```

```
15    this->next=p;
16    return this;
17  }
18  void deletePolynomial(Polynomial this){
19    if (this->next!=NULL) deletePolynomial(this->next);
20    free(this);
21  }
22  double evalMonomial(Monomial m,double x){
23    return m.coefficient*pow(x,m.exponent);
24  }
25  double eval(Polynomial this,double x){
26    double result=evalMonomial(this->mono,x);
27    if (this->next!=NULL)result=result+eval(this->next,x);
28    return result;
29  }
30  Monomial derivationMonomial(Monomial m){
31    Monomial result;
32    if (m.exponent==0){
33      result.exponent=0;
34      result.coefficient=0;
35    }else {
36      result.exponent=m.exponent-1;
37      result.coefficient=m.coefficient*m.exponent;
38    }
39    return result;
40  }
41
42  Polynomial derivation(Polynomial this){
43    Monomial mono = derivationMonomial(this->mono);
44    if (this->next==NULL) return  newMonoPolynomial(mono);
45    return  newPolynomial(mono,derivation(this->next));
46  }
47
48  void drawPolynomial(Bmp* this,Color c,Polynomial poly
49                      ,int minX,int maxX, int minY){
50    double valuePerPixel = ((double)maxX-minX)/this->width;
51    for (int x=0;x<this->width;x++){
52      int y1
53       =(int)((eval(poly,minX+valuePerPixel*x)-minY)/valuePerPixel);
54      int y2=(int)((eval(poly,minX+valuePerPixel*(x+1))-minY)
55                   /valuePerPixel);
56      drawLine(this,c,x,y1,x+1,y2);
57    }
58  }
```

testPoly.c

```
1   #include "BMP.h"
2   #include "poly.h"
3   #include "geo.h"
```

```
4
5  int main(){
6    Bmp* bmp=newBmp(200,400);
7    background(bmp, WHITE);
8
9    Monomial m = {0.02,3};
10   Polynomial p=newMonoPolynomial(m);
11
12   drawPolynomial(bmp,RED,p,-20,20,-20);
13   Polynomial p1=derivation(p);
14   drawPolynomial(bmp,BLUE,p1,-20,20,-20);
15   Polynomial p2=derivation(p1);
16   drawPolynomial(bmp,GREEN,p2,-20,20,-20);
17   Polynomial p3=derivation(p2);
18   drawPolynomial(bmp,BLACK,p3,-20,20,-20);
19   writeBmpToFile(bmp,"b4.bmp");
20
21   deletePolynomial(p);
22   deletePolynomial(p1);
23   deletePolynomial(p2);
24   deletePolynomial(p3);
25   deleteBmp(bmp);
26   return 0;
27 }
28
```

Aufgabe 43 In dieser Aufgabe sollen Sie die Mandelbrotmenge in einer Bitmapdatei zeichnen. Hierzu brauchen Sie die Bibliothek für komplexe Zahlen aus Übungsblatt Nummer 5.

a) Erzeugen Sie eine Bitmapdatei mit der Weite `480` und der Höhe `430`. Pro Pixel soll dabei ein reeller Abstand von 0.00625 dargestellt werden. Die linke untere Ecke soll der Punkt $(x, y) = (-2, -1, 35)$ sein.
Berechnen Sie für jeden Punkt (x, y) nun die Schwelle der komplexen Zahl $x + yi$ mit dem Schwellwert 4. Sie dürfen die Funktion `schwelle` aus Ihrer Bibliothek für komplexe Zahlen so abändern, dass sie maximal 50 als Schwelle berechnet. Ansonsten könnte es passieren, dass die Funktion `schwelle` nicht terminiert.
Färben Sie nun die Punkte für unterschiedlichen Schwellwerte unterschiedlich ein, so dass Sie eine interessante farbige Darstellung der Mandelbrotmenge erhalten.

Lösung

apfel.c

```
1  #include "Complex.h"
2  #include "BMP.h"
3
4  Bmp* newApfel(){
5    Bmp* result = newBmp(480,430);
6    for (int y=0;y<result->height;y++)
7      for (int x=0;x<result->width;x++){
```

```
8           Complex c={-2+x*0.00625,-1.35+y*0.00625};
9           unsigned int s=schwelle(c,4)%50;
10          Color col={5*s%255,s%5*30,s%5*50};
11          setBmpPoint(result,col,x,y);
12      }
13    return result;
14  }
15
16  int main(){
17      Bmp* apfel=newApfel();
18      writeBmpToFile(apfel,"apfel.bmp");
19      deleteBmp(apfel);
20      return 0;
21  }
22
23
```

b) Erzeugen Sie mehrere Bilder, in denen Sie in die in Aufgabe a) dargestellte Figur hineinzoomen.

Kapitel 5

Programmiersprachen

Nachdem wir nun schon sehr tief in die Programmiersprache C eingetaucht sind, und bereits Programme mit einer komplexen Funktionalität schreiben können, ist vielleicht Zeit einmal innezuhalten und sich zu überlegen, was eine Programmiersprache eigentlich genau ist, wie eine Programmiersprache definiert und beschrieben wird und wie im Endeffekt ein Compiler arbeitet.

5.1 Syntax und Semantik

Programmiersprachen stellen formale Systeme dar. In einem formalen System gilt zu unterscheiden:

- die äußere Struktur, in dem Sätze des Systems notiert werden, der sogenannten Syntax,
- und der Bedeutung dieser Sätze, der sogenannten Semantik.

5.1.1 Syntax

Sprachen sind ein fundamentales Konzept nicht nur der Informatik. In der Informatik begegnen uns als auffälligste Form der Sprache*Programmiersprachen*. Es gibt gewisse Regeln, nach denen die Sätze einen Sprache aus einer Menge von Wörtern geformt werden können. Die Regeln nennen wir im allgemeinen eine *Grammatik*. Ein Satz einer Programmiersprache nennen wir Programm. Die Wörter sind, Schlüsselwörter, Bezeichner, Konstanten und Sonderzeichen.

Ausführlich beschäftigt sich die Vorlesung Informatik 2 mit formalen Sprachen. Die praktische Umsetzung der Theorie in Form eines Compilers wird in der Vorlesung *Compilerbau*[Web02][Pan04] behandelt. Wir werden in diesem Kapitel die wichtigsten Grundkenntnisse hierzu betrachten, wie sie zum Handwerkszeug eines jeden Informatikers gehören.

Eine der bahnbrechenden Erfindungen des 20. Jahrhunderts geht auf den Sprachwissenschaftler Noam Chomsky[Cho56][1] zurück. Er präsentierte als erster ein formales Regelsystem, mit dem

[1] Chomsky gilt als der am häufigsten zitierte Wissenschaftler des 20. Jahrhunderts. Heutzutage tritt Chomsky weniger durch seine wissenschaftlichen Arbeiten als vielmehr durch seinen Einsatz für Menschenrechte und bedrohte Völker in Erscheinung.

die Grammatik einer Sprache beschrieben werden kann. Dieses Regelsystem ist in seiner Idee verblüffend einfach. Es bietet Regeln an, mit denen mechanisch die Sätze einer Sprache generiert werden können.

Systematisch wurden Chomsky Ideen zum erstenmal für die Beschreibung der Syntax der Programmiersprache Algol angewendet[NB60].

5.1.2 kontextfreie Grammatik

Eine kontextfreie Grammatik besteht aus

- einer Menge $\mathcal{T}$ von Wörteren, den *Terminalsymbole.*
- einer Menge $\mathcal{N}$ von Nichtterminalsymbolen.
- ein ausgezeichnetes Startsymbol $S \in \mathcal{N}$.
- einer endlichen Menge $\mathcal{R}$ von Regeln der Form:
  nt: := $t_1 \ldots t_n$, wobei $nt \in \mathcal{N}$, $t_i \in \mathcal{N} \cup \mathcal{T}$.

Mit den Regeln einer kontextfreien Grammatik werden Sätze gebildet, indem ausgehend vom Startsymbol Regel angewendet werden. Bei einer Regelanwendung wird ein Nichtterminalzeichen t durch die Rechte Seite einer Regel, die t auf der linken Seite hat, ersetzt.

Beispiel 5.1.1 *Wir geben eine Grammatik an, die einfache Sätze über unser Sonnensystem auf Englisch bilden kann:*

- $\mathcal{T}$= {*mars,mercury,deimos,phoebus,orbits,is,a,moon,planet*}
- $\mathcal{N}$= {start,noun-phrase,verb-phrase,noun,verb,article}
- S =start
- start *::=* noun-phrase verb-phrase

  noun-phrase *::=* noun
  noun-phrase *::=* article noun

  verb-phrase *::=* verb noun-phrase

  noun *::= planet*
  noun *::= moon*
  noun *::= mars*
  noun *::= deimos*
  noun *::= phoebus*

  verb *::= orbits*
  verb *::= is*

  article *::= a*

Wir können mit dieser Grammatik Sätze in der folgenden Art bilden:

- start
  →noun-phrase verb-phrase
  →article noun verb-phrase
  →article noun verb noun-phrase
  → *a* noun verb noun-phrase
  → *a moon* verb noun-phrase
  → *a moon orbits* noun-phrase
  → *a moon orbits* noun
  → *a moon orbits mars*

- start
  →noun-phrase verb-phrase
  →noun verb-phrase
  → *mercury* verb-phrase
  → *mercury* verb noun-phrase
  → *mercury is* noun-phrase
  → *mercury is* article noun
  → *mercury is a* noun
  → *mercury is a planet*

- *Mit dieser einfachen Grammatik lassen sich auch Sätze bilden, die weder korrektes Englisch sind, noch eine vernünftige inhaltliche Aussage machen:*

  start
  →noun-phrase verb-phrase
  →noun verb-phrase
  → *planet* verb-phrase
  → *planet* verb noun-phrase
  → *planet orbits* noun-phrase
  → *planet orbits* article noun
  → *planet orbits a* noun
  → *planet orbits a phoebus*

Eine Grammatik beschreibt die Syntax einer Sprache im Gegensatz zur Semantik, der Bedeutung, einer Sprache.

Rekursive Grammatiken

Die Grammatik aus dem letzen Beispiel kann nur endlich viele Sätze generieren. Will man mit einer Grammatik unendlich viele Sätze beschreiben, so wie eine Programmiersprache unendlich viele Programme hat, so kann man sich dem Trick der Rekursion bedienen. Eine Grammatik kann rekursive Regeln enthalten; das sind Regeln, in denen auf der rechten Seite das Nichtterminalsymbol der linken Seite wieder auftaucht.

Beispiel 5.1.2 *Die folgende Grammatik erlaubt es arithmetische Ausdrücke zu generieren:*

- $\mathcal{T} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, *, /\}$

- $\mathcal{N}$= {start,expr,op,integer,digit,}
- S =start
- start *::=* expr

  expr *::=* integer
  expr *::=* integer op expr

  integer *::=* digit

  integer *::=* digit integer

  op *::= +*
  op *::=* -
  op *::= **
  op *::= /*

  digit *::= 0*
  digit *::= 1*
  digit *::= 2*
  digit *::= 3*
  digit *::= 4*
  digit *::= 5*
  digit *::= 6*
  digit *::= 7*
  digit *::= 8*
  digit *::= 9*

Diese Grammatik hat zwei rekursive Regeln: eine für das Nichtterminal expr *und eines für das Nichtterminal* integer*.*

Folgende Abeleitung generiert einen arithmetischen Ausdrucke mit dieser Grammatik:

- start
  →expr
  →integer op expr
  →integer op integer op expr
  →integer op integer op integer op expr
  →integer op integer op integer op integer
  →integer + integer op integer op integer
  →integer + integer ∗ integer op integer
  →integer + integer ∗ integer −integer
  →digit integer + integer ∗ integer − integer
  → 1 integer + integer ∗ integer − integer
  → 1 digit integer + integer ∗integer − integer
  → 12 integer + integer ∗integer − integer
  → 12 digit + integer ∗ integer − integer
  → 129+ integer ∗ integer −integer
  → 129+ digit ∗ integer −integer
  → 129 + 4∗ integer − integer
  → 129 + 4∗ digit integer − integer
  → 129 + 4 ∗ 5integer − integer
  → 129 + 4 ∗ 5digit − integer

$\rightarrow 129 + 4 * 53-$ integer
$\rightarrow 129 + 4 * 53-$ digit integer
$\rightarrow 129 + 4 * 53 - 8$ integer
$\rightarrow 129 + 4 * 53 - 8$ digit
$\rightarrow 129 + 4 * 53 - 87$

Aufgabe 44 Erweitern Sie die obige Grammatik so, dass sie mit ihr auch geklammerte arithmetische Ausdrücke ableiten können. Hierfür gibt es zwei neue Terminalsymbolde: `(` und `)`.

Schreiben Sie eine Ableitung für den Ausdruck: `1+(2*20)+1`

Grenzen kontextfreier Grammatiken

Kontextfreie Grammatiken sind ein einfaches und dennoch mächtiges Beschreibungsmittel für Sprachen. Dennoch gibt es viele Sprachen, die nicht durch eine kontextfreie Grammatik beschrieben werden können.

syntaktische Grenzen Es gibt syntaktisch recht einfache Sprachen, die sich nicht durch eine kontextfreie Grammatik beschreiben lassen. Eine sehr einfache solche Sprache besteht aus drei Wörtern: $\mathcal{T} =$ {a,b,c}. Die Sätze dieser Sprache sollen so gebildet sein, daß für eine Zahl n eine Folge von n mal dem Zeichen a, n mal das Zeichen b und schließlich n mal das Zeichen c folgt, also

$$\{a^n b^n c^n | n \in I\!N\}.$$

Die Sätze dieser Sprache lassen sich aufzählen:

abc
aabbcc
aaabbbccc
aaaabbbbcccc
aaaaabbbbbccccc
aaaaaabbbbbbcccccc
...

Es gibt formale Beweise, daß derartige Sprachen sich nicht mit kontextfreie Grammatiken bilden lassen. Versuchen Sie einmal das Unmögliche: eine Grammatik aufzustellen, die diese Sprache erzeugt.

Eine weitere einfache Sprache, die nicht durch eine kontextfreie auszudrücken ist, hat zwei Terminalsymbole und verlangt, daß in jedem Satz die beiden Symbole gleich oft vorkommen, die Reihenfolge jedoch beliebig sein kann.

semantische Grenzen Über die Syntax hinaus, haben Sprachen noch weitere Einschränkungen, die sich nicht in der Grammatik ausdrücken lassen. Die meisten syntaktisch korrekten C-Programme werden trotzdem vom C-Übersetzer als inkorrekt zurückgewiesen. Diese Programme verstoßen gegen semantische Beschränkungen, wie z.B. gegen die Zuweisungskompatibilität. Das Programm:

```
1 int i = "1";
```

ist syntaktisch nach den Regeln der C-Grammatik korrekt gebildet, verletzt aber die Beschränkung, daß einem Feld vom Typ `int` kein Objekt des Typs `String` zugewiesen werden darf.

Aus diesen Grund besteht ein Übersetzer aus zwei großen Teilen. Der syntaktischen Analyse, die prüft, ob der Satz mit den Regeln der Grammatik erzeugt werden kann und der semantischen Analyse, die anschließend zusätzliche semantische Bedingungen prüft.

Das leere Wort

Manchmal will man in einer Grammatik ausdrücken, dass in Nichtterminalsymbol auch zu einem leeren Folge von Symbolen reduzieren soll. Hierzu könnte man die Regel

$t ::=$

mit leerer rechter Seite schreiben. Es ist eine Konvention ein spezielles Zeichen für das leere Wort zu benutzen. Hierzu bedient man sich des griechischen Buchtstabens ϵ. Obige Regel würde man also schreiben als:

$t ::= \epsilon$

Lexikalische Struktur

Bisher haben wir uns keine Gedanken gemacht, woher die Wörter unserer Sprache kommen. Wir haben bisher immer eine gegebene Menge angenommen. Die Wörter einer Sprache bestimmen ihre lexikalische Struktur. In unseren obigen Beispielen haben wir sehr unterschiedliche Arten von Wörtern: einmal Wörter der englischen Sprache und einmal Ziffernsymbole und arithmetische Operatorsymbole. Im Kontext von Programmiersprachen spricht man von Token.

Bevor wir testen können, ob ein Satz mit einer Grammatik erzeugt werden kann, sind die einzelnen Wörter in diesem Satz zu identifizieren. Dieses geschieht in einer lexikalischen Analyse. Man spricht auch vom *Lexer* und *Tokenizer*. Um zu beschreiben, wie die einzelnen lexikalischen Einheiten einer Sprache aussehen, bedient man sich eines weiteren Formalismus, den regulären Ausdrücken.

Andere Grammatiken

Die in diesem Kapitel vorgestellten Grammatiken heißen *kontextfrei*, weil eine Regel für ein Nichtterminalzeichen angewendet wird, ohne dabei zu betrachten, was vor oder nach dem Zeichen für ein weiteres Zeichen steht, der Kontext also nicht betrachtet wird. Läßt man auch Regeln zu, die auf der linken Seite nicht ein Nichtterminalzeichen stehen haben, so kann man mächtigere Sprachen beschreiben, als mit einer kontextfreien Grammatik.

Beispiel 5.1.3 *Wir können mit der folgenden nicht-kontextfreien Grammatik die Sprache beschreiben, in der jeder Satz gleich oft die beiden Terminalsymbole, eber in beliebiger Reihenfolge enthält.*

- $\mathcal{T}$= {a,b}
- $\mathcal{N}$= {start,A,B}
- S =start
- start ::= ABstart
  start ::= ϵ

  AB ::= BA

  BA ::= AB

  A ::= a

  B ::= b

  start
  →ABstart
  →ABABstart
  →ABABABstart
  →ABABABABstart
  →ABABABABABstart
  →ABABABABAB
  →AABBABABAB
  →AABBBAABAB
  →AABBBABAAB
  →AABBBABABA
  →AABBBABBAA
  →AABBBBABAA
  →AABBBBBAAA
  → aABBBBBAAA
  → aaBBBBBAAA
  → aabBBBBAAA
  → $aabb$BBBAAA
  → $aabbb$BBAAA
  → $aabbbb$BAAA
  → $aabbbbb$AAA
  → $aabbbbba$AA
  → $aabbbbbaa$A
  → $aabbbbbaaa$

Beispiel 5.1.4 *Auch die nicht durch eine kontextfreie Grammatik darstellbare Sprache:*

$\{a^n b^n c^n | n \in I\!N\}$.

läßt sich mit einer solchen Grammatik generieren:

S::=abTc
T::=AbT$c|\epsilon$
bA::=Ab
aA::=aa

Eine Ableitung mit dieser Grammatik sieht wie folgt aus:

S
$\rightarrow ab\mathrm{T}c$
$\rightarrow ab\mathrm{A}b\mathrm{T}cc$
$\rightarrow ab\mathrm{A}b\mathrm{A}b\mathrm{T}ccc$
$\rightarrow ab\mathrm{A}b\mathrm{A}b\mathrm{A}b\mathrm{T}cccc$
$\rightarrow ab\mathrm{A}b\mathrm{A}b\mathrm{A}b\mathrm{A}b\mathrm{T}ccccc$
$\rightarrow ab\mathrm{A}b\mathrm{A}b\mathrm{A}b\mathrm{A}bccccc$
$\rightarrow ab\mathrm{AA}bb\mathrm{A}b\mathrm{A}bccccc$
$\rightarrow ab\mathrm{AA}b\mathrm{A}bb\mathrm{A}bccccc$
$\rightarrow ab\mathrm{AAA}bbb\mathrm{A}bccccc$
$\rightarrow ab\mathrm{AAA}bb\mathrm{A}bbccccc$
$\rightarrow ab\mathrm{AAA}b\mathrm{A}bbbccccc$
$\rightarrow ab\mathrm{AAAA}bbbbccccc$
$\rightarrow a\mathrm{A}b\mathrm{AAA}bbbbccccc$
$\rightarrow a\mathrm{AA}b\mathrm{AA}bbbbccccc$
$\rightarrow a\mathrm{AAA}b\mathrm{A}bbbbccccc$
$\rightarrow a\mathrm{AAAA}bbbbbccccc$
$\rightarrow aa\mathrm{AAA}bbbbbccccc$
$\rightarrow aaa\mathrm{AA}bbbbbccccc$
$\rightarrow aaaa\mathrm{A}bbbbbccccc$
$\rightarrow aaaaabbbbbccccc$

Als Preis dafür, daß man sich nicht auf kontextfreie Grammatiken beschränkt, kann nicht immer leicht und eindeutig erkannt werden, ob ein bestimmter vorgegebener Satz mit dieser Grammatik erzeugt werden kann.

Grammatiken und Bäume

Im letzten Kapitel haben wir uns ausführlich mit Bäumen beschäftigt. Eine Grammatik stellt in naheliegender Weise nicht nur ein Beschreibung einer Sprache dar, sondern jede Generierung eines Satzes dieser Sprache entspricht einem Baum, dem Ableitungsbaum. Die Knoten des Baumes sind mit Terminal- und Nichtterminalzeichen markiert, wobei Blätter mit Terminalzeichen markiert sind. Die Kinder eines Knotens sind die Knoten, die mit der rechten Seite einer Regelanwendung markiert sind.

Liest man die Blätter eines solchen Baumes von links nach rechts, so ergibt sich der generierte Satz.

Beispiel 5.1.5 *Die Ableitungen der Sätze unserer ersten Grammatik haben folgende Baumdarstellung:*

Gegenüber unserer bisherigen Darstellung der Ableitung eines Wortes mit den Regeln einer Grammatik, ist die Reihenfolge, in der die Regeln angewendet werden in der Baumdarstellung nicht mehr ersichtlich. Daher spricht man häufiger auch vom Syntaxbaum des Satzes.

Aufgabe 45 Betrachten Sie die einfache Grammatik für arithmetische Ausdrücke aus dem letzen Abschnitt. Zeichnen Sie einen Syntaxbaum für den Audruck `1+1+2*20`.

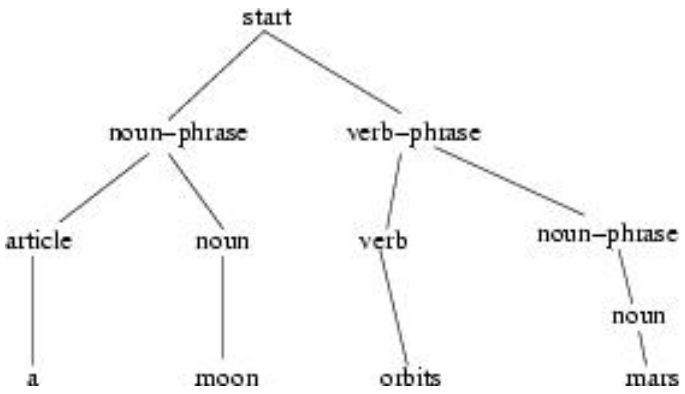


Abbildung 5.1: Ableitungsbaum für a moon orbits mars.

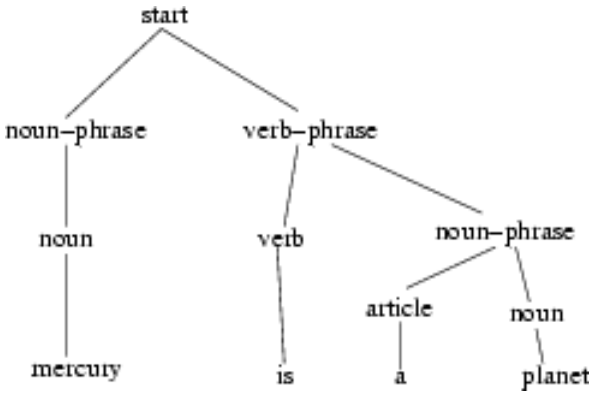


Abbildung 5.2: Ableitungsbaum für mercury is a planet.

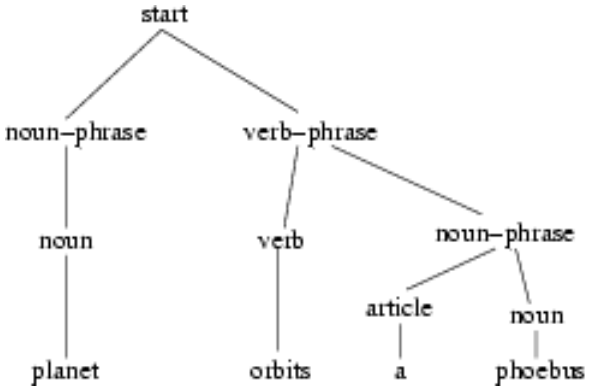


Abbildung 5.3: Ableitungsbaum für planet orbits a phoebus.

5.1.3 Erweiterte Backus-Naur-Form

Die Ausdrucksmöglichkeit einer kontextfreien Grammatik ist auf wenige Konstrukte beschränkt. Das macht das Konzept einfach. Im Kontext von Programmiersprachen gibt es häufig sprachliche Konstrukte, wie die mehrfache Wiederholung oder eine Liste von bestimmten Teilen, die

zwar mit einer kontextfreien Grammatik darstellbar ist, für die aber spezielles zusätzliche Ausdrucksmittel in der Grammatik eingeführt werden. In den nächsten Abschnitten werden wir diese Erweiterungen kennenlernen, allerdings werden wir in unseren Algorithmen für Sprachen und Grammatiken diese Erweiterungen nicht berücksichtigen.

Alternativen

Wenn es für ein Nichtterminalzeichen mehrere Regeln gibt, so werden diese Regeln zu einer Regel umgestaltet. Die unterschiedlichen rechten Seiten werden dann durch einen vertikalen Strich | gestrennt.

Durch diese Darstellung verliert man leider den direkten Zusammenhang zwischen den Regeln und einen Ableitungsbaum.

Beispiel 5.1.6 *Die Regeln unserer ersten Grammatik können damit wie folgt geschrieben werden:*

start *::=* noun-phrase verb-phrase

noun-phrase *::=* noun|article noun

verb-phrase *::=* verb noun-phrase

noun *::=* *planet*|*moon*|*mars*|*deimos*|*phoebus*

verb *::=* *orbits*|*is*

article *::=* *a*

Gruppierung

Bestimmte Teile der rechten Seite einer Grammatik können durch Klammern gruppiert werden. In diesen Klammern können wieder durch einen vertikalen Strich getrennte Alternativen stehen.

Beispiel 5.1.7 *Die einfache Grammatik für arithmetische Ausdrücke läßt sich damit ohne das Nichtterminalzeichen* op *schreiben:*

start *::=* expr

expr *::=* integer|integer $(+|-|*|/)$ expr

integer *::=* digit|digit integer

digit *::=* *0*|*1*|*2*|*3*|*4*|*5*|*6*|*7*|*8*|*9*

Wiederholungen

Ein typische Konstrukt in Programmiersprachen ist, daß bestimmte Kontrukte wiederholt werden können. So stehen z.B. in C im Rumpf einer Funktion mehrere Anweisungen. Solche Sprachkonstrukte lassen sich mit einer kontextfreien Grammatik ausdrücken.

Beispiel 5.1.8 *Eine Zahl besteht aus einer Folge von n Ziffern ($n > 0$). Dieses läßt sich durch folgende Regel ausdrücken:*

Zahl *::=* Ziffer Zahl | Ziffer

1 bis n-fach Im obigen Beispiel handelt es sich um eine 1 bis n-fache Wiederholung des Zeichens *Ziffer*. Hierzu gibt es eine abkürzende Schreibweise. Dem zu wiederholenden Teil wird das Zeichen + nachgestellt.

Beispiel 5.1.9 *Obige Regel für das Nichtterminal* Zahl *läßt sich mit dieser abkürzenden Schreibweise schreiben als:*

Zahl *::=* Ziffer+

0 bis n-fach Soll ein Teil in einer Wiederholung auch keinmal vorkommen, so wird statt des Zeichens + das Zeichen * genommen.

Beispiel 5.1.10 *Folgende Regel drückt aus, daß ein Ausdruck eine durch Operatoren getrennte Liste von Zahlen ist.*

expr *::=* Zahl | *(*Op Zahl*)**

Option Ein weiterer Spezialfall der Wiederholung ist die, in der der entsprechende Teil keinmal oder einmal vorkommen darf, d.h. der Teil ist optional. Optionale Teile werden in der erweiterten Form in eckige Klammern gesetzt.

5.1.4 Parser

Grammatiken geben an, wie Sätze einer Sprache gebildet werden können. In der Informatik interessiert der umgekehrte Fall. Ein Satz ist vorgegeben und es soll geprüft werden, ob dieser Satz mit einer bestimmten Grammatik erzeugt werden kann. Ein C-Übersetzer prüft z.B. ob ein vorgegebenes Programm syntaktisch zur durch die Javagrammatik beschriebenen Sprache gehört. Hierzu ist ein Prüfalgorithmus anzugeben, der testet, ob die Grammatik einen bestimmten Satz generieren kann. Ein solches Programm wird *Parser* genannt. Ein Parser[2] zerteilt einen Satz in seine syntaktischen Bestandteile.

5.1.5 Parsstrategien

Man kann unterschiedliche Algorithmen benutzen, um zu testen, daß ein zu der Sprache einer Grammatik gehört. Eine Strategie geht primär von der Grammatik aus, die andere betrachten eher den zu prüfenden Satz.

Rekursiv absteigend

Eine einfache Idee zum Schreiben eines Parsers ist es, einfach nacheinander auszuprobieren, ob die Regelanwendung nach und nach einen Satz bilden kann. Hierzu startet man mit dem Startsymbol und wendet nacheinander die Regeln an. Dabei wendet man immer eine Regel auf das linkeste Nichtterminalzeichen an, so lange, bis der Satzanfang abgelitten wurde, oder aber ein falscher Satzanfang abgelitten wurde. Im letzteren Fall hat man offensichtlich bei einer Regel die falsche Alternative gewählt. Man ist in eine Sackgasse geraten. Nun geht man in

[2]Lateinisch Pars bedeutet Teil.

seiner Ableitung zurück bis zu der letzten Regelanwendung, an der eine andere Alternative hätte gewählt werden können. Man spricht dann von *backtracking*; auf deutsch kann man treffend von Zurückverfolgen sprechen.

Auf diese Weise gelangt man entweder zu einer Ableitung des Satzes, oder stellt irgendwann fest, daß man alle Alternativen ausprobiert hat und immer in eine Sackgasse geraten ist. Dann ist der Satz nicht in der Sprache.

Diese Strategie ist eine Suche. Es wird systematisch aus allen möglichen Ableitungen in der Grammatik nach der Ableitung gesucht, die den gewünschten Satz findet.

Beispiel 5.1.11 *Wir suchen mit dieser Strategie die Ableitung des Satzes* a moon orbits mars *in dem Sonnensystembeispiel. Sackgassen sind durch einen Punkt • markiert.*

(0)		*start*	
(1) aus 0	→	*noun-phrase verb-phrase*	
(2a) aus 1	→	*noun verb-phrase*	
(2a3a) aus 2a	→	planet *verb-phrase*	•
(2a3b) aus 2a	→	moon *verb-phrase*	•
(2a3c) aus 2a	→	mars *verb-phrase*	•
(2a3d) aus 2a	→	deimos *verb-phrase*	•
(2a3e) aus 2a	→	phobus *verb-phrase*	•
(2b) aus 1	→	*article noun verb-phrase*	
(3) aus 2b	→	a *noun verb-phrase*	
(4a) aus 3	→	a planet *verb-phrase*	•
(4b) aus 3	→	a moon *verb-phrase*	
(5) aus 4b	→	a moon *verb noun-phrase*	
(6) aus 5	→	a moon orbits *noun-phrase*	
(7) aus 6	→	a moon orbits *noun*	
(8a) aus 7	→	a moon orbits planet	•
(8b) aus 7	→	a moon orbits moon	•
(8c) aus 7	→	a moon orbits mars	

Alternativ könnte man diese Strategie auch symetrisch nicht von links sondern von der rechten Seite an ausführen, also immer eine Regel für das rechtestes Nichtterminalsymbol anwenden.

Linksrekursion Die Strategie des rekursiven Abstiegs auf der linken Seite funktioniert für eine bestimmte Art von Regeln nicht. Dieses sind Regeln, die in einer Alternative als erstes Symbol wieder das Nichtterminalsymbol stehen haben, das auch auf der linken Seite steht. Solche Regeln heißen linksrekursiv. Unsere Strategie terminiert in diesen Fall nicht.

Beispiel 5.1.12 *Gegeben sei eine Grammatik mit einer linksrekursiven Regel:*

expr*::=*expr*+*zahl|zahl

Der Versuch den Satz

zahl+zahl

mit einem links rekursiv absteigenden Parser abzuleiten, führt zu einem nicht terminierenden rekursiven Abstieg. Wir gelangen nie in eine Sackgasse:

$$
\begin{array}{ll}
 & expr \\
\rightarrow & expr{+}zahl \\
\rightarrow & expr{+}zahl{+}zahl \\
\rightarrow & expr{+}zahl{+}zahl{+}zahl \\
\rightarrow & expr{+}zahl{+}zahl{+}zahl{+}zahl \\
 & \ldots
\end{array}
$$

Es läßt sich also nicht für alle Grammatiken mit dem Verfahren des rekursiven Abstiegs entscheiden, ob ein Satz mit der Grammatik erzeugt werden kann; aber es ist möglich, eine Grammatik mit linksrekursiven Regeln so umzuschreiben, daß sie die gleiche Sprache generiert, jedoch nicht mehr linksrekursiv ist. Hierzu gibt es ein einfaches Schema:

Eine Regel nach dem Schema:

$A ::= A\ rest | alt2$

ist zu ersetzen durch die zwei Regeln:

$A ::= alt2\ R$
$R ::= rest\ R | \epsilon$

wobei R ein neues Nichtterminalsymbol ist.

Linkseindeutigkeit Die rekursiv absteigende Strategie hat einen weiteren Nachteil. Wenn sie streng schematisch angewendet wird, führt sie dazu, daß bestimmte Prüfungen mehrfach durchgeführt werden. Diese mehrfache Ausführung kann sich bei wiederholter Regelanwendung multipizieren und zu einem sehr ineffizienten Parser führen. Grund dafür sind Regeln, die zwei Alternativen mit gleichem Anfang haben:

$S ::= AB | A$

Beide Alternativen für das Nichtterminalzeichen S starten mit dem Zeichen A. Für unseren Parser bedeutet das soweit: versuche nach der ersten Alternative zu parsen. Hierzu parse erst nach dem Symbol A. Das kann eine sehr komplexe Berechnung sein. Wenn sie gelingt, dann versuche anschließend weiter nach dem Symbol B zu parsen. Wenn das fehlschlägt, dann verwerfe die Regelalternative und versuche nach der zweiten Regelalternative zu parsen. Jetzt ist wieder nach dem Symbol A zu parsen, was wir bereits gemacht haben.

Regeln der obigen Art wirken sich auf unsere Parsstrategie ungünstig aus. Wir können aber ein Schema angegeben, wie man solche Regeln aus der Grammatik eleminiert, ohne die erzeugte Sprache zu ändern:

Regeln der Form

$S ::= AB | A$

sind zu ersetzen durch

$S ::= AT$
$T ::= B | \epsilon$

wobei T ein neues Nichtterminalzeichen ist.

Vorausschau Im letzten Abschnitt haben wir gesehen, wie wir die Grammatik umschreiben können, so daß nach dem Zurücksetzen in unserem Algorithmus es nicht vorkommt, bereits ausgeführte Regeln ein weiteres Mal zu durchlaufen. Schöner noch wäre es, wenn wir auf das Zurücksetzen ganz verzichten könnten. Dieses läßt sich allgemein nicht erreichen, aber es gibt Grammatiken, in denen man durch Betrachtung des nächsten zu parsenden Zeichens erkennen kann, welche der Regelalternativen als einzige Alternative in betracht kommt. Hierzu kann man für eine Grammatik für jedes Nichtterminalzeichen in jeder Regelalternative berechnen, welche Terminalzeichen als linkestes Zeichen in einem mit dieser Regel abgelittenen Satz auftreten kann. Wenn die Regelalternativen disjunkte solche Menge des ersten Zeichens haben, so ist eindeutig bei Betrachtung des ersten Zeichens eines zu parsenden Satzes erkennbar, ob und mit welcher Alternative dieser Satz nur parsbar sein kann.

Die gängigsten Parser benutzen diese Entscheidung, nach dem erstem Zeichen. Diese Parser sind darauf angewiesen, daß die Menge der ersten Zeichen der verschiedenen Regelalternativen disjunkt sind.

Der Vorteil an diesem Verfahren ist, daß die Token nach und nach von links nach rechts stückweise konsumiert werden. Sie können durch einen Datenstro, relisiert werden. Wurden sie einmal konsumiert, so werden sie nicht mehr zum Parsen benötigt, weil es kein Zurücksetzen gibt.

Schieben und Reduzieren

Der rekursiv absteigende Parser geht vom Startsymbol aus und versucht durch Regelanwendung den in Frage stehenden Satz abzuleiten. Eine andere Strategie ist die des Schiebens-und-Reduzierens (shift-reduce). Diese geht vom im Frage stehenden Satz aus und versucht die Regeln rückwärts anzuwenden, bis das Startsymbol erreicht wurde. Hierzu wird der Satz von links nach rechts (oder symmetrisch von rechts nach links) betrachtet und versucht, rechte Seiten von Regeln zu finden und durch ihre linke Seite zu ersetzen. Dazu benötigt man einen Marker, der angibt, bis zu welchen Teil man den Satz betrachtet. Wenn links des Markers keine linke Seite einer Regel steht, so wird der Marker ein Zeichen weiter nach rechts verschoben.

Beispiel 5.1.13 *Wir leiten im folgenden unserer allseits bekannten Beispielsatz aus dem Sonnensystem durch Schieben und Reduzieren ab. Als Marker benutzen wir einen Punkt.*

		. a moon orbits mars
(shift)	→	a . moon orbits mars
(reduce)	→	*article* . moon orbits mars
(shift)	→	*article* moon . orbits mars
(reduce)	→	*article noun* . orbits mars
(reduce)	→	*noun-phrase* . orbits mars
(shift)	→	*noun-phrase* orbits . mars
(reduce)	→	*noun-phrase verb* . mars
(shift)	→	*noun-phrase verb* mars .
(reduce)	→	*noun-phrase verb noun* .
(reduce)	→	*noun-phrase verb noun-phrase* .
(reduce)	→	*noun-phrase verb-phrase* .
(reduce)	→	*start*.

Wir werden im Laufe dieser Vorlesung diese Parserstrategie nicht weiter verfolgen.

5.1.6 Semantik

operationale Semantik

denotationale Semantik

5.2 Geschichte von Programmiersprachen

Auch wenn Programmiersprachen als formales System auf dem Papier entworfen werden könnten, ist die Entwicklung von Programmiersprachen stark an der zur Verfügung stehenden Hardware gekoppelt. Aber es gibt durchaus auch formale Systeme, die im weitesten Sinne als Programmiersprachen bezeichnet werden können, die vollkommen unabhängig zu irgendeiner Hardware entwickelt wurden. Die zwei einflußreichsten hiervon sind:

- die Turingmaschine[Tur36]
- der Lambdakalkül[Kle35][Bar84]

Diese beiden formalen Systeme sind in etwa der gleichen Zeit entstanden, um mathematisch dem Begriff der Berechenbarkeit auf die Spur zu kommen (fest gemacht an dem sogenannten *Entscheidungsproblem*, das, wie man den Zitaten entnehmen kann, auch in der englischsprachigen Literatur auf deutsch bezeichnet wurde). Sie sind entwickelt worden, lange bevor programmierbare elektronische Rechensysteme überhaupt zu bauen erwogen wurden. Beide Systeme sind vollkommen unterschiedlich und ermöglichen die Programmierung der gleichen Klasse mathematischer Funktionen.

Erstaunlicher Weise spiegeln sich die beiden Systeme noch heute in den großen Klassen der gängigen Programmiersprachen wieder: währen die Turingmaschine als der theoretische Urvater aller imperativen Sprachen betrachtet werden kann, legt der Lamda-Kalkül die theoretische Grundlage für funktionale Sprachen.

> Die **Turingmaschine** besteht aus einem unendlich langen Speicherband mit unendlich vielen sequentiell angeordneten Feldern. In jedem dieser Felder kann genau ein Zeichen gespeichert werden. (Man darf sich das unendliche lange Band auch als ein endliches vorstellen, es muss jedoch lang genug sein, um die aktuelle Berechnung ungehindert ausführen zu können, d.h. der Lese- und Schreibkopf darf nicht an das Ende stoßen.) einem programmgesteuerten Lese- und Schreibkopf, der sich auf dem Speicherband feldweise bewegen und die Zeichen verändern kann.
> Eine Turingmaschine modifiziert die Eingabe auf dem Band nach einem gegebenen Programm. Ist die Berechnung beendet, so befindet sich das Ergebnis auf dem Band. Es wird somit jedem Eingabewert ein Ausgabewert zugeordnet. Eine Turingmaschine muss aber nicht für alle Eingaben stoppen. In diesem Fall ist die Funktion für die Eingabe undefiniert.
> Als Ergebnis der Berechnung wird manchmal diejenige Zeichenfolge definiert, die nach dem Anhalten auf dem Band steht. Die Turingmaschine wird jedoch meistens (wie viele andere Automaten auch) für Entscheidungsprobleme eingesetzt, also für Fragen, die mit *ja* oder *nein* zu beantworten sind. Hierbei werden zum Beispiel zwei Zeichen vereinbart, wobei das eine als *ja* und das andere als *nein* interpretiert wird. Nach dem Anhalten der Turingmaschine liegt die Antwort als eines der beiden Zeichen auf dem Ausgabeband vor. Zu beachten ist dabei, dass sich jedes Problem als Entscheidungsproblem formulieren lässt, indem man fragt, ob ein bestimmter Wert eine Lösung für ein konkretes Problem ist.

Abbildung 5.4: **Wikipediaeintrag** (24. September 2006): Turingmaschine (informelle Beschreibung)

Ebenso wie in C geht die Turingmaschine von einem sequentiellen Speicher aus. Speicherzellen können gelesen und neu überschrieben werden, alles Konzepte, wie wir sie aus C kennen. Dieses Modell der Programmierung ist eng an der noch heute gebräuchlichen Hardware gebunden.

Der **Lambda-Kalkül** ist eine formale Sprache zur Untersuchung von Funktionen.

Geschichte

Der Lambda-Kalkül wurde von Alonzo Church und Stephen Kleene in den 1930er Jahren eingeführt. Church benutzte den Lambda-Kalkül, sowohl um 1936 eine negative Antwort auf das Entscheidungsproblem zu geben, als auch eine Fundierung eines logischen Systems zu finden, wie es Russells und Whiteheads Principia Mathematica zugrunde lag. Mittels des untypisierten Lambda-Kalküls kann man klar definieren, was eine berechenbare Funktion ist. Die Frage, ob zwei Lambda-Ausdrücke (s.u.) äquivalent sind, kann im Allgemeinen nicht algorithmisch entschieden werden. In seiner typisierten Form kann der Kalkül benutzt werden, um Logik höherer Stufe darzustellen. Der Lambda-Kalkül hat die Entwicklung funktionaler Programmiersprachen, die Forschung um Typsysteme von Programmiersprachen im Allgemeinen als auch moderne Teildisziplinen in der Logik wie Typtheorie wesentlich beeinflusst.
Meilensteine der Entwicklung waren im Einzelnen:

- Nach der Einführung die frühe Entdeckung, dass der Lambda-Kalkül im Sinne des Konzepts der Berechenbarkeit ebenso mächtig ist wie die Turing-Maschine oder jede moderne Programmiersprache.
- Konrad Zuse hat Ideen aus dem Lambda-Kalkül 1942 bis 1946 in seinen Plankalkül einfließen lassen.
- John McCarthy hat sie Ende der fünfziger Jahre verwendet und damit die minimalen Funktionen der Programmiersprache LISP definiert.
- Die typisierten Varianten des Lambda-Kalküls führten zu modernen Programmiersprachen wie ML oder Haskell.
- Als überaus fruchtbar erwies sich die Idee, Ausdrücke des typisierten Lambda-Kalküls zur Repräsentation von Termen einer Logik zugrunde zu legen, den Lambda-Kalkül also als Meta-Logik zu verwenden. Erstmals von Church 1940 in seiner Theory of Simple Types präsentiert, führte sie einerseits zu modernen Theorembeweisern für Logiken höherer Stufe und
- andererseits in den 70er und 80er Jahren zu Logiken mit immer mächtigeren Typsystemen, in dem sich z.B. logische Beweise an sich als Lambda-Ausdruck darstellen lassen.
- In Anlehnung an den Lambda-Kalkül wurde für die Beschreibung nebenläufiger Prozesse der Pi-Kalkül von Robin Milner in den 90er Jahren entwickelt.

Der untypisierte Lambda-Kalkül

Im Lambda-Kalkül steht jeder Ausdruck für eine Funktion mit nur einem Argument; sowohl Argumente als auch Resultate solcher Funktionen sind wiederum Funktionen. Eine Funktion kann anonym durch eine so genannte Lambda-Abstraktion definiert werden, die die Zuordnung des Arguments zum Resultat beschreibt. Zum Beispiel wird die erhöhe-um-2 Funktion $f(x) = x + 2$ im Lambda-Kalkül durch die Lambda-Abstraktion $\lambda x.x + 2$ (oder äquivalent durch $\lambda y.y + 2$; der Name des formalen Parameters ist unerheblich) beschrieben; $f(3)$ (die so genannte Funktionsanwendung) kann daher als $(\lambda x.x + 2)3$ geschrieben werden. Die Funktionsanwendung ist linksassoziativ: fxy ist syntaktisch gleichbedeutend mit $(fx)y$. Offensichtlich sollten die Ausdrücke: $(\lambda x.x + 3)((\lambda x.x + 2)0)$ und $(\lambda x.x + 3)2$ und $3 + 2$ äquivalent sein - dies motiviert die β-Kongruenzregel. Eine zweistellige Funktion kann im Lambda-Kalkül durch eine einstellige Funktion dargestellt werden, die eine einstellige Funktion als Resultat zurückgibt (siehe auch Schönfinkeln bzw. Currying). Die Funktion $f(x, y) = x - y$ kann zum Beispiel durch $\lambda x.\lambda y.x - y$ dargestellt werden. Denn mit der β-Kongruenzregel sind die Ausdrücke $(\lambda x.\lambda y.x - y)7\ 2$, $(\lambda y.7 - y)2$ und $7 - 2$ äquivalent.
...

Abbildung 5.5: **Wikipediaeintrag** (24. September 2006): Lambda-Kalkül (Ausschnitt)

Der Lambda-Kalkül hingegen kennt keine Speicherzellen. Die einzige Form von Datenhaltung sind die Parameter von Funktionen. Funktionen sind können dabei auch wieder als Argumente anderer Funktionen verwendet werden. Mit der Programmiersprache Lisp[McC60] wurden diese Konzepte direkt in eine Programmiersprache umgesetzt und damit eine lange Tradition von Programmiersprachen begründet auf deren Weg sich z.B. die Sprachen Scheme, ML, Miranda, Haskell, Clean, F#, Pizza und Scala befinden.

Man kann auch die Turingmaschine als den Versuch das Konzept der Berechenbarkeit über eine operationale Semantik zu definieren und den Lambdakalkül als den Versuch die Berechenbarkeit über eine denotationale Semantik zu definieren, auffassen.

Aber noch lange vor den theoretische formalen Systemen zur Berechenbarkeit und weit vor der Erfindung von elektronischen Rechenanlagen, gab es eine Programmiersprache: Es handelt sich um eine Art Assemblersprache, zur Programmierung der von Charles Babbage erdachten *analytical machine*. Hierbei handelte es sich um eine rein mechanische Maschine, die die in gleicher Weise areiten sollte, wie wir es von einem heutigen Computer kennen. Tatsächlich ist die Maschine zu Babbages Lebzeiten nie gebaut worden. Theoretisch wurde die Arbeit unterstützt durch Ada Lovelace.

Die **Analytical Engine**, die einen wichtigen Schritt in der Geschichte der Computer darstellt, ist der Entwurf einer mechanischen Rechenmaschine für allgemeine Anwendungen von Charles Babbage, einem britischen Professor der Mathematik. Sie wurde 1837 zum ersten Mal beschrieben, Babbage setzte die Arbeit an dem Entwurf aber bis zum Ende seines Lebens (1871) fort. Bedingt durch finanzielle und technische Probleme wurde die Analytical Engine nie gebaut. Es ist mittlerweile allerdings allgemein anerkannt, dass der Entwurf korrekt war und dass das Gerät funktioniert hätte. Vergleichbare Computer für allgemeine Anwendungen wurden erst 100 Jahre später wirklich konstruiert.
Babbage begann mit der Konstruktion seiner Difference Engine, einem mechanischen Computer, der speziell für die Lösung polynomialer Funktionen konzipiert war. Als ihm klar wurde, dass eine viel allgemeinere Bauweise möglich wäre, begann er mit der Arbeit an der Analytical Engine.
Diese sollte von einer Dampfmaschine angetrieben werden und wäre über 30 Meter lang und 10 Meter breit gewesen. Die Eingabe (Befehle und Daten) sollte über Lochkarten erfolgen, eine Methode, die in der damaligen Zeit der Steuerung mechanischer Webstühle diente. Für die Ausgabe waren ein Drucker, ein Kurvenplotter und eine Glocke geplant. Die Maschine sollte außerdem Zahlen in Lochkarten oder wahlweise Metallplatten stanzen können. Sie benutzte dezimale Fließkommaarithmetik und es war Speicher für 1000 Wörter zu 50 Dezimalstellen vorgesehen. Die Recheneinheit (*Mühle* genannt) sollte in der Lage sein, die vier Grundrechenarten durchzuführen.
Die vorgesehene Programmiersprache war ähnlich den heute verwendeten Assemblersprachen. Schleifen und bedingte Verzweigungen waren möglich. Drei verschiedene Arten von Lochkarten wurden benutzt: eine für arithmetische Operationen, eine für numerische Konstanten und eine für Lade- und Speicheroperationen, um Zahlen aus dem Speicher in die Recheneinheit und wieder zurück zu transferieren. Es gab wahrscheinlich drei separate Lochkartenleser für die drei Kartenarten.
1842 schrieb der italienische Mathematiker Menabrea, der den reisenden Babbage in Italien getroffen hatte, eine Beschreibung der Analytical Engine auf französisch, die von Lady Ada Augusta, Countess of Lovelace ins Englische übersetzt und (nach einer Anfrage von Babbage, warum sie denn nicht eine eigene Abhandlung verfasst hätte) ausführlich kommentiert wurde. Ihr Interesse für die Engine war bereits zehn Jahre früher geweckt worden. Ihre Anmerkungen zu Menabreas Beschreibung haben ihr später den Titel *erste Programmiererin* eingebracht.
1878 empfahl ein Komitee der British Association for the Advancement of Science, die Analytical Engine nicht zu bauen.
1910 berichtete Babbages Sohn, Henry P. Babbage, dass ein Teil der Recheneinheit und der Drucker gebaut und dazu benutzt worden wären, eine (fehlerhafte) Liste von Vielfachen von pi auszurechnen. Dies war nur ein kleiner Teil der ganzen Engine, nicht programmierbar und ohne Speicher.
Danach geriet die Maschine in Vergessenheit. Die Pläne für ihren Aufbau gelten jedoch als funktionsfähig, und erst um 1960 realisierten Computer die von Babbage geplante Rechengenauigkeit (50 Dezimalstellen sind ca. 166 Bit bzw. 20 Byte). Howard Hathaway Aiken, der später die elektrische Rechenmaschine Havard Mark I baute, wurde durch ihren Aufbau beeinflusst.
Aus Babbages Autobiographie: *Sobald eine Analytical Engine existiert, wird sie notwendigerweise der Wissenschaft die zukünftige Richtung weisen.*

Abbildung 5.6: **Wikipediaeintrag** (24. September 2006): analytical machine

Sie hat Kommentare zu Babbage arbeiten geschrieben, die länger, genauer und ausführlicher als sein Abhandlungen sind. In diesen Artikeln entwirft sie auch Programme zur Steuerung von Babbages Maschine, so dass ihr zu Recht der Titel als erste Programmierin zuerkannt wird.

Ada Lovelace (auch Ada Augusta Byron, Ada King oder Countess of Lovelace) (∗10. Dezember 1815 in London; †27. November 1852 in London; eigentlich Augusta Ada King Byron, Countess of Lovelace) war eine britische Mathematikerin. Sie war die Tochter Lord Byrons und Mitarbeiterin Charles Babbages.

Leben

Lord Byron hatte drei Kinder von drei Frauen, nur Ada war ehelich geboren. Doch Adas Mutter Anne Isabella (Annabella) Milbanke ließ sich von dem Dichter scheiden, als Ada erst einige Wochen alt war, und so lernte diese ihren berühmten Vater nie kennen. Ihre mathematisch interessierte Mutter ermöglichte Ada eine naturwissenschaftliche Ausbildung, in deren Verlauf sie Charles Babbage und die Mathematikerin Mary Somerville kennenlernte.
Mit 19 Jahren heiratete Ada Byron William King, 8. Baron King (1805-1893), der 1838 zum 1. Earl of Lovelace erhoben wurde. Sie gebar drei Kinder in sehr kurzen Abständen, eine ihrer Töchter war Annabelle Isabella Blunt. In ihrer Korrespondenz mit Mary Somerville schrieb sie, dass sie eine unglückliche Ehe führe, weil ihr neben Schwangerschaften und Kinderbetreuung so wenig Zeit für ihr Studium der Mathematik und ihrer zweite Leidenschaft, der Musik, blieben. Um sich abzulenken stürzte sie sich ins Gesellschaftsleben und hatte mehrere Affären. Mit großer Begeisterung wettete sie auf Pferde. Die letzten Jahre ihres Lebens, bereits mit Krebs ans Bett gefesselt, soll sie mit der Entwicklung eines mathematisch ausgefeilten *sicheren* Wettsystems verbracht haben. Ada Lovelace starb im jungen Alter von 36 Jahren.

Werk

1843 übersetzte sie die durch den italienischen Mathematiker Luigi Menebrea auf Französisch angefertigte Beschreibung von Babbages Analytical Engine ins Englische und ergänzte eigene Notizen und Überlegungen zur Maschine. Babbages Maschine wurde zu seinen Lebzeiten niemals erbaut, da ihm das britische Parlament die Finanzierung versagte. Dessen ungeachtet legte Ada Lovelace einen schriftlichen Plan vor, wie man Bernoulli-Zahlen mit der Maschine berechnen könnte. Dieser Plan brachte ihr den Ruhm ein, nicht nur die erste Programmiererin, sondern auch der erste Programmierer überhaupt gewesen zu sein. Auch deswegen wurde die Programmiersprache Ada nach ihr benannt.
Einige Biographen vertreten die Meinung, dass Ada Lovelace trotz ihrer Ausbildung einige Schwierigkeiten mit Mathematik hatte, und äußern deshalb Zweifel, ob sie Babbages Maschine wirklich vollständig verstanden hatte, oder nicht eher von Babbage als Aushängeschild zu Zwecken der Öffentlichkeitsarbeit missbraucht wurde. Diese Frage wird wohl mit den heute zur Verfügung stehenden Informationen niemals abschließend geklärt werden.
Auf jeden Fall muss man Ada Lovelace anrechnen, dass ihre Vorstellungskraft weit über die Babbages hinausging; so formulierte sie z. B. in ihren Notizen zur Analytical Engine die Idee, dass ein Nachfolger der Maschine eines Tages auch in der Lage sei, Musik zu komponieren oder Grafiken zu zeichnen.

Abbildung 5.7: **Wikipediaeintrag** (24. September 2006): Ada Lovelace

Kapitel 6

Bibliotheken

6.1 Eigene Bibliotheken bauen

Wir haben in den meisten Aufgaben bereits Wert darauf gelegt, Softwarekomponenten zu erstellen, die in verschiedenen Programmen wiederverwendet werden kann, wie z.B. die kleine Bibliothek zum Erzeugen von Bitmapdateien oder die Bibliothek zur Bearbeitung von einfachen Listen mit beliebigen Objekten. Bisher haben wir für solche Bibliotheken mit der Option `-c` der Compilers Objektdateien erzeugt, die wir dann manuell zu unserem Programm hinzugelinkt haben.

Dabei entstehen ausführbare Programme, die den kompletten Objektode der benutzten Bibliothek enthalten. Man spricht davon, dass die Bibliothek statisch zum Programm hinzugefügt wird. Dabei wird quasi der Objektcode der Bibliothek aus der `.o`-Datei mehr oder weniger in den Code des ausführbaren Programms hineinkopiert. Wenn mehrere Programme die gleiche Bibliothek benutzen, dann wird dieser Code in mehrere Programme hineinkopiert. Er existiert dann also in mehreren ausführbaren Programmen. Werden diese Pragramme gleichzeitig auf einem Betriebssystem egstartet, dann defindet sich der Code der Bibliothek mehrfach im Hauptspeicher.

6.1.1 dynamische Bibliotheken

Der eigentliche Idee einer Bibliothek ist, dass sie einmal für ganz viele Programme existiert und nicht für jedes Programm, das sie benutzen will, neu kopiert wird. Hierzu bieten die Betriebssysteme sogenannte *shared libraries* an. Dabei ist die Idee, den Objektcode für die Bibliothek nur einmal im Betriebssystem vorzuhalten. Die ausführbaren Pragramme enthalten nur einen Hinweis darauf, dass sie diese Bibliothek benötigen.

Erzeugung dynamischer Bibliotheken

```
1 gcc -fPIC -c PList.c
2   gcc -fPIC -c PListUtil.c
3   gcc -fPIC -c PListUtil2.c
4   gcc -shared -o libplist.so PList.o PListUtil.o PListUtil2.o
```

Benutzung dynamischer Bibliotheken

```
1 gcc -o PlistTest -L. -lplist PListTest.c
```

6.2 Kleines Beispiel einer GUI-Bibliothek

Mit die aufwendigsten und größten Bibliotheken beschäftigen sich mit dem Erstellen einer graphischen Benutzeroberfläche kurz *GUI*. Eine deratige Bibliothek für die Programmiersprache C ist die Bibliothek `GTK`.

In diesem Abschnitt soll exemplarisch ein minimaler Einstieg in die Bibliothek `GTK` gezeigt werden.

Jedes GUI-Programm der Bibliothek `GTK` benötigt einen Aufruf an die Initialisierung der Bibliothek, hierzu dient die Funktion `gtk_init`. Anschließend können Funktionen aufgerufen werden, die einzelne Gui-Komponenten erzeugen. Eine solche Funktion ist die Funktion `gtk_window_new`, die eine neues Fensterobjekt erzeugt. Sie kann mit einem Parameter aufgerufen werden, der die Art des Fensters genauer spezifiziert.

Folgende Konstanten können Sie für den Fenstertypen (GtkWindowType) verwenden:

- `GTK_WINDOW_TOPLEVEL` Wird meist für typische Applikationen verwendet
- `GTK_WINDOW_DIALOG` Für Fenster die für Nachrichten oder Dialoge verwendet
- `GTK_WINDOW_POPUP` Ein Fenster für Popup's

Schließlich können Komponenten mit dem Aufruf der Funktion `gtk_widget_show` sichtbar gemacht werden. Wenn auf diese Weise alle Komponenten erzeugt sind, kann die GUI-Anwendung mit dem Aufruf der Funktion `gtk_main` gestartet werden. Insgesamt läßt sich folgendes minimale Programm schreiben, welches ein kleines Fenster öffnet:

GtkWindow.c

```
1  #include <gtk/gtk.h>
2
3  int main(int argc, char **argv){
4    gtk_init(&argc, &argv);
5
6    GtkWidget* fenster = gtk_window_new(GTK_WINDOW_TOPLEVEL);
7    gtk_widget_show(fenster);
8
9    gtk_main();
10   return 0;
11 }
```

Wie man sieht, beginnen alle Funktionen der Bibliothek mit dem Präfix `gtk_`. Dies soll einiegermaßen sicherstellen, dass nicht zwei unterschiedliche Bibliotheken ein und denselben Funktionsnamen benutzen. Es ist nicht davon auszugehen, dass eine andere Bibliothek vernünftiger Weise eine Funktion mit dem Präfix `gtk_` definietr hat.

Das obige Programm ist gar nicht so leicht zu kompilieren. Versuchen wir dieses kommt es gleich schon am Anfang zu einer Fehlermeldung, weil die Headerdateien nicht gefunden werden:

```
ep@pc305-3:~/fh/c/tutor> gcc -c src/GtkWindow.c
src/GtkWindow.c:1:21: gtk/gtk.h: Datei oder Verzeichnis nicht gefunden
src/GtkWindow.c: In function ‘main’:
src/GtkWindow.c:6: error: ‘GtkWidget’ undeclared (first use in this function)
src/GtkWindow.c:6: error: (Each undeclared identifier is reported only once
src/GtkWindow.c:6: error: for each function it appears in.)
src/GtkWindow.c:6: error: ‘fenster’ undeclared (first use in this function)
src/GtkWindow.c:6: error: ‘GTK_WINDOW_TOPLEVEL’ undeclared (first use in this function)
sep@pc305-3:~/fh/c/tutor>
```

Tatsächlich beinhaltet die Bibliothek `GTK` sehr viele Headerdateien und inkludiert weitere Subbibliotheken. Diese liegen aufgrund der Anzahl nicht direkt in den Standardpfaden des Compilers. Daher sind sehr viele Optionen anzugeben, die dem Compiler sagen, wo er überall Headerdateien findet. Da im Falle der Bibliothek `GTK` diese unterschiedlichen Inkludierungsordner so viele sind, das niemand diese alle im Kopf behalten kann, bedient sich die Bibliothek eines kleinen Tools, das die Information verwaltet. Dieses Tool heißt: `gtk-config`. Dieses kann mit der Option `--cflags` nach den zusätzlichen Optionen für den Compiler gefragt werden und mit `--libs` nach den zusätzlichen Optionen für den Linkvorgang geragt werden.

Für die Kompilerflags gibt es dabei z.B. die folgende Ausgabe:

```
sep@pc305-3:~/fh/c/tutor> gtk-config --cflags
-I/opt/gnome/include -I/opt/gnome/include/gtk-1.2
-I/opt/gnome/include/glib-1.2
-I/opt/gnome/lib/glib/include -I/usr/X11R6/include
```

Für die Linkerflags die nachfolgende Ausgabe:

```
sep@pc305-3:~/fh/c/tutor> gtk-config --libs
-L/opt/gnome/lib -L/usr/X11R6/lib -lgtk -lgdk -rdynamic -lgmodule -lglib
-ldl -lXi -lXext -lX11 -lm
```

Diese Flags sind nun eigentlich dem Compiler beim Übersetzen und beim Linken jeweils anzugeben. Auf Unix-artigen Betriebssystemen gibt es eine schöne Möglichkeit auf der Kommandozeile, den Aufruf an `gtk-config` im Aufruf des Übersetzers zu integrieren. Hierzu ist der Aufruf in rückwertige einfache Anführungszeichen einzuschließen, wie in den folgenden zwei Aufrufe zu sehen ist.

```
sep@pc305-3:~/fh/c/tutor> gcc ‘gtk-config --cflags‘ -c src/GtkWindow.c
sep@pc305-3:~/fh/c/tutor> gcc -o GtkWindow ‘gtk-config --libs‘ GtkWindow.o
```

Beispielhaft soll im folgenden eine kleine eigene Gui-Komponente erzeugt werden, die ein Eingabetextfeld, ein Ausgabetextfeld und einen Knopf enthält. Beim Drücken des Knopfes soll der Text des Eingabefeldes als Eingabe einer Funtion genutzt werden. Diese Funktion wird als Funktionszeiger gespeichert. Das Ergebnis des Funktionsaufrufs wird in Ausgabefeld angezeigt.

Hierzu definieren wir eine Struktur für derartige Dialogobjekte. Diese Struktur enthalte die drei oben genannten Bestandteile, den benötigten Funktionszeiger und zusätzlich ein `GtkVBox`-Objekt, das die drei Gui-Komponenten des Dialogs zusammenfasst:

Dialogue.h

```
1 #ifndef DIALOGUE__H
2 #define DIALOGUE__H
```

```
3  #include <gtk/gtk.h>
4
5  typedef struct {
6    GtkVBox super;
7    GtkEntry* eingabe;
8    GtkButton* okKnopf;
9    GtkLabel* ausgabe;
10   char* (*f)(char*);
11 } GtkDialogue;
12
13 GtkWidget* dialogue_new(char* f(char*));
14
15 #endif
```

Es folgt die Implementierung der Klasse `Dialogue`. Im Rahmen dieser Vorlesung soll nicht weiter auf diese Implementierung eingegangen werden.

Dialogue.c

```
1  #include "Dialogue.h"
2  #include <stdlib.h>
3  #include "StringLength.h"
4
5  void reagiere(GtkObject* object){
6    GtkDialogue* dialogue = (GtkDialogue*)object;
7    char* eingabeText = gtk_entry_get_text(dialogue->eingabe);
8    char* ausgabeText = dialogue->f(eingabeText);
9    gtk_label_set_text(dialogue->ausgabe,ausgabeText);
10   free(ausgabeText);
11 }
12
13 void addToBox(GtkBox* box,GtkWidget* item){
14   gtk_box_pack_start(box,GTK_WIDGET(item), FALSE, FALSE, 0);
15 }
16
17 GtkWidget* dialogue_new(char* f(char*)){
18   GtkDialogue* dialogue = (GtkDialogue*) malloc(sizeof(GtkDialogue));
19
20   GtkVBox* tmp = GTK_VBOX(gtk_vbox_new (FALSE, 0));
21   dialogue->super =  *tmp;
22   gtk_widget_destroy(GTK_WIDGET(tmp));
23
24   dialogue->f = f;
25   dialogue->eingabe = (GtkEntry*)gtk_entry_new();
26   addToBox(GTK_BOX(dialogue), GTK_WIDGET(dialogue->eingabe));
27
28   dialogue->okKnopf = (GtkButton*)gtk_button_new_with_label("Run");
29   addToBox(GTK_BOX(dialogue), GTK_WIDGET(dialogue->okKnopf));
30
31   dialogue->ausgabe = (GtkLabel*)gtk_label_new("");
32   addToBox(GTK_BOX(dialogue), GTK_WIDGET(dialogue->ausgabe));
```

```
33
34    gtk_signal_connect_object
35      (GTK_OBJECT(dialogue->okKnopf), "clicked"
36      ,GTK_SIGNAL_FUNC(reagiere)    , GTK_OBJECT(dialogue));
37
38    return GTK_WIDGET(dialogue);
39  }
```

Die somit entstandene Komponente von einfachen Dialogfenster ist nun recht einfach zu benutzen. Hierzu braucht nur die Funktion angegeben zu werden, die ausgeführt wird, wenn auf dem Knopf gedrückt wird. In unserem Beispiel soll dabei der Eingabetext in Großbuchstaben umgewandelt werden.

TestDialogue.c

```
1   #include <gtk/gtk.h>
2   #include <stdlib.h>
3   #include "Dialogue.h"
4
5   unsigned int length(char* text){
6     unsigned int i;for (i=0;*(text+i)!='\0';i++){}return i;
7   }
8
9   #define LOWER_TO_UPPER  'A'-'a';
10
11  char charToUpper(char c){
12    if (c>='a' && c <= 'z') return c+LOWER_TO_UPPER;
13    return c;
14  }
15
16  char* toUpper(char* text){
17    char* result = malloc((length(text)+1)*sizeof(char));
18    int i=0;
19    for(;text[i]!='\0';i++) result[i]=charToUpper(text[i]);
20    result[i]='\0';
21    return result;
22  }
23
24  int main(int argc, char **argv){
25    gtk_init(&argc, &argv);
26    GtkWidget* fenster = gtk_window_new (GTK_WINDOW_TOPLEVEL);
27
28    GtkWidget* dialogue = dialogue_new(toUpper);
29    gtk_container_add (GTK_CONTAINER(fenster), dialogue);
30
31    gtk_widget_show_all(fenster);
32    gtk_main();
33    return 0;
34  }
```

Klassischer Weise sind GUI-Bibliotheken in einer objektorientierten Programmiesprache geschrieben, da sich die graphischen Komponenten sehr anschaulich als abgeschlossene eigenständige Objekte betrachten lassen. Tatsächlich hält die Bibliothek `Gtk` sich weitgehendst an einen objektorientierten Entwurf. Im nächsten Semester werden wir in GUI-Bibliohteken für C++ kennenlernen.

Kapitel 7

Schlußkapitel

Hiermit endet das erste Semester.

Ein paar einzelne kleinere Themen sind im Laufe der Vorlesung schlichtweg vergessen worden oder absichtlich übergangen worden. In diesem Kapitel werden diese Themen kurz an Beispielen angesprochen.

7.1 weitere Tricks des Präprozessors

C Programmieren sind geradezu verliebt in den Präprozessor. Er wird nicht nur zum Deklarieren von Konstanten und Inkludieren von Kopfdateien benutzt, sondern für noch recht unterschiedliche Aufgaben ge-(miß-)braucht.

7.1.1 Auskommentierung

Ein Problem der Programmiersprache C ist, dass die Kommentare nicht verschachtelt möglich sind. Die Zeichenfolge `*/` beendet einen Kommentarblock. Nach dieser Zeichenfolde werden die nachfolgenden Zeichen garantiert als Code und nicht als Kommentar betrachtet. Dieses ist unabhängig davon wie of zuvor die Zeichenfolge `/*`, die den Kommentar eröffnet, aufgetreten ist.

Dieses führt dazu, dass man nicht beliebigen Code, schnell einmal auskommentieren kann, wenn innerhalb dieses Codes bereits Kommentare stehen. Hierzu betrachte man folgendes Codefragment:

```
1  /* Dieser Code soll auskommentiert werden */
2  if (a != 0) {
3    b = 0; /* setze b auf 0 */
4  }
```

Schließt man diesen Code nun komplett in einen Kommentar ein, so endet der Kommentar sobald die Zeichenkette `*/` auftritt, also schon am Ende des ersten Kommentars.

```
/*  /* Dieser Code soll auskommentiert werden */
  if (a != 0) {
    b = 0; /* setze b auf 0 */
  }
*/
```

Mit der `if`-Anweisung des Präprozessors, kann man Blöcke unabhängig von ihrer C-Bedeutung komplett aus der Übersetzung ausschließen: `#if 0` bedeutet, das der nun folgende Text vom Präprozessor nicht zu berücksichtigen ist. Die so eingeleitete Kommentierung endet mit `#endif`.

So läßt sich der obige Code wie folgt von der Kompilierung ausschließen:

```
#if 0   /* Dieser Code soll auskommentiert werden */  if (a != 0) {    b = 0;
```

7.1.2 Parameterisierte Makros

```
#define square(x) ((x) * (x))
```

```
#define swap(x, y) { int tmp = x; x = y; y = tmp }
```

7.2 register, extern, auto Variablen

7.3 der Kommaoperator

7.4 Labels und Goto

Artikel [Dij68].

GoTo.c

```
#include <stdio.h>

double power1(double x,unsigned int e){
  double result=1;
  while (e>0){
    result=result*x;
    e=e-1;
  }
  return result;
}

double power2(double x,unsigned int e){
  double result=1;
 startWhile:
  if (e==0) goto endWhile;
```

```
16    result=result*x;
17    e=e-1;
18    goto startWhile;
19   endWhile:
20    return result;
21  }
22
23
24  int main(){
25      printf("%f\n",power2(2,10));
26      return 0;
27  }
28
```

7.4.1 Tailcall-Optimierung

tailcall1.c

```
1   #include <stdio.h>
2
3   void printNTimes(char* text,int i){
4     if (i==0) return;
5     printf("%s\n",text);
6     printNTimes(text,i-1);
7   }
8
9   void printNTimesOpt(char* text,int i){
10   start:
11    if (i==0) return;
12    printf("%s\n",text);
13
14    text=text;
15    i=i-1;
16    goto start;
17  }
18
19  int main(){
20      printNTimes("hallo",5);
21      printNTimesOpt("hallo",5);
22      return 0;
23  }
24
```

7.4.2 Ergebnis akkumulieren

tailcall2.c

```
1   #include <stdio.h>
2   #include <PList.h>
3
4   int laenge(PList xs){
```

```
5   if (isEmpty(xs)) return 0;
6   return 1+laenge(xs->tail);
7  }
8
9  int lengthAcc(int result,PList xs){
10  if (isEmpty(xs)) return result;
11  return lengthAcc(result+1,xs->tail);
12 }
13
14 int lengthAccOpt(int result,PList xs){
15  start:
16    if (isEmpty(xs)) return result;
17    result=result+1;
18    xs=xs->tail;
19    goto start;
20 }
21
22 int size(PList xs){return lengthAccOpt(0,xs);}
23
24 int main(){
25     int x=42;
26     PList xs=cons(&x,cons(&x,cons(&x,cons(&x,cons(&x,nil()))))));
27     printf("%i\n",laenge(xs));
28     printf("%i\n",lengthAcc(0,xs));
29     printf("%i\n",lengthAccOpt(0,xs));
30     printf("%i\n",size(xs));
31     return 0;
32 }
33
```

In computer science and related disciplines, *considered harmful* is a phrase popularly used in the titles of diatribes and other critical essays. It originates with Edsger Dijkstra's letter *Go To Statement Considered Harmful*,[] published in the March 1968 Communications of the ACM, in which he criticized the excessive use of the GOTO statement in programming languages of the day and advocated structured programming instead. The original title of the letter, as submitted to ACM was *A Case Against the Goto Statement,* but ACM editor, Niklaus Wirth changed the title to the now immortalized *Go To Statement Considered Harmful.*
Frank Rubin published a criticism of Dijkstra's letter in the March 1987 Communications of the ACM titled *"GOTO Considered Harmful" Considered Harmful.* Donald Moore et al. published a reply in the May 1987 CACM titled *' "GOTO Considered Harmful" Considered Harmful' Considered Harmful?.* (Dijkstra's own response to this controversy was thankfully titled *On a somewhat disappointing correspondence.*)
Some variants with replacement adjectives (*considered silly*, etc.) have been noted in hacker jargon. The phrase is also occasionally capitalized for effect.

Abbildung 7.1: **Wikipediaeintrag** (5. Februar 2007): George Boole

Anhang A

C Operatoren

Operator	Beschreibung
`()`	Gruppierung und Funktionsanwendung
`[]`	Array Index
`.`	Auswahl von Objekteigenschaft
`->`	Auswahl von Objekteigenschaft über Zeiger
`++ --`	einstelliges Increment und Decrement
`+ -`	einstelliges plus und minus
`! ~`	logische Negation, bitweises Komplement
`(type)`	Typkonversion
`*`	Dereferenzierung
`&`	Adressoperator
`sizeof`	Berechnung der Länge in Bytes
`* / %`	Multiplikation, Division, Modulo
`+ -`	Addition, Subtraktion
`<< >>`	bitweise links-/Rechts-Shift
`< <=`	kleiner- und kleiner-gleich-Relation
`> >=`	größer- und größer-gleich-Relation
`== !=`	gleich- und ungleich-Relation
`&`	bitweises Und
`^`	bitweises exklusives Oder
`|`	bitweises inklusives Oder
`&&`	logisches Und
`||`	logisches Oder
`?:`	dreistelliger Bedingungsoperator
`=`	Zuweisung
`+= -=`	Additions-/Subtraktions-Zuweisung
`*= /=`	Multiplikations-/Divisions-Zuweisung
`%= &=`	Modulo-/bitweise-Und-Zuweisung
`^= |=`	bitweise exklusive/inklusive Oder-Zuweisung
`<<= >>=`	Bitshift-Zuweisung
`,`	Folge von Ausdrücken

Anhang B

primitive Typen

In diesem Anhang findet sich ein Auszug aus dem The GNU C Reference Manual (`http://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html`) .

B.1 4.1 Integer Data Types

The integer data types range in size from at least 8 bits to at least 64 bits. You should use them for storing whole number values (and the `char` data type for storing characters). (Note that the sizes and ranges listed for these types are minimums; your particular compiler may allow for larger values.)

- `signed char`
  The 8-bit `signed char` data type can hold integer values in the range of -128 to 127.
- `unsigned char`
  The 8-bit `unsigned char` data type can hold integer values in the range of 0 to 255.
- `char`
  Depending on your system, the `char` data type is defined as equivalent to either the `signed char` or the `unsigned char` data type. By convention, you should use the `char` data type specifically for storing ASCII characters (such as 'm') or escape sequences (such as `'\n'`).
- `int`
  The 32-bit `int` data type can hold integer values in the range of -2,147,483,648 to 2,147,483,647. You may also refer to this data type as `signed int` or `signed`.
- `unsigned int`
  The 32-bit `unsigned int` data type can hold integer values in the range of 0 to 4,294,967,295. You may also refer to this data type simply as `unsigned`.
- `long int`
  The 32-bit `long int` data type can hold integer values in the range of at least -2,147,483,648 to 2,147,483,647. (Depending on your system, this data type might be 64-bit, in which case its range is identical to that of the `long long int` data type.) You may also refer to this data type as `long`, `signed long int`, or `signed long`.

- `unsigned long int`
  The 32-bit `unsigned long int` data type can hold integer values in the range of at least 0 to 4,294,967,295. (Depending on your system, this data type might be 64-bit, in which case its range is identical to that of the `unsigned long long int` data type.) You may also refer to this data type as `unsigned long`.

- `short int`
  The 16-bit `short int` data type can hold integer values in the range of -32,768 to 32,767. You may also refer to this data type as `short`, `signed short int`, or `signed short`.

- `unsigned short int`
  The 16-bit `unsigned short int` data type can hold integer values in the range of 0 to 65,535. You may also refer to this data type as `unsigned short`.

- `long long int`
  The 64-bit `long int` data type can hold integer values in the range of -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. You may also refer to this data type as `long long`, `signed long long int` or `signed long long`.

- `unsigned long long int`
  The 64-bit `unsigned long long int` data type can hold integer values in the range of at least 0 to 18,446,744,073,709,551,615. You may also refer to this data type as `unsigned long long`.

B.2 4.2 Floating Point Data Types

There are several data types that represent fractional numbers, such as 3.14159 and 2.83. The exact sizes and ranges for the floating point data types can vary from system to system; these values are stored in macro definitions in the library header file `float.h`. In this section, we include the names of the macro definitions in place of their possible values:

- `float`
  The `float` data type is the smallest of the three floating point types, if they differ in size at all. Its minimum value is stored in the `FLT_MIN`, and is supposed to be no greater than 1e(-37). Its maximum value is stored in `FLT_MAX`, and is supposed to be no less than 1e37.

- `double`
  The `double` data type is at least as large as the `float` type, and it may be larger. Its minimum value is stored in `DBL_MIN`, and its maximum value is stored in `DBL_MAX`.

- `long double`
  The `long double` data type is at least as large as the `float` type, and it may be larger. Its minimum value is stored in `LDBL_MIN`, and its maximum value is stored in `LDBL_MAX`.

All floating point data types are signed; trying to use `unsigned float`, for example, will cause a compile-time error.

Begriffsindex

Compiler, 1-2

Funktion, 2-6

Infix, 2-1

Methode, 3-2, 3-3
Modulo, 2-2

Operator, 2-1

Prozedur, 2-12

Quelltext, 1-2

Rekursion, 2-12

Seiteneffekt, 2-9

Wahrheitswert, 2-3

Programmverzeichnis

ABisZ, 4-10
AddInts, 4-51
Allreadydefined, 2-20
AlsDual, 4-15
AlsText, 2-43
Answer1, 3-5
Answer2, 3-6
Answer3, 3-6
Answer4, 3-7
Answer5, 3-8
Answer6, 3-8
apfel, 4-106
ArithTypes, 4-9
ArrowOperator, 4-32
AssignError, 2-46
AssignOperation, 2-45
AssignParam, 2-22
AuchDasLeereProgramm, 1-1

B6A1, 4-42
B6A2, 4-43
B6A3, 4-43
B6A4, 4-44
BinTree, 4-93–4-95
BMP, 4-96, 4-97
Bool, 4-25, 4-26
Boolean, 3-11
BubbleSort1, 4-57
BubbleSort2, 4-58
BubbleSort3, 4-59, 4-60
BubbleSort4, 4-61–4-64

C1, 3-9
C2, 3-9
C3, 3-9
Calloc, 4-39
CommandLineOption, 4-68
Comments, 3-14
Complex, 4-21
Condition, 2-10, 2-11
Continue, 2-40
Convert, 4-11
CountDown, 2-13
CountDownFor, 2-34
CountDownFor2, 2-47
CountDownForNonC, 2-35
CountDownRecursive, 2-31
CountDownWhile, 2-30
CountDownWhile2, 2-35
CountDownWhileBreak, 2-39
CString2, 4-65
CString3, 4-65
CString4, 4-66

D1, 3-10
D2, 3-10
D3, 3-10
DasFalscheLeereProgramm1, 1-3
DasFalscheLeereProgramm2, 1-3
DasFalscheLeereProgramm3, 1-3
DasFalscheLeereProgramm4, 1-4
DasLeereProgramm, 1-1
DeadVariable, 4-34
Diagonal, 2-39
Dialogue, 6-3, 6-4
Division, 2-2
DoubleTest, 4-12
DoWhile, 2-37
DoxygenMe, 3-15

EinsPlusEins, 2-1
Else, 2-25
ErsteAntwort, 1-6

FakIter, 2-33
Fakultaet, 2-14
Fakultaet2, 2-14
Fakultaet3, 2-29
Fib, 2-15
FirstArray, 4-48, 4-49
FirstBitmap, 4-99
FirstPointer, 4-31
Flaeche, 4-14
Fold, 4-55
FoldTest, 4-56
Free, 4-45
fun, 2-48

Geklammert, 2-2
geo, 4-100
GetAddress, 4-29
GetAddressError, 4-30
GleichUndUngleich, 2-3
Global, 2-19

GoTo, 7-2
Greater, 2-8
GtkWindow, 6-2

HalloFreunde, 1-4
HalloIllja1, 1-5
HalloIllja2, 1-5
HalloIllja3, 1-5
HalloNerver, 2-12
HalloZaehler, 2-12

If1a, 2-24
If1b, 2-25
If2, 2-25
Implication, 2-8
IncDec, 2-46
input, 2-37
IntMap, 4-54, 4-55

KnownSize, 4-49

LichtAusSpotAn, 1-6
LikeThis, 2-29
List, 4-77–4-79
ListUtil, 4-80–4-83
LocalArray, 4-50
LogikOPs, 2-4
Lokal, 2-18

Makefile, 3-13
makeHalloFreunde, 3-12
Malloc1, 4-38
Malloc2, 4-39
MathExample, 4-13
MehrAntworten, 1-6
Messwerte, 4-16–4-18
Modulo, 2-2

Nat, 4-16
Negation, 2-5
NoBrackets, 4-52
NoFree, 4-45
NoInit, 2-18
NonterminatingIteration, 2-30
NotLikeThis, 2-29
NullVar, 4-35

PList, 4-84, 4-85
PListUtil, 4-87
PListUtil2, 4-90
PListUtil2Test, 4-91
poly, 4-104
Potenz, 2-33
printNumber, 2-11
PseudoWhile, 2-36
Punkt, 4-86
Punkt1, 4-18, 4-19
Punkt2, 4-19
Punkt3, 4-20
Punkt4, 4-33
Punkt5, 4-46–4-48
PunktVorStrich, 2-1

Quadrat, 2-6
QuadratUndDoppel, 2-7

ReadRoman, 3-17
RefParam, 4-33
Repeat, 4-28
Repeat2, 4-28
ReturnArray, 4-53
Rounded, 4-12

Seiteneffekt, 2-9
Signum1, 2-5
Signum2, 2-7
SimpleSum, 2-28
SoOderSo, 2-5
StarOperator, 4-31
Static, 2-20
StringLength, 4-66
StringToUpper, 4-67
Summe1, 2-27
Summe2, 2-27
SummeDerQuadrate, 2-7
SummeFor, 2-36
SummeWhile1, 2-31
SummeWhile2, 2-32
Switch1, 2-41
Switch2, 2-42

Tage, 4-23, 4-24
Tage1, 4-24
Tage2, 4-25
tailcall1, 7-3
tailcall2, 7-3
TestBool, 4-26
TestBoolean, 3-11
TestComplex, 4-22
TestCondition, 2-10
TestDialogue, 6-5

testGeo, 4-102
TestList, 4-79
TestListUtil, 4-83
TestPList, 4-86
TestPListUtil, 4-88
testPoly, 4-105
TestReadRoman, 3-17
TestStringLength, 4-67
TestToUpper, 4-67
Twice, 4-27

Umfang, 4-14
Union1, 4-69
Union2, 4-69
Union3, 4-70, 4-71
Union4, 4-71
UnknownSize, 4-49

Var1, 2-16
Var2, 2-16
Var3, 2-17
VarQuadrat, 2-17
VoidPointer, 4-36, 4-37

WahrPlusEins, 2-4
WhileReturn, 2-40
WiederholtHallo1, 2-26
WiederholtHallo2, 2-26
WiederholtHallo3, 2-26
Word1, 4-40
Word2, 4-40
Word3, 4-41
Word4, 4-42
Word5, 4-42
WrongArrayParameter, 4-51
WrongArrayReturn, 4-52
WrongIndex, 4-50

ZeigerArith, 4-37
Zufall, 2-22
ZweiDimensionaleSchleife, 2-38

Abbildungsverzeichnis

2.1 **Wikipediaeintrag** (30. August 2006): George Boole 2-4

3.1 ddd in Aktion. 3-20

4.1 Schachtel Zeiger Darstellung einer dreielementigen Liste. 4-75
4.2 Schachtel Zeiger Darstellung der Funktionsanwendung von concat auf zwei Listen. 4-76
4.3 Schachtel-Zeiger-Darstellung nach dem zweiten Reduktionsschritt. 4-76
4.4 Schachtel-Zeiger-Darstellung nach dem zweiten Reduktionsschritt. 4-76
4.5 Schachtel Zeiger Darstellung des Ergebnisses nach der Reduktion. 4-77

5.1 Ableitungsbaum für a moon orbits mars. 5-9
5.2 Ableitungsbaum für mercury is a planet. 5-9
5.3 Ableitungsbaum für planet orbits a phoebus. 5-9
5.4 **Wikipediaeintrag** (24. September 2006): Turingmaschine (informelle Beschreibung) 5-15
5.5 **Wikipediaeintrag** (24. September 2006): Lambda-Kalkül (Ausschnitt) 5-16
5.6 **Wikipediaeintrag** (24. September 2006): analytical machine 5-18
5.7 **Wikipediaeintrag** (24. September 2006): Ada Lovelace 5-19

7.1 **Wikipediaeintrag** (5. Februar 2007): George Boole 5

Literaturverzeichnis

[Bar84] H.P. Barendregt. *The Lambda Calculus. Its Syntax and Semantics*. North-Holland, Amsterdam, New York, 1984.

[Cho56] Noam Chomsky. Three models for the description of language. *IRE Transactions of Information Theory*, 2:113–124, 1956.

[Dij68] Edsger Wybe Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968. This paper inspired scores of others, published mainly in SIGPLAN Notices up to the mid-1980s. The best-known is [Knu74].

[Kle35] Stephen Kleene. A theory of positive integers in formal logic. *American Journal of Mathematics*, 57:153–173 and 219–244, 1935.

[Knu74] Donald E. Knuth. Structured programming with **go to** statements. *ACM Computing Surveys*, 6(4):261–301, December 1974. Reprinted with revisions in *Current Trends in Programming Methodology*, Raymond T. Yeh, ed., **1** (Englewood Cliffs, NJ: Prentice-Hall, 1977), 140–194; *Classics in Software Engineering*, Edward Nash Yourdon, ed. (New York: Yourdon Press, 1979), 259–321. Reprinted with "final" revisions in [Knu92, pp. 17–89]. This paper is a response to [Dij68].

[Knu92] Donald E. Knuth. *Literate Programming*. CSLI Lecture Notes Number 27. Stanford University Center for the Study of Language and Information, Stanford, CA, USA, 1992. Distributed by the University of Chicago Press.

[KR78] B. W. Kernighan and D. M. Ritchie. *The C programming language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1978.

[McC60] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(3):184–195, 1960.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. IT Press, Cambridge, Massachusetts, 1990.

[NB60] Peter Naur and J. Backus. Report on the algorithmic language ALGOL 60. *Communications of the ACM*, 3(5):299–314, may 1960.

[Pan04] Sven Eric Panitz. Compilerbau. Skript zur Vorlesung, FH Wiesbaden, 2004.

[PHA+97] John Peterson [ed.], Kevin Hammond [ed.], Lennart Augustsson, Brian Boutel, Warren Burton, Joseph Fasel, Andrew D. Gordon, John Hughes, Paul Hudak, Thomas Johnsson, Mark Jones, Erik Meijer, Simon Peyton Jones, Alastair Reid, and Philip Wadler. Report on the programming language Haskell: A non-strict, purely functional language, Version 1.4, 1997.

[PJ03] Simon L. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[Pv97] M. J. Plasmeijer and M van Eekelen. *Concurrent Clean Language Report 1.2*. University of Nijmegen, 1997.

[T. 04] T. Bray, and al. XML 1.1. W3C Recommendation, February 2004. `http://www.w3.org/TR/xml11`.

[Tur36] A. Turing. On computable numbers, with an application to the entscheidungsproblem. In *Proceedings of the London Mathematical Society*, volume Series 2, Volume 42, 1936.

[Web02] Helmut Weber. Compiler. Skript zur Vorlesung, FH Wiesbaden, 2002.

[Wir] N. Wirth. The programming language pascal. *Acta Informatica*.