{C, C++;}

Kutil, 2006

Inhaltsverzeichnis

1 Einführung **3**
- 1.1 Geschichte . . . 3
- 1.2 Ein einfaches Programm . . . 3

2 Object-Files, Libraries und Makefiles **5**

3 Version-Control-Systeme: CVS und Subversion **10**

4 Preprozessor **16**
- 4.1 #include . . . 16
- 4.2 #define . . . 16
- 4.3 ## . . . 17
- 4.4 #if . . . 17
- 4.5 __FILE__, __LINE__ . . . 18
- 4.6 #error, #warning . . . 18
- 4.7 Line-Concatenation . . . 18
- 4.8 Kommentare . . . 19

5 C-Syntax **19**
- 5.1 Datenstrukturen . . . 19
  - 5.1.1 Primitive Datentypen . . . 20
  - 5.1.2 Pointer . . . 21
  - 5.1.3 Arrays . . . 21
  - 5.1.4 Strings . . . 23
  - 5.1.5 Strukturen . . . 23
  - 5.1.6 Enumerations . . . 25
  - 5.1.7 Typedef . . . 25
  - 5.1.8 Globale und lokale Variablen . . . 25
  - 5.1.9 Statische Variablen . . . 26
  - 5.1.10 Konstante Variablen (const) . . . 27
- 5.2 Operatoren . . . 28
- 5.3 Kontroll-Strukturen . . . 29
  - 5.3.1 Selektion . . . 29
  - 5.3.2 Iteration . . . 30
  - 5.3.3 Sprünge . . . 31
- 5.4 Funktionen . . . 32
- 5.5 Memory Allocation . . . 35

6 Input/Output in C **37**
- 6.1 Einfaches Text-I/O . . . 37
- 6.2 POSIX-File-Descriptors . . . 39
- 6.3 C-File-Handles . . . 40

6.4 Binary I/O 41

7 System-Calls **42**
7.1 Datum und Zeit 42
7.2 Errors 44

8 C++-Syntax **45**
8.1 Boolscher Typ 45
8.2 Default-Parameter 46
8.3 Namespaces 46
8.4 Klassen 47
8.5 Vererbung 49
8.6 Const-Member-Funktionen 50
8.7 Friends 51
8.8 Strings 52
8.9 References 52
8.10 Memory-Management 54
8.11 Operator Overloading 54
8.12 Casts und Typ-Identifikation 57
8.13 Inlining 58
8.14 Static Members 59
8.15 Templates 60
8.16 Exceptions 64

9 Input/Output in C++ **67**

10 Standard Template Library **70**
10.1 Container und Iteratoren 71
10.1.1 Sequenzen 71
10.1.2 Assoziative Container 74
10.2 Algorithmen 76

11 Threads **79**

12 Wo finde ich Information? **84**

1 Einführung

1.1 Geschichte

C entstand Anfang der 70er-Jahre bei AT&T Bell Labs. Dennis Ritchie entwickelte es anscheinend, um auf einem damaligen "Mini-Computer", einer PDP-7 ein Betriebssystem schreiben zu können, um darauf wiederum ein bestimmtes Computerspiel laufen zu lassen. C war stark an die Programmiersprache B (eine vereinfachte Version von BCPL) angelehnt, daher der Name.

Die Sprache war ursprünglich sehr primitiv und wurde sukzessive weiterentwickelt. 1978 veröffentlichten Dennis Ritchie und Brian Kernighan ein Buch über die Sprache. Dieses Buch diente lange als der Standard für die Sprache. Dieser Stand von C wird oft K&R-C genannt. 1989 wurde die Sprache von ANSI standardisiert und 1990 auch von ISO. Dieser Stand der Sprache heißt oft C89 oder C90. 1999 wurde eine neue Version des Standards herausgebracht (C99), mit vielen Elementen, die aus C++ übernommen wurden, das parallel entwickelt wurde. Es gibt in diesen Standards einige Details und auch ein paar umstrittene Teile, die von vielen Compilern nicht umgesetzt werden. Es gilt also bis heute, dass kein C-Compiler dem anderen gleicht. Bei der Verwendung von exotischen Konstrukten ist also Vorsicht geboten und auf die Kompatibilität mit anderen Compilern (Gnu, Intel, Microsoft, ...) zu achten.

C++ wurde ab 1979 von Bjarne Stroustrup als Erweiterung von C entwickelt. Der Name spielt auf den `++`-Operator an, der in C häufig gebraucht wird und eine Variable erhöht. Auch Stroustrup veröffentlichte ein Buch, das lange *die* Referenz für C++ war. Parallel zur Weiterentwicklung von C++ entwickelte Alexander Stepanov die Idee einer Library von generischem Programmcode, zuerst in anderen Sprachen, dann in C++. Diese Library, die Standard-Template-Library (STL) gehört HP bzw. SGI und fand Einzug in den ersten C++-Standard. Die Implementierung der STL wurde später freigegeben. C++ wurde 1998 standardisiert. Ein neuer Standard ist in Vorbereitung.

C und auch C++ stehen unter der Kritik, dass die damit geschriebenen Programme fehleranfällig sind und Sicherheitslöcher offenbaren. Auch der mit C und C++ verbundene Programmierstil wird oft als unschön bezeichnet. Auf der anderen Seite bietet C und C++ die Möglichkeit, sehr Hardware-nah und damit hochoptimiert zu programmieren. Außerdem liegt der Programmierstil in der Hand des Programmierers und die Möglichkeit unschöne Dinge zu tun wird oft auch als Freiheit interpretiert.

1.2 Ein einfaches Programm

So sieht ein minimalistisches C-Programm aus:

```
1 #include <stdio.h>
2
3 int main (int argc, char *argv[])
4 {
5   int i;
6   for (i = 0; i < argc; i ++)
7     printf ("%d: %s\n", i, argv[i]);
8   return 0;
9 }
```

Schauen wir uns das mal an, Zeile für Zeile.

Zeile 1 Hier wird mit einer *C-Preprozessor*-Anweisung das (wichtigste) *Header-File* inkludiert. Preprozessor-Anweisungen erkennt man am # zu Beginn der Zeile. Die <>-Zeichen bedeu-

ten hier, dass die Datei (auch) in den System-Verzeichnissen gesucht wird. Das ist (auf Unix-Systemen) hauptsächlich `/usr/include`. Würde man stattdessen `"stdio.h"` schreiben, würde die Datei `stdio.h` zuerst im aktuellen Verzeichnis gesucht werden. `stdio.h` deklariert alle Funktionen und sonstige Elemente für Standard-I/O. Daher auch der Name. In Header-Files werden Funktionen und Strukturen nur *deklariert*. Implementiert werden sie (meistens) in `.c`-Files. Es ist aber auch möglich, Funktionen in `.c`-Files zu deklarieren oder ganz auf die Deklaration zu verzichten.

Zeile 3 Hier wird die Funktion (oder Prozedur) `main` deklariert. Diese wird bei Programmstart aufgerufen. Dieser Funktion werden zwei Parameter übergeben, `argc` und `argv`. Zusammen stellen sie ein Array von Zeichenketten dar, das die Kommandozeilen-Argumente beinhaltet. `argc` ist vom Typ Integer (`int`) und gibt die Länge des Arrays an. `argv` ist ein Array (`[]`) von Pointern auf Zeichen (`char *`). Zeichenketten (Strings) werden in C einfach als Pointer auf das erste Zeichen der Zeichenkette implementiert (C-Strings). Das Ende der Zeichenkette wird durch ein 0-Zeichen markiert. Der erste Eintrag in `argv` übrigens ist der Programmname selbst. `argc` ist daher immer mindestens 1.

Zeile 4 und 9 Der Funktionsrumpf wird so wie andere Anweisungsblöcke in geschwungene Klammern gesetzt.

Zeile 5 Deklaration der Variable `i` vom Typ Integer (`int`). In C müssen innerhalb eines Anweisungsblocks immer zuerst alle Deklarationen gemacht werden bevor die erste Anweisung kommt. In C++ ist das nicht so.

Zeile 6 In einer *For-Schleife* wird die Variable `i` von 0 bis `argc-1` hochgezählt und in jeder Iteration die folgende Anweisung (bzw. Anweisungsblock) ausgeführt (Zeile 7).

Zeile 7 Die beliebte Funktion `printf` (aus `stdio.h`) wird aufgerufen. Sie gibt beliebig viele Argumente formatiert aus. Der erste Parameter ist ein String, der das Format angibt. Strings werden in Anführungszeichen gesetzt. `%d` bedeutet die Ausgabe eines Integer-Werts in Dezimalform. Als Wert wird der zweite Parameter, also `i`, eingesetzt. `%s` bedeutet String (`char *`). Hier wird der dritte Parameter eingesetzt, das ist hier `argv[i]`, also das `i`-te Element von `argv`. `\n` steht für Zeilenumbruch (auf Unix-Systemen LF, auf Windows-Systemen CR+LF).

Zeile 8 Mit `return` wird der Rückgabewert der Funktion `main` übergeben und die Funktion beendet. Dieser Rückgabewert muss bei `main` immer vom Typ `int` sein. Er wird an die aufrufende Shell weitergegeben. Siehe unten.

Nehmen wir an, wir haben dieses Programm in eine Datei `minimal.c` geschrieben. Wie bringen wir dieses Programm zur Ausführung? Zuerst muss es compiliert werden. Das passiert (meist) mit dem `cc` (C-Compiler). Unter Linux ist das gleichbedeutend mit `gcc` (Gnu-C-Compiler oder Gnu-Compiler-Collection). Wir führen also folgenden Befehl aus:

```
1 $ cc -o minimal minimal.c
```

`$` ist nur das Shell-Prompt. Die Option `-o` bewirkt, dass die ausführbare Datei den Namen `minimal` erhält. Ansonsten würde sie aus historischen Gründen `a.out` heißen. Eine weitere häufige Compiler-Option ist `-O3`, Compiler-Optimierung Stufe 3. Wir können das Programm jetzt ausführen.

```
$ minimal hello world
0: minimal
1: hello
2: world
```

2 Object-Files, Libraries und Makefiles

C-Programme werden in mehreren Schritten in ausführbare Programme umgewandelt. Der erste Schritt ist das Compilieren, das eigentlich Assembler-Code erzeugt (`.s`-Files). Wenn nicht anders gewünscht, wird auch der nächste Schritt, der Assembler, gleich mit ausgeführt und die `.s`-Files werden gar nicht erzeugt. Das Ergebnis des Assemblens ist ein *Object-File* (`.o`). Üblicherweise wird zu jedem `.c`-File ein zugehöriges `.o`-File erzeugt. Es enthält die Codeteile des `.c`-Files als Maschinencode inklusive Linking-Information (Position und Name der Codeteile im `.o`-File, Name von referenzierten Codeteilen).

Um z.B. ein Object-File unseres Programms `minimal.c` zu erzeugen, reicht der Befehl

```
$ cc -c minimal.c
```

Es wird die Datei `minimal.o` erzeugt.

Als nächstes kommt der Linker. Dieser verbindet ein oder mehrere Object-Files zu einem ausführbaren Programm. Sprünge zu Codeteilen, die im `.o`-File nur als Name (Symbol) verzeichnet waren, werden mit effektiven Adressen vervollständigt. Der Linker wird üblicherweise auch mittels `cc` aufgerufen. Um das ausführbare Programm `minimal` aus dem Object-File zu erzeugen, reicht der Befehl

```
$ cc -o minimal minimal.o
```

Der C-Compiler erkennt also an der Endung der Files, um welche Art von Files es sich handelt und weiß, was zu tun ist.

Der Linker kann aber nicht nur mit Object-Files etwas anfangen, sondern auch mit Libraries. Das sind Ansammlungen compilierter Codeteile, ähnlich Object-Files. Allerdings werden hier nur jene Codeteile herausgepickt und in das ausführbare Programm inkludiert, die dort wirklich gebraucht werden.

Libraries werden mit der Option `-l` inkludiert. Mit `-lxyz` wird in den Library-Verzeichnissen, das ist hauptsächlich `/usr/lib`, eine Library mit dem Namen `libxyz.a` gesucht. Um ein zusätzliches Verzeichnis zum Library-Suchpfad hinzuzufügen, verwendet man die Option `-L /foo/bar`. Eine Standardlibrary, die "C-Lib" (`libc.a`), wird automatisch dazugelinkt. Dort finden sich u.a. die Standard-I/O-Funktionen wie `printf`.

Es ist wichtig, den Unterschied zwischen Header-Files und Libraries zu kennen. Header-Files *deklarieren* Funktionen auf C-Ebene. Dadurch weiß der C-Compiler, welche Funktionen wie aufgerufen werden sollen. Die Verbindung zur Implementierung dieser Funktionen passiert erst beim Linken mit der Library oder dem Object-File. Also: Deklaration im Header-File, Source-Code im `.c`-File, Maschinen-Code im Object-File oder in der Library.

Es können Funktionen aus mehreren Header-Files in einer Library vereinigt sein. Man kann auch selbst Libraries erzeugen, indem man ein oder mehrere Object-Files mit dem Befehl `ar` zusammenfasst. Es gibt auch die Möglichkeit, Runtime-Libraries zu erzeugen. Diese haben die Endung `.so` und werden erst geladen, wenn ein Programm gestartet wird, das die Library-Funktionen benötigt. Dadurch werden Programme schlanker (kleiner) und verschiedene

Programme können bereits geladene Libraries mitbenutzen. Dieses Thema würde aber hier den Rahmen sprengen.

Wenn man größere Projekte verwaltet, hat man es mit einer Menge von Source-Files, Object-Files, Executables und möglicherweise auch Libraries zu tun. Wenn man ein Source-File ändert, ist es nicht notwendig, alle Object-Files und Executables neu zu erzeugen. Man kann sich darauf beschränken, jene neu zu übersetzen, die von dem Source-File abhängen.

Um das zu unterstützen, hat man das Tool `make` erfunden. Es verwendet ein File, das üblicherweise `Makefile` heißt, und das alle Abhängigkeiten zwischen den beteiligten Files sowie die Anweisungen enthält, um Files aus anderen zu erzeugen.

Um das zu demonstrieren, schreiben wir zuerst ein kleines Programm, das zwei Source-Files verwendet. Es soll eine komplexe Multiplikation durchführen. Dazu müssen wir zuerst eine Funktion deklarieren, die diese Multiplikation durchführt.

complex.h

```
1 void ComplexMult (double rx, double ix, double ry, double iy,
2                   double *rz, double *iz);
```

Es soll $z = x \cdot y$ berechnet werden, wobei die Parameter, die mit `r` beginnen, die Realteile sind und die mit `i` die Imaginärteile. Die Komponenten von `z` werden als Pointer übergeben (`*`), damit sie verändert werden können (Output). Diese Funktion wird nun in `complex.c` ausprogrammiert.

complex.c

```
1  #include "complex.h"
2
3  void ComplexMult (double rx, double ix, double ry, double iy,
4                    double *rz, double *iz)
5  {
6    *rz = rx * ry - ix * iy;
7    *iz = rx * iy + ix * ry;
8  }
```

Man beachte, dass das `.c`-File das `.h`-File inkludiert. Das ist zwar nicht unbedingt notwendig, aber sonst bemerkt man Unstimmigkeiten nicht. Ein Hauptprogramm `cmult.c` soll nun diese Funktion benutzen.

cmult.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "complex.h"
4
5  int main (int argc, char *argv[])
6  {
7    double rz, iz;
8    if (argc != 5)
9    { fputs ("usage: cmult rx ix ry iy\n", stderr);
10     exit (1);
11   }
12   ComplexMult (atof (argv[1]), atof (argv[2]), atof (argv[3]), atof (argv[4]),
13                &rz, &iz);
14   printf ("%f %f\n", rz, iz);
15 }
```

In Zeile 3 wird unser Header-File `complex.h` inkludiert. Das Programm soll mit 4 Kommandozeilen-Argumenten aufgerufen werden, den Real- und Imaginärteilen der zwei komplexen Zahlen. In Zeile 8–11 wird die Anzahl der Argumente überprüft und eine Fehlermeldung ausgegeben, wenn sie nicht passt. Außerdem wird dann das Programm mit `exit` brutal beendet. In Zeile 12 wird schlussendlich unsere Funktion aufgerufen. Die Kommandozeilen-Argumente werden dazu noch vom String- ins Gleitkommaformat (`double`) umgewandelt, und zwar mit der Funktion `atof` aus dem Headerfile `stdlib.h`, das in Zeile 2 inkludiert wird.

Jetzt bauen wir uns ein `Makefile`, das diese Programme übersetzt. Zuerst die ganz einfache Version.

Makefile (Version 1)

```
1 default: cmult
2
3 cmult: cmult.c complex.c
4         cc -o cmult cmult.c complex.c
```

Dieses Makefile beinhaltet zwei Regeln. Die untere übersetzt unser Programm. Eine Regel besteht aus einem Target, Prerequisites und Kommandos. Das Target steht in der ersten Zeile der Regel vor dem Doppelpunkt, in diesem Fall cmult. Es stellt einfach das File dar, das erzeugt werden soll. Die Prerequisites kommen nach dem Doppelpunkt. Es sind die Files, aus denen das Target erzeugt werden soll. Die Kommandos sind eine oder mehrere nachfolgende Zeilen, die von `make` nacheinander ausgeführt werden und das Target erzeugen sollen. Achtung: die Zeilen müssen mit einem Tabulator-Zeichen beginnen.

Wir können nun einfach `make` aufrufen und unser Programm cmult wird erzeugt.

```
1 $ make cmult
2 cc -o cmult cmult.c complex.c
3 $ cmult 1 2 3 4
4 -5.000000 10.000000
```

Zeile 1 im Makefile gibt die Default-Targets an. Der Name `default` ist allerdings nicht ausschlaggebend. Default-Target ist immer die erste Regel im Makefile.

`make` kann aber noch viel mehr. Hier Version 2 unseres Makefiles.

Makefile (Version 2)

```
1 CFLAGS=-O3
2 CC=/usr/bin/gcc
3
4 default: cmult
5
6 cmult: cmult.c complex.c
7
8 clean:
9         -rm -f *.o cmult
```

Hier kommt eine implizite Regel zur Anwendung. Da hier in Zeile 6 nur die erste Zeile der Regel vorhanden ist, sind keine Kommandos zur Erzeugung des Targets spezifiziert, nur die Prerequisites. `make` kennt aber eine implizite Regel, um C-Programme zu übersetzen. Diese Regel benutzt vorbelegte Variablen, um die Regel noch verändern zu können. Zum Beispiel wird für den C-Compiler die Variable `CC` verwendet. In Zeile 2 wird diese Variable verändert, sodass `make` nun default-mäßig `/usr/bin/gcc` als C-Compiler verwendet. Zeile 1 definiert auf gleiche Weise die Flags für den Compiler. Wir bekommen also

```
1 $ make
2 /usr/bin/gcc -O3    cmult.c complex.c   -o cmult
```

Zeile 8–9 definiert dann noch eine beliebte Regel, um alle Files eines Projekts außer den Source-Files zu löschen. Mit `make clean` kann man also ganz einfach aufräumen. Das `-` vor dem `rm`-Befehl hat den Zweck, dass `make` nicht abbricht, falls `rm` einen Fehler meldet. Das wäre nämlich der Fall, wenn ein zu löschendes File gar nicht mehr da ist.

Richtig schlau wird es aber erst, wenn nur jene C-Programme übersetzt werden, die sich seit dem letzten Compiliervorgang geändert haben. Dazu muss man die Zwischenstufe der Object-Files einführen.

Makefile (Version 3)

```
1  CFLAGS=-O3
2
3  PROGRAMS=cmult
4
5  default: $(PROGRAMS)
6
7  cmult: cmult.o complex.o
8
9  clean:
10         -rm -f *.o $(PROGRAMS)
```

Jetzt werden in Zeile 7 nur noch die Object-Files als Prerequisites angeführt. Wie diese Object-Files erzeugt werden, weiß `make` selbst. Es findet ein `.c`-File gleichen Namens im aktuellen Verzeichnis und startet daher den C-Compiler.

Wenn sich jetzt nur das File `complex.c` ändert, dann muss nur `complex.c` nach `complex.o` neu übersetzt werden und dann mit `cmult.o` zusammengelinkt werden.

```
1 $ touch complex.c
2 $ make
3 cc -O3   -c -o complex.o complex.c
4 cc   cmult.o complex.o   -o cmult
```

Der Befehl `touch` setzt das Änderungsdatum des Files `complex.c` auf "Jetzt". Da dieses Datum neuer ist als das von `complex.o`, weiß `make`, dass sich `complex.c` geändert hat und übersetzt `complex.c` neu. Danach werden die zwei Object-Files gelinkt.

Obiges Makefile verwendet außerdem die Variable `PROGRAMS`, die alle ausführbaren Programme des Projekts anführt. Das ist oft recht hilfreich. Z.B. kann sie in Zeile 5 und 9 verwendet werden. Variablen werden in Makefiles mit `$(VARIABLE)`, also mit runden Klammern abgerufen.

Es ist oft nützlich, implizite Regeln selber zu definieren. Folgendes Makefile demonstriert das.

Makefile (Version 4)

```
1  OBJECTS=complex.o
2  PROGRAMS=cmult
3
4  default: $(PROGRAMS)
5
6  %.o: %.c
7          $(CC) -O3 -c $<
```

```
8
9  cmult: cmult.o $(OBJECTS)
10         $(CC) $(LFLAGS) -o $@ $^
11
12 clean:
13         -rm -f $(OBJECTS) $(PROGRAMS)
```

In Zeile 6–7 wird die Regel zur Erzeugung von Object-Files aus C-Files umdefiniert. Das Target ist dabei ein Pattern, wobei das `%` für eine beliebiege Zeichenkette steht. Bei den Prerequisites tritt das `%` noch einmal auf. Dort wird die selbe Zeichenkette wie beim Target eingesetzt. Das Kommando muss nun das C-File übersetzen. Dazu wird der C-Complier aufgerufen, unter Verwendung der bekannten Variable `CC`. Um nun zum Namen des C-Files zu kommen, das bei einem Match der Regel mit einem echten Target eingesetzt wird, wird die Spezialvariable `$<` verwendet. Sie gibt den Namen des ersten Prerequisites an. Weitere Spezialvariablen sieht man in Zeile 9. `$@` ist der Name des Targets und `$^` gibt alle Prerequisites durch Leerzeichen getrennt an. Spezialvariablen sind auch für normale Regeln ganz nett. Sie minimieren z.B. den Aufwand bei Namensänderungen oder beim Kopieren von Regeln.

Der Überblick, welche Files von welchen abhängen, kann leicht verloren gehen, vor allem wenn auch die Abhängigkeit von Header-Files mit einbezogen wird. Wenn sich in unserem Beispiel `complex.h` ändert, sieht `make` nämlich bis jetzt keine Veranlassung, irgendetwas neu zu übersetzen. Vor allem in C++-Programmen sind aber sehr viele Codeteile auch in Header-Files zu finden und es müssen alle Programm-Files neu übersetzt werden, die ein geändertes Header-File inkludieren. Da Header-Files selbst wiederum Header-Files inkludieren (können), führt das leicht zu Haareraufen.

Es gibt daher eine gefinkelte Methode, den C-Compiler (eigentlich nur den Preprozessor) zu beauftragen, die Abhängigkeiten herauszufinden und in `make`-kompatibler Form auszugeben (mit Option `-M` oder `-MM`). Die Ausgabe wird für jedes Object-File in eine Datei gleichen Namens mit Endung `.d` geschrieben. Diese Files werden dann im Makefile inkludiert. Folgendes Makefile demonstriert diese Technik.

Makefile (Version 5)

```
1  CFLAGS=-O3
2
3  PROGRAMS=cmult
4  OBJECTS=complex.o
5
6  ALLOBJECTS=$(OBJECTS) $(patsubst %,%.o,$(PROGRAMS))
7
8  %.d: %.c
9          $(SHELL) -ec '$(CC) -MM $(CFLAGS) $< | \
10         perl -pe '"'"'s|($*\.o)[ :]*|\1 $@: |g'"'"' > $@'
11
12 default: $(PROGRAMS)
13
14 cmult: $(OBJECTS)
15
16 clean:
17         -rm -f $(ALLOBJECTS) $(ALLOBJECTS:%.o=%.d) $(PROGRAMS)
18
19 include $(ALLOBJECTS:%.o=%.d)
```

In Zeile 8–10 wird die Regel definiert, mit der ein `.d`-File aus einem `.c`-File erzeugt wird. Eigentlich wird nur `cc -MM` für das C-File (Prerequisite `$<`) aufgerufen und die Ausgabe in das `.d`-File umgelenkt (`> $@`). Das Mini-Perl-Script ist dazu da, als Target zusätzlich zum `.o`-File auch das `.d`-File in das `.d`-File einzutragen. Dadurch weiß `make` dann auch, wann es das `.d`-File neu erzeugen muss. Nach Aufruf enthält z.B. das File `cmult.d` den Inhalt

cmult.d

```
1 cmult.o cmult.d: cmult.c complex.h
```

All diese `.d`-Files werden in der letzten Zeile inkludiert. Dazu wird die Variable `ALLOBJECTS` verwendet, die alle Object-Files des Projekts enthalten soll. Durch die spezielle Referenz-Modifikation (`:%.o=%.d`) wird bei jedem Eintrag in `ALLOBJECTS` die Endung `.o` in `.d` verwandelt. Die Variable `ALLOBJECTS` selbst wird in Zeile 6 erzeugt, indem zu `$(OBJECTS)` noch die `.o`-Files der ausführbaren Programme hinzugefügt werden (durch Anhängen von `.o` an `$(PROGRAMS)` mittels `$(patsubst ...)`). Wird nun `complex.h` geändert, erkennt `make`, dass sowohl `complex.o` als auch `cmult.o` neu erzeugt werden muss.

3 Version-Control-Systeme: CVS und Subversion

Bei größeren Projekten arbeiten meistens mehrere Personen an einem Projekt. Wenn aber zwei Personen die gleiche Datei editieren, dann gibt es ein Problem. Eine Möglichkeit wäre, jede Datei vor dem Editieren mit einem Lock zu sperren und danach freizugeben. Das führt aber erfahrungsgemäß zu Locks, die zu entfernen vergessen werden. Weiters ist auch die Versionsverwaltung ein schwieriges Problem und, alte Versionen von bestimmten Files aufzustöbern, ist auch oft wichtig. All das wird von sogenannten Version-Control-Systemen ermöglicht.

Mit einem solchen System kann ein sogenanntes *Repository* eingerichtet werden, das alle Dateien eines Projekts inklusive aller vergangenen Versionen und Änderungen enthält. In diesen Repositorys wird aber nicht direkt gearbeitet sondern es werden jene Teile, die bearbeitet werden sollen, *ausgecheckt*, d.h. in ein lokales Arbeitsverzeichnis heruntergeladen. Das Repository kann sich in einem einfachen Verzeichnis befinden oder auch auf einem eigenen Server.

CVS und *Subversion* (SVN) sind zwei beliebte Version-Control-Systeme, wobei CVS das ältere, weiter verbreitete und Subversion relativ jung ist. Subversion ist CVS sehr ähnlich, bietet aber einige äußerst sinnvolle Features, die CVS nicht hat.

Im Folgenden werden die wichtigsten Aktivitäten in Subversion und CVS beschrieben.

Checkout Wenn bereits ein Repository vorhanden ist, z.B. im Verzeichnis `/tmp/repos` und dort ein Projekt mit dem Namen `xyprojekt` existiert, dann wird dieses Projekt ausgecheckt mit

```
1 $ svn checkout file:///tmp/repos/xyprojekt
```

bzw. mit

```
1 $ cvs -d /tmp/repos/ checkout xyprojekt
```

Als Ergebnis wird im aktuellen Verzeichnis ein Verzeichnis `xyprojekt` angelegt, das alle Dateien und Unterverzeichnisse des Projekts in der aktuellen Revision enthält. Dieses Verzeichnis nennt man *(lokale) Arbeitskopie* des Projekts. Jedes Verzeichnis enthält außerdem ein (verstecktes) Verzeichnis `.svn` bzw. `CVS`, das wichtige Subversion- bzw. CVS-interne Informationen enthält, vor allem die Adresse des Repositorys, sodass man später nicht immer bei allen Kommandos das Repository angeben muss.

Die folgenden Befehle führt man alle in einem Verzeichnis der lokalen Arbeitskopie aus. Manche wirken auf das Repository, schreibend oder lesend, manche sind rein lokal. Wenn nicht anders beschrieben, ist das Kommando bei CVS gleich wie bei Subversion, es ist nur `svn` durch `cvs` zu ersetzen.

Commit Man kann nun die lokale Arbeitskopie nach Belieben verändern. Das Repository wird aber erst verändert, wenn man diese Änderungen an das Repository überträgt. Das macht man mit

```
$ svn commit -m "Ein paar Änderungen durchgeführt"
```

Die Option `-m text` trägt eine kurze Meldung in die Änderungshistorie ein. Diese Kurzbeschreibungen der durchgeführten Änderungen sollten nie leer gelassen werden, weil sie beim Aufspüren der Herkunft von Dateiinhalten unverzichtbar sind.

Sollen nur einzelne Files übertragen werden, sagt man

```
$ svn commit file1 file2 ... -m "Ein paar Änderungen durchgeführt"
```

Falls ein zu übertragendes File in der Zwischenzeit im Repository von einem anderen Benutzer verändert wurde, scheitert das `commit`. Der Zustand muss erst mit `update` bereinigt werden.

Update Die Files in der lokalen Arbeitskopie können veraltet sein, wenn sie zwischenzeitlich von einem anderen Benutzer verändert wurden. Mit

```
$ svn update
```

oder

```
$ svn update file1 file2 ...
```

werden alle Files im aktuellen Verzeichnis und allen Unterverzeichnissen bzw. alle angegebenen Files auf den aktuellen Stand gebracht. Wurde ein File sowohl lokal als auch im Repository verändert, dann versucht `update` die zwei Änderungen zu vereinen. Wenn die Änderungen in verschiedenen Teilen des Files stattgefunden haben, klappt das hervorragend. Ansonsten werden in dem File die unterschiedlichen Textteile beider Files eingebaut und mit Marker-Zeilen getrennt. Es obliegt dann dem Benutzer, den Konflikt händisch zu lösen. Bei Subversion werden außerdem eigene Files mit der aktuellen lokalen Revision, der Repository-Revision und der letzten gemeinsamen Revision erzeugt. Bei Subversion muss nach dem Lösen des Konflikts der Befehl

```
$ svn resolved file
```

ausgeführt werden.

Add, Delete, Copy, Move Wenn neue Dateien und Verzeichnisse erstellt werden, fügt Subversion oder CVS diese bei einem `commit` *nicht* automatisch in das Repository ein. Dazu muss man diese Dateien erst als neu bekanntgeben. Das macht man so:

```
$ echo Neuer Inhalt >neuesfile.txt
$ mkdir neuesverzeichnis
$ echo Noch ein Inhalt >neuesverzeichnis/zweitesfile.txt
$ svn add neuesfile.txt neuesverzeichnis
```

Bei Subversion werden die Inhalte eines Verzeichnisses rekursiv ebenfalls geaddet, daher ist die Angabe von `zweitesfile.txt` bei `add` nicht notwendig. Bei CVS ist das anders. Es ist außerdem zu beachten, dass die neuen Dateien damit noch nicht an das Repository übertragen werden. Das passiert erst bei `commit`.

Das Löschen von Dateien muss man ebenfalls explizit mitteilen. Bei Subversion löscht der Befehl

```
$ svn rm neuesfile.txt
```

die Datei `neuesfile.txt` und sieht sie auch zur Entfernung aus dem Repository beim nächsten `commit` vor. Dateien verschwinden allerdings nie ganz aus dem Repository, es ist immer möglich, eine alte Version auch einer gelöschten Datei wiederherzustellen. Bei CVS muss die Datei vorher selbst gelöscht werden:

```
$ rm neuesfile.txt
$ cvs rm neuesfile.txt
```

Wenn eine Datei kopiert oder verschoben wird, dann bildet CVS das als das Einfügen einer neuen Datei bzw. das Löschen der alten ab (bei Verschieben).

```
$ mv file.txt file_a.txt
$ cvs rm file.txt
$ cvs add file_a.txt
```

Die neue Datei verliert dadurch die Änderungshistorie. Beim Suchen einer alten Revision muss man daher drauf achten, ob eine Datei vielleicht früher einen anderen Namen hatte. Wird eine neue Datei erstellt, die den gleichen Namen hat wie eine, die früher existierte, erbt die neue Datei die Änderungshistorie der alten Datei.

Bei Subversion wird die Änderungshistorie mit der Datei mitkopiert bzw. mitverschoben. Dazu muss man die `svn`-Befehle `copy` und `move` verwenden.

```
$ svn copy file1.txt file1_a.txt
$ svn move file2.txt file2_a.txt
```

Das funktioniert auch mit ganzen Verzeichnissen. Im Repository wird dabei nicht eine Kopie des ganzen Verzeichnisbaumes erstellt sondern nur eine Art Link ("cheap copy"). Weitere Änderungen werden dann für die zwei Kopien der Datei getrennt fortgeführt.

Status Die lokale Arbeitskopie kann dem Stand des Repositorys entsprechen, sie kann aber auch teilweise verändert oder auch teilweise nicht mehr auf dem aktuellen Stand sein. Um nachzusehen, welche Dateien lokal wie verändert wurden, gibt es den Befehl `status`.

```
$ svn status
D      file2.txt
A      file3.txt
M      file1.txt
```

Die Ausgabe des Befehls gibt in der ersten Spalte mit einem Buchstaben an, welchen Zustand die Datei hat. `M` heißt modifiziert, `A` neu hinzugefügt und `D` gelöscht. Dieser Befehl greift aber nicht auf das Repository zu sondern gibt nur die Änderungen aus, die seit dem letzten `update` lokal gemacht wurden. Um einen Vergleich mit dem aktuellen Stand des Repositorys zu haben, gibt man die Option `-u` an.

```
$ svn status -u
D               8   file2.txt
A      *        0   file3.txt
       *        8   file4.txt
M               8   file1.txt
```

Der * gibt an, dass sich die Datei im Repository geändert hat. Die Kombination `M` plus * würde auf einen möglichen Konflikt hindeuten. Hier wird auch die Revision ausgegeben, die die Datei im Repository hat.

CVS hat auch einen Befehl `status`, der im Prinzip das gleiche tut. Allerdings ist die Ausgabe so überladen, dass niemand den Befehl benutzt. In CVS wird stattdessen meistens der Befehl `update` zu diesem Zweck missbraucht.

Revisions Jede Datei (und bei Subversion auch jedes Verzeichnis) macht über die Zeit mehrere Veränderungen durch. Jedes Mal wenn ein neuer Zustand mit `commit` an das Repository übertragen wird, wird eine neue Revisionsnummer übergeben. Bei Subversion ist diese Revisionsnummer global, d.h. es gibt für das ganze Repository eine einzige aktuelle Revisionsnummer, die für alle Dateien gilt. Das heißt, dass die meisten Dateien über viele Revisionen hinweg nicht geändert werden. CVS führt hingegen für jede Datei eine eigene Revisionsnummer. Die Datei bekommt nur dann eine neue Revisionsnummer, wenn sie in einem `commit` inkludiert ist.

Mit dem Befehl `log` kann man sich die Änderungshistorie einer Datei anschauen.

```
$ svn log file1.txt
------------------------------------------------------------------------
r11 | rkutil | 2006-10-05 14:30:30 +0200 (Thu, 05 Oct 2006) | 1 line

file1 geändert
------------------------------------------------------------------------
r1 | rkutil | 2006-10-05 14:21:11 +0200 (Thu, 05 Oct 2006) | 1 line

initial import
------------------------------------------------------------------------
```

Alle Revisionen, in denen `file1.txt` geändert wurde, werden mit der Kurzbeschreibung des `commit`s aufgelistet. Gibt man die Option `-v` an, wird auch noch aufgelistet, welche Dateien in den einzelnen Revisionen geändert wurden.

Möchte man nun wissen, welche Änderungen genau durchgeführt wurden, verwendet man den Befehl `diff`.

```
$ svn diff -r10:11 file1.txt
--- file1.txt   (revision 10)
+++ file1.txt   (revision 11)
@@ -1,4 +1,3 @@
 hallo
 hier
-bin
-ich
+bist du
```

Hier wird der Unterschied zwischen Revision 10 und 11 abgefragt. Die Antwort ist, dass Zeile 3 und 4 (`bin` und `ich`) durch eine Zeile 3 (`bist du`) ersetzt wurden. Wenn eine Datei nur lokal verändert wurde, kann man auch den Unterschied zur Arbeitskopie anzeigen lassen, indem man einfach die Option `-r` weglässt.

Es gibt ein paar Keywords, die man anstelle von Revisionsnummern einsetzen kann. Die zwei wichtigsten sind `HEAD` und `BASE`. `HEAD` steht für die aktuelle Revision im Repository, `BASE` steht für jene Revision mit der eine Datei das letzte mal geupdatet oder commited wurde. So gibt z.B.

```
1 $ svn diff -r BASE:HEAD file1.txt
```

die Änderungen aus, die seit dem letzten `update` oder `commit` der Datei `file1.txt` *im Repository* gemacht wurden.

Neuerstellen Ein neues (leeres) Repository erstellt man mit

```
1 $ svnadmin create /tmp/repos
```

bzw.

```
1 $ cvs -d /tmp/repos init
```

Meistens soll dann ein bestehendes Projektverzeichnis in das Repository importiert werden. Angenommen, das Projektverzeichnis befindet sich in `/tmp/xyprojekt`. Folgende Anweisung importiert alle Inhalte dieses Verzeichnisses in das Repository:

```
1 $ cd /tmp
2 $ svn import xyprojekt file:///tmp/repos/xyprojekt
```

Mit CVS müsste man das so machen:

```
1 $ cd /tmp/xyprojekt
2 $ cvs -d /tmp/repos import xyprojekt none start
```

Jetzt ist aber das Verzeichnis `/tmp/xyprojekt` noch nicht mit dem Repository verbunden. Dazu müsste man es erst neu auschecken (siehe oben).

Man kann aber auch ein neues Projektverzeichnis direkt im Repository anlegen

```
1 $ svn mkdir file:///tmp/repos/xyprojekt
```

und dann dieses leere Verzeichnis auschecken. Neue Dateien dann einfach mit `add` hinzufügen und `commit`en.

Branches und Tags In einem großen Projekt ist es üblich, von Zeit zu Zeit einen Zustand des Projekts abzuzweigen (Branch) und nicht mehr weiterzuentwickeln. Die Dateien werden dann nur noch auf Bugs untersucht und getestet und eventuelle Bugs behoben. Danach werden die Datei als Release verpackt und an den Kunden weitergegeben. Am Projekt selbst kann in der Zwischenzeit parallel weitergearbeitet werden. Dazu muss in CVS und Subversion aber kein neues Repository erzeugt werden sondern das erledigen Version-Control-Systeme selbst.

In CVS verwendet man dazu den Befehl `tag`. Damit wird jeder Datei ein Tag zugewiesen. Man bearbeitet von nun an Dateien mit demselben Tag. Die Revisionsnummer wird um einen "Punkt" erweitert (z.B. `1.2`→`1.2.0`). Das alles ist sehr gewöhnungsbedürftig, vor allem weil verschiedene Dateien verschiedene Revisions-Präfixe haben können (z.B. `1.5.2` und `1.7.1`), obwohl sie zum selben Tag (Branch) gehören.

Während Branches also in CVS in einer "Extra-Dimension" verwaltet werden, löst Subversion diese Aufgabe ganz anders und sehr elegant. Weil Subversion in der Lage ist, logische Kopien (cheap copies) ganzer Verzeichnisse zu erstellen, kann man diesen Mechanismus für Branches verwenden. Man kopiert einfach mit `svn copy` das ganze Projekt und kann so getrennt weiterentwickeln. Dafür hat sich eine Standard-Verzeichnisstruktur für Subversion-Projekte eingebürgert:

```
1 xyprojekt/trunk
2 xyprojekt/branches
3 xyprojekt/branches/1.0
4 xyprojekt/branches/1.1
5 xyprojekt/tags/1.0.0
6 xyprojekt/tags/1.0.1
7 xyprojekt/tags/1.1.0
```

Das Verzeichnis `trunk` enthält immer den aktuellen Zustand des Projekts. Beabsichtigt man, ein Release zu veröffentlichen, wird das ganze Projekt, also das Verzeichnis `trunk` als Release-Branch in das Verzeichnis `branches` kopiert und bekommt dort einen Namen, der der Release-Version entspricht (z.B. `1.0`). Dort wird dann getestet. Sind die Tests abgeschlossen, wird dieser Zustand als *Tag* in das Verzeichnis `tags` kopiert und erhält noch eine Tag-Nummer (z.B. `1.0.0`). Von dort aus wird das Projekt verpackt, compiliert oder was auch immer und an den Kunden weitergegeben. Treten danach noch weitere Bugs auf, werden sie im Release-Branch behoben und ein neuer Tag erstellt (z.B. `1.0.1`).

All diese Kopiervorgänge verursachen im Repository keine Datenvervielfachung, weil Subversion immer nur die Differenzen zwischen den verschieden Versionen der Kopien abspeichert. Wird in einer Kopie eines ganzen Verzeichnisbaumes nur eine Datei verändert, beansprucht die Kopie des Verzeichnisbaumes nur den Speicherplatz dieser einen Veränderung. Es ist aber darauf zu achten, beim Auschecken nicht `xyprojekt` sondern nur z.B. `xyprojekt/trunk` oder `xyprojekt/branches/1.0` anzugeben, weil sonst für jeden Branch der Verzeichnisbaum des ganzen Projekts ins lokale Filesystem kopiert wird, was einer Datenexplosion gleichkommt.

URLs Repositorys können – wie erwähnt – nicht nur im lokalen Filesystem liegen sondern auch auf eigenen Servern. Die Server können über verschiedene Protokolle mit dem Client-Programm `svn` reden. Die Repositorys werden entsprechend dem verwendeten Protokoll mit eigenen URLs angesprochen. Subversion stellt folgende Protokolle zur Verfügung.

Protokoll	Beschreibung
`file:///`	Direkter Zugriff auf das Repository im lokalen Filesystem
`http://`	Zugriff via WebDAV-Protokoll auf einen Subversion-awaren Apache-Server
`https://`	Wie `http://`, aber mit SSL-Verschlüsselung
`svn://`	Zugriff via Subversion-eigenem Protokoll auf einen `svnserve`-Server
`svn+ssh://`	Wie `svn://`, aber durch einen SSH-Tunnel

Hilfe In Subversion ist es möglich, mit `svn help kommando` eine detaillierte Auflistung der Möglichkeiten des Kommandos `kommando` zu erhalten. In CVS muss man dazu wohl `info cvs` bemühen.

4 Preprozessor

Der C-Preprozessor ist der erste Schritt des C-Compilers bei der Übersetzung eines Programms. Genau genommen sind Preprozessor-Direktiven kein Teil der Sprache C. Der Preprozessor wird auch in anderen Programmen und Dateiarten verwendet.

4.1 #include

Die wichtigste Direktive ist `#include`. Sie hat einen Parameter, einen Dateinamen in Anführungszeichen oder <>-Zeichen geklammert. Der Inhalt der betrefflichen Datei wird einfach anstelle der `#include`-Direktive eingesetzt. Der C-Compiler compiliert dann das, was durch das (rekursive) Ersetzen aller `#include`s entsteht. Daher sollte man gegenseitige `#include`s vermeiden, also dass z.B. `graph.h` `node.h` inkludiert und `node.h` auch `graph.h`.

Wird die Datei in <>-Zeichen gesetzt, wird nur in System-Include-Verzeichnissen gesucht, also vor allem `/usr/include`, und in Verzeichnissen, die mit der Option `-I` beim Aufruf von `cc` angegeben werden. Werden Anführungszeichen gesetzt, wird sie zuerst auch im aktuellen Verzeichnis gesucht.

4.2 #define

Eine weitere wichtige Direktive ist `#define`. Damit können Makros definiert werden. Die einfachste und häufigste Art, so ein Makro zu benutzen ist für Konstanten. Beispiel:

```
1 #define PI 3.14
2
3 double CosDegree (double angle)
4 { return cos (angle * PI / 180.0); }
```

Überall im Code, wo die Zeichenkette `PI` auftaucht, wird `3.14` eingesetzt, bevor mit dem eigentlichen Compilieren begonnen wird. Da C eine eigene Syntax besitzt, Konstanten zu definieren (`const` und `enum`), ist die Verwendung von `#define` zur Definition von Konstanten zwar nicht erwünscht, trotzdem aber übliche Praxis.

Es können aber auch richtige Makros mit Parametern geschrieben werden. Beispiel:

```
1 #define DEGTORAD(x) ((x) * 0.01745)
2
3 double CosDegree (double angle)
4 { return cos (DEGTORAD (angle)); }
```

Das Makro wird wie eine Funktion aufgerufen. Der Aufruf wird aber schon zur Compilezeit ausgeführt. Allerdings wird nicht das Ergebnis der Berechnung eingesetzt, sondern nur die Zeichenkette aus der Definition mit ersetzten Parametern. In unserem Fall wäre das

```
4 { return cos (((angle) * 0.01745)); }
```

Makros stellen im Prinzip händisches *Inlining* dar. Beim Inlining fügt der Compiler zur Performance-Optimierung den Code einer Funktion anstelle des Aufrufs ein, um die Anzahl der Jumps zu verringern. Hier zwingt man den Compiler dazu. Da in C++ explizites Inlining möglich ist, besteht dort keine Notwendigkeit, Makros zu benutzen.

Hinweis: Wichtig ist, die Parameter immer in Klammer zu setzen, denn sonst würde z.B. `DEGTORAD(a + b)` zu `a + b * 0.01745`, was ein falsches Ergebnis liefern würde. Auch die äußeren Klammern sind aus ähnlichem Grund ratsam.

4.3

Eine gefinkelte Methode, `#define`s anzuwenden, ist, in Kombination mit dem Verkettungs-Operator `##` Funktions- und Variablennamen zu konstruieren.

```
1  #include <stdio.h>
2
3  #define VECOP(name,op) \
4    void vec_##name (int *a, int *b, int *c) \
5    { int i; for (i=0;i<5;i++) c[i] = a[i] op b[i]; }
6
7  VECOP(add,+)
8  VECOP(mult,*)
9
10 int main ()
11 {
12   int i, x[5] = {1, 2, 3, 4, 5}, y[5] = {5, 4, 3, 2, 1}, z[5];
13   vec_mult (x,y,z);
14   vec_add (x,z,z);
15   for (i = 0; i < 5; i ++)  printf ("%d ", z[i]);
16   putchar ('\n');
17 }
```

Hier wird in Zeile 3–5 das "Schema" einer Funktion als Makro definiert. Abhängig vom Namen einer Operation und des Operations-Symbols selbst, wird eine Funktion erzeugt, deren Name sich aus der Zeichenkette `vec_` und dem Operationsnamen `name` zusammensetzt, und die die Elemente zweier Vektoren mit dem Operationssymbol `op` verknüpft und in den dritten Vektor (`c`) schreibt. In Zeile 7–8 werden zwei solche Funktionen "instanziert", `vec_add` und `vec_mult`. Das Hauptprogramm ruft die zwei Funktionen auf und gibt das Ergebnis aus. Diese Technik kann mit Hilfe von `#include`s auf die Spitze getrieben werden, man kann sich damit eine Menge Programmierarbeit ersparen und Programme erzeugen, die niemand mehr versteht. C++ bietet allerdings *Templates* an, mit denen man meistens das gleiche auf schönere Art und Weise erreichen kann.

4.4 #if

Es gibt auch eine `#if`-Direktive, mit der man selektiv compilieren kann. Die `#if`-Direktive akzeptiert einfache Operationen und Vergleiche. Diese werden allerdings fast nie benötigt. Wichtiger ist die Variante `#ifdef`, mit der überprüft werden kann, ob ein Makro definiert ist. So kann man z.B. die Ausgabe von Debugging-Informationen durch die Definition eines Makros ein- und ausschalten.

```
123   i = i + 1;
124 #ifdef DEBUG
125   printf ("DEBUG: i = %d\n", i);
126 #endif
127   a[i] = a[i] + 1;
```

Gibt man nun beim Übersetzen mit `cc` die Option `-DDEBUG` an, dann ist das Makro `DEBUG` zwar leer, aber definiert und die Anweisung in Zeile 125 gibt den aktuellen Stand von `i` aus. Ansonsten verschwindet die Anweisung komplett aus dem Programm. Das ist natürlich schneller als die Abfrage einer globalen C-Variable mit `if`.

Häufig wird `#ifdef` auch in Header-Files verwendet, um das Problem der mehrfachen Inkludierung zu lösen. Wenn z.B. `tree.c` die Header-Files `tree.h` und `node.h` inkludiert, aber `tree.h` ebenfalls `node.h` inkludiert, werden alle Deklarationen von `node.h` zweimal durchgeführt, was wahrscheinlich zu Fehlermeldungen führt. Man könnte zwar das `#include "node.h"` aus `tree.c` entfernen, aber falls `tree.h` irgendwann doch `node.h` nicht mehr inkludiert, hat man plötzlich Fehlermeldungen in allen `.c`-Files, die `tree.h` *und* `node.h` benötigen. Abhilfe schafft folgende häufig zu findende Konstruktion in Header-Files:

node.h

```
1  #ifndef NODE_H
2  #define NODE_H
3  (... Deklarationen ...)
123 #endif
```

`#ifndef` ist einfach das Gegenteil von `#ifdef`, "if not defined". Falls also `node.h` das erste Mal inkludiert wird, ist `NODE_H` noch nicht definiert. Daher wird `NODE_H` definiert und alle Deklarationen durchgeführt. Beim zweiten Mal ist `NODE_H` schon definiert und alles bis zum `#endif` wird ausgelassen.

4.5 __FILE__, __LINE__

Zu Debugging-Zwecken sind die zwei vordefinierten Makros `__FILE__` und `__LINE__` hilfreich. Sie werden an der Stelle, an der sie benutzt werden, durch durch den Namen des aktuellen Source-Files bzw. durch die aktuelle Zeilennummer ersetzt. Im Falle einer Ausnahmesituation kann damit die Position des Problems ausgegeben werden, um das Fehlerfinden zu erleichtern. Zum Beispiel:

```
123     if (Divisor == 0)
124     {  printf ("Divisor=0 in File %s Zeile %d\n", __FILE__, __LINE__);
125        exit (1);
126     }
```

Damit lassen sich auch bequeme Debug-Makros und Assertions bauen.

4.6 #error, #warning

Mit der Direktive `#error` kann explizit der Compilation-Prozess mit Fehler beendet werden. Zum Beispiel könnte ein Programm aus bestimmten Gründen in bestimmten Betriebssystemen nicht funktionieren. Das Betriebssystem lässt sich meistens mittels `#ifdef`-Abfragen bestimmen.

```
1  #ifdef WINDOWS
2  #error This program does not work with Windows!
3  #endif
```

Ähnlich funktioniert `#warning`. Der Unterschied ist, dass bei `#warning` die Compilierung nicht abgebrochen wird.

4.7 Line-Concatenation

Manchmal müssten Zeilen sehr lang werden, z.B. bei längeren Makros, die ja in einer Zeile definiert werden müssen. Das führt dann zu unlesbaren Würsten. Daher bietet der Preprozessor

die Möglichkeit, den Zeilenumbruch mit einem Backslash-Zeichen unwirksam zu machen. Das Beispiel in Abschnitt 4.3 verwendet diese Möglichkeit. Nach dem Preprocessor-Pass stehen die drei Zeilen des Makros `VECOP` in einer Zeile. Man kann das auch für sehr lange String-Konstanten verwenden.

4.8 Kommentare

Kommentare werden in C auch vom Preprozessor entfernt. Die gebräuchlichste Art, Kommentare einzufügen ist mit `/*` und `*/`. Diese Art von Kommentaren ist zeilenübergreifend. `/*` markiert den Beginn des Kommentars, `*/` das Ende.

Mit C++ hat auch der einzeilige Kommentar mittels `//` Einzug gehalten. Tritt `//` in einer Zeile auf, gilt der Rest der Zeile als Kommentar und wird entfernt. Diese Art Kommentar ist mittlerweile auch in den meisten Versionen von C verfügbar.

Es ist zu beachten, dass `/*`-Kommentare nicht schachtelbar sind. Das heißt,

```
1 /* Äußerer /* Innerer Kommentar */ Kommentar */
```

würde nicht funktionieren, da das erste `*/` den Kommentar beendet und das zweite `Kommentar` als C-Code zu interpretieren versucht wird. Es macht allerdings keine Probleme, einen `//`-Kommentar innerhalb eines `/*` zu platzieren.

Um schnell große Teile von Code, in denen auch Kommentare vorkommen, auszukommentieren, kann man auch die `#if`-Direktive verwenden. Zum Beispiel so:

```
1 #if 0
2    a = 0;  /* wird nicht ausgeführt */
3 #endif
```

5 C-Syntax

5.1 Datenstrukturen

Im traditionellen C werden Variablen in Funktionen immer *vor* dem Programmcode deklariert. Das heißt, der Funktionskörper hat zwei Teile, einen Deklarationsteil und einen Anweisungsteil. Es ist aber auch möglich, wenn auch nicht häufig genutzt, Variablen am Anfang einer Code-Gruppe zu deklarieren, die mit `{}` eingeklammert ist. Das gilt z.B. für Schleifenkörper oder in `if`-Bedingungen usw. . Ganz modern und (noch) nicht in allen C-Varianten verfügbar ist die Deklaration einer Variablen im Kontroll-Teil einer Schleife, also z.B. `for (int i=0;i<10;i++)`, so wie es in C++ üblich ist.

Variablen werden definiert durch Angabe des Typs und des Namens, gefolgt einem Strichpunkt, in dieser Reihenfolge. Manchmal wird dieses Schema durchbrochen, z.B. bei Arrays, wo die Dimensionen *hinter* dem Variablennamen stehen.

Variablen können auch initialisiert werden, indem man hinter die Deklaration (aber noch vor dem `;`) ein `=` gefolgt vom zugewiesenen Wert angibt. Dieser zugewiesene Wert kann auch durch einen Funktionsaufruf erzeugt werden.

```
1 int main (int argc, char *argv[])
2 {
3   // Deklarationsteil:
4   int a;                  // Einfache Variable
```

```
5   int b = 2;              // Initialisierte Variable
6   double q = sqrt (2.0);  // Initialisierung mit Funktionsaufruf
7   // Anweisungsteil:
8   a = b;
9 }
```

Ein Datentyp hat eine Sonderstellung, und zwar `void`. Das ist ein leerer Datentyp. Er wird z.B. gebraucht, um Funktionen zu deklarieren, die keinen Rückgabewert liefern oder keine Parameter haben. Ansonsten hat er keine große Bedeutung. Im Folgenden werden alle möglichen Datentypen genauer erklärt.

5.1.1 Primitive Datentypen

In C gibt es einige Zahlentypen für verschiedene Wertebereiche. Allerdings sind die Wertebereiche nicht genau definiert und können von Plattform zu Plattform variieren. Zuerst die ganzzahligen Typen:

int Der gebräuchlichste Typ. Man kann davon ausgehen, dass dieser Typ auf 32-Bit-Maschinen 32 Bit umfasst. Das entspricht meistens dem `long`-Typ.

char Der Typ für Zeichen. Umfasst fast immer 8 Bits. Der Standard schreibt das allerdings nicht vor.

short Meistens 16 Bits.

long Meistens 32 Bits.

long long Meistens 64 Bits. Gibt es erst seit dem C99-Standard, davor nur als Gnu-Extension.

Die Typen `short`, `long` und `long long` sind eigentlich Abkömmlinge vom `int`-Typ und müssten `short int`, `long int`, usw. geschrieben werden. Als Abkürzung kann man das `int` aber weglassen. All diese Typen gibt es in einer `signed`- und einer `unsigned`-Version, also z.B. `unsigned int` oder `signed char`. Default ist `signed`, bis auf `char`, dort ist das Default nicht genau definiert.

Ähnlich steht es mit den Gleitkommazahlen. Hier gibt es

float 32 Bits.

double 64 Bits. Wird am häufigsten verwendet. Ist Default bei den meisten mathematischen Funktionen.

long double 96 Bits.

Gleitkommazahlen haben natürlich immer ein Vorzeichen.

Um die Länge der Datentypen abzufragen, gibt es die Anweisung `sizeof`. Sie gibt die Anzahl der Bytes an, die ein Datentyp belegt. `sizeof (int)` ergibt also meistens `4`.

Die Ober- und Untergrenzen der Datentypen sind im Header-File `limits.h` definiert. Z.B. `INT_MAX` oder `SHRT_MIN`.

Konstante für ganze Zahlen werden einfach durch Ziffernfolgen angegeben (optional mit `+` oder `-` davor), die für Gleitkommazahlen können natürlich ein `.` enthalten. Hinten kann bei beiden noch ein Exponentenzusatz stehen, `1.2e5` steht also für $1.2 \cdot 10^5$. Konstante für Characters werden in einfachen Anführungszeichen geschrieben, z.B. `'a'`. Hier gibt es noch

spezielle Kombinationen mit dem Backslash-Zeichen: `'\n'` steht für Newline (ASCII 10), `'\r'` für Carriage-Return (ASCII 13), `'\t'` für Tabulator (ASCII 9), `'\abc'` für das Zeichen mit dem *Oktal*-Code `abc`, wenn `a`, `b` und `c` Ziffern zwischen `0` und `7` sind, `'\xab'` für das Zeichen mit dem *Hexadezimal*-Code `ab`, wenn `a` und `b` Hexadezimal-Ziffern sind (`0` bis `f`), `'\0'` für ASCII 0 und `'\\'` für das Backslash-Zeichen.

5.1.2 Pointer

Pointer sind die große Stärke und die große Schwäche von C. Einerseits kann man damit ziemlich Hardware-nah und mit einigem Geschick ziemlich Performance-optimal programmieren, andererseits handelt man sich damit hässliche Abstürze, Memory-Leaks und Sicherheitslücken ein.

Ein Pointer ist im Prinzip ein Zeiger auf eine Speicherstelle. Er ist außerdem mit einem Datentyp assoziiert, den das Datum hat, das an der Speicherstelle sitzt, wo der Pointer hinzeigt. (Oder zumindest haben sollte.)

Ein Pointer wird deklariert, indem ein `*` vor den Variablennamen gesetzt wird. Das Datum, wo der Pointer hinzeigt, wird referenziert, in dem wiederum ein `*` vor den Variablennamen gesetzt wird. Es kann außerdem ein Pointer auf eine Variable erzeugt werden, indem ein `&` vor den Namen der Variablen gesetzt wird.

```
1  int a, *p;
2  p = &a;
3  *p = 2;
4  printf ("a=%d *p=%d p=%d\n", a, *p, p);
```

Hier wird eine normale `int`-Variable `a` und ein Pointer `p` deklariert. Der Pointer wird in Zeile 2 mit der Adresse von `a` beschrieben und zeigt daher jetzt auf `a`. In Zeile 3 wird an die Stelle, wo der Pointer hinzeigt, die Zahl 2 geschrieben, also in `a`. Als Ergebnis erhalten wir

```
1 a=2 *p=2 p=-1073743196
```

`a` und `*p` enthalten natürlich nicht nur die gleiche sondern sogar die selbe Zahl. Der Wert von `p` ist die Übersetzung einer Speicheradresse in einen Integer-Wert.

Ein spezieller Wert für Pointer ist `NULL`. `NULL` ist eigentlich ein Makro, das `((void*)0)` beinhaltet. `NULL` ist also ein Pointer, der an die Speicherstelle mit der Adresse `0` zeigt. Er wird häufig dafür verwendet, das Nicht-vorhanden-Sein eines Datums oder eines Parameters anzuzeigen. Abfragen wie `if(ptr==NULL)...` findet man häufig. In C++ ist `NULL` übrigens unerwünscht, man schreibt stattdessen einfach `0`.

5.1.3 Arrays

Arrays in C sind nichts als Pointer. Der Pointer zeigt auf den ersten Eintrag des Arrays. Um die Länge des Arrays muss man sich selbst kümmern. C erkennt auch keine Index-Bereichsüberschreitung, d.h. wenn man den Index zu groß werden lässt schreibt man irgendwo im Speicher herum und verursacht hässliche Abstürze.

Deklariert wird ein Array, indem man `[]` hinter den Variablennamen schreibt. Gibt man außerdem eine Zahl in den eckigen Klammern an, so wird ein Speicherbereich alloziert, der einem Array der Länge dieser Zahl entspricht und die Array-Variable zeigt auf diesen Bereich. Ansonsten muss man sich um diese Allokation selbst kümmern. Dazu später. Man kann ein Array auch initialisieren, indem man seine Elemente in `{}`-Klammern und durch Beistrich getrennt anführt. In diesem Fall kann die Längenangabe entfallen, muss aber nicht.

```
1  int a[5];
2  int b[] = {1, 2, 3, 4, 5};
3  int i;
4  for (i = 0; i < 5; i ++)
5    a[i] = b[i];
```

In Zeile 1 wird ein Integer-Array mit Längenangabe deklariert. In Zeile 2 durch Initialisierung. In Zeile 4–5 wird in einer Schleife auf beide Arrays zugegriffen. `a[i]` steht für das `i`-te Element des Arrays `a`, wobei der Index `i` bei 0 zu zählen beginnt. Man darf den Index nur bis Länge minus 1 laufen lassen, also hier bis 4.

Um zu zeigen, dass Arrays wirklich nur Pointer sind, und zu welchen Syntax-Eskapaden C fähig ist, hier noch einmal das (fast) gleiche Programm:

```
1  int *a = (int *) malloc (5 * sizeof (int));
2  int *b = (int []) {1, 2, 3, 4, 5};
3  int i, *pb = b;
4  for (i = 0; i < 5; i ++)
5    *(a+i) = *pb ++;
```

In Zeile 1 wird das Array `a` angelegt. Da es jetzt wirklich nur noch ein Pointer ist, müssen wir die Allokation des Speichers für die Elemente selbst übernehmen. Die Funktion `malloc` alloziert eine bestimmte Anzahl von Bytes. Da wir fünf Integer-Elemente haben, müssen wir fünf mal so viele Bytes, wie ein Integer hat, allozieren. `malloc` wird später noch genauer besprochen.

In Zeile 2 wird das Array `b` angelegt. Damit die Initialisierung funktioniert, ist ein *Cast* notwendig (`(int [])`), um C mitzuteilen, dass das, was danach kommt, ein Array darstellen soll. Das Ergebnis ist dann vom Typ Integer-Array und kann daher dem `b`-Pointer zugewiesen werden.

In Zeile 5–6 wird wieder das Array kopiert. Auf `a` wird hier zugegriffen, indem der Pointer um `i` Elemente erhöht wird (durch `+`) und dann der Pointer dereferenziert wird (durch `*`). Die Addition eines Pointers und eines Integers `i` ergibt in C einen Pointer, der um `i` Elemente verschoben ist. Das heißt, das Ergebnis hängt vom Typ des Arrays ab. Bei einem Character-Pointer `p` würde `p+3` den Pointer um den Wert 3 erhöhen, bei einem Integer-Pointer hingegen um 12, weil ein Integer 4 Bytes groß ist.

Auf `b` wird auf ähnliche Weise zugegriffen. Und zwar wird hier ein Pointer `pb` zuerst in Zeile 3 von `b` kopiert. Er zeigt also am Anfang ebenfalls auf den Beginn des Arrays `b`. In Zeile 5 wird auf die Elemente von `b` über `*pb` zugegriffen, wobei der Pointer `pb` *nach* jedem Zugriff mittels `++` um eins erhöht wird. Auch hier wird auf den Typ Rücksicht genommen, das heißt eigentlich um 4 Bytes erhöht. Man beachte, dass `++` stärker bindet als `*`, daher wird `pb` erhöht und nicht das Element von `b`, auf das `*pb` zeigt. Diese Konstruktion sieht man häufig und heißt "Pointer-Increment".

Zweidimensionale Arrays programmiert man als Array von Arrays. Das entspricht dann einem Pointer, der auf ein Array aus Pointern zeigt, die wiederum auf die Elemente des 2D-Arrays zeigen. Jede Zeile ist also ein 1D-Array, das durch einen Pointer auf das erste Element repräsentiert ist; all diese Pointer stehen wiederum in einem eigenen Array.

```
1  char s[10][20];  // 10 Strings der Länge 20
2  int *a[15];      // 15 unallozierte int-Arrays
3  a[2] = (int *) malloc (30 * sizeof (int));  // 2. int-Array alloziert
4  a[2][25] = s[2][15];   // 2D-Array lesen und schreiben
```

5.1.4 Strings

C-Strings sind wiederum nichts als Character-Arrays. Allerdings gibt es hier die Konvention, dass das Array mit einem 0-Zeichen abgeschlossen werden muss (ASCII-0).

```
1  char *a = "hallo";
2  char b[6];
3  char *pb;
4  strcpy (b, a);
5  for (pb = b; *pb != 0; pb ++)
6    printf ("%c\n", *pb);
```

In Zeile 1 wird ein Character-Pointer mit einem String initialisiert. `a` zeigt also auf das `h` von `hallo`. Das abschließende 0-Zeichen ist in dem String inkludiert. In Zeile 4 wird String `a` nach String `b` kopiert mittels der Funktion `strcpy`. Allerlei String-Funktionen sind in `string.h` definiert. Siehe auch die Manpage `man string`. *Wichtig:* `b` muss mindestens mit der Länge 6 angelegt werden, da `"hallo"` inklusive der abschließenden Null 6 Zeichen lang ist.

In Zeile 5–6 wird der String Zeichen für Zeichen ausgegeben, wobei die Schleife mittels Pointer-Increment implementiert ist und abbricht, sobald `*pb == 0` ist, also wenn das Ende des Strings erreicht ist.

Auch bei Strings kann man die Spezial-Characters (z.B. `\n`) verwenden.

```
1 char *langerstring = "Vor einem großen Walde wohnte ein armer Holzhacker \
2 mit seiner Frau und seinen zwei Kindern; das Bübchen hieß Hänsel";
3 char *nocheinstring = "und das Mädchen Gretel. Er hatte wenig zu beißen "
4                       "und zu brechen, und einmal, als große Teuerung";
5 char *falscherstring = "ins Land kam,
6 konnte er das tägliche Brot nicht mehr schaffen."  // ERROR!
7 char *zweizeiligerstring = "ins Land kam, \n"
8       "konnte er das tägliche Brot nicht mehr schaffen."
```

Lange String-Konstanten können auf zwei Arten geschrieben werden (abgesehen von überlangen Code-Zeilen). Einerseits kann man den Preprozessor verwenden, um einen Zeilenumbruch mit `\` zu entfernen (Zeile 1–2). Andererseits kann man auch einfach zwei String-Konstanten hintereinanderstellen (Zeile 3–4). Diese fügt der Compiler aneinander. Will man in den String einen Zeilenumbruch einbauen, so konnte man früher einfach einen mehrzeiligen String verwenden (Zeile 5–6). Das ist heute verboten. Man muss jetzt den Zeilenumbruch selbst in den String einbauen (`\n`) und in üblicher Weise einen langen String erzeugen (Zeile 7–8).

5.1.5 Strukturen

Zusammengesetzte Datentypen erzeugt man in C mangels Klassen mittels `struct`.

```
1  struct
2  { int Id;
3    char Name[10];
4  } User;
5  User.Id = 15;
6  strcpy (User.Name, "Zyprian");
```

Hier wird eine Variable `User` deklariert, die sich aus einem Integer namens `Id` und einem String namens `Name` zusammensetzt. Diese Members werden über `.` referenziert, ähnlich wie Members einer Klasse. In Zeile 5–6 wird auf die Members der Variable User zugegriffen.

Eine Struktur kann auch für sich deklariert werden, ohne eine Variable zu deklarieren. Dabei ist es sinnvoll, der Struktur einen Namen zu geben. Der Name kommt direkt nach `struct`.

```
1  struct UserStruct
2  { int Id;
3    char Name[10];
4  };
5  struct UserStruct UserA;
6  struct UserStruct UserB;
7  UserA.Id = 15;
8  strcpy (UserA.Name, "Zyprian");
9  UserB = UserA;
10 printf ("%d %s\n", UserB.Id, UserB.Name);
```

Variablen können dann unter Verwendung des Namens der Struktur deklariert werden. Siehe Zeile 5–6. Variablen vom Typ einer Struktur können einfach zugewiesen werden, wie in Zeile 9. Die Ausgabe des Programms ist natürlich `15 Zyprian`.

Wenn ein Pointer `P` auf eine Struktur gegeben ist, kann auf die Members z.B. mittels `(*P).Id` zugegriffen werden. Dafür gibt es allerdings eine abgekürzte Schreibweise, und zwar `->`.

```
11 struct UserStruct *P = &UserB;
12 P->Id = 16;
```

Eine zusätzliche Datenart, die häufig in Zusammenhang mit Strukturen verwendet wird, ist die *Union*. Sie wird ähnlich deklariert wie eine Struktur. Allerdings *überlagern* sich die Members. Das macht dann Sinn, wenn man eine Variable (oder eben einen Teil einer Struktur) mal so und mal so verwenden möchte.

```
1  union
2  { signed char s;
3    unsigned char u;
4  } x;
5  x.s = -1;
6  printf ("%d\n", x.u);
```

Hier wird eine Union `x` deklariert mit zwei Members, einem signed und einem unsigned Character. Wird nun wie in Zeile 5 der signed Member mit einer negativen Zahl beschrieben und in Zeile 6 der unsigned Member ausgelesen, so wird einfach die Binärdarstellung der negativen Zahl als Binärdarstellung eine positiven uminterpretiert. Das Ergebnis hier ist 255. Durch Verschachteln von Strukturen und Unions können natürlich sinnvollere Anwendungen gefunden werden.

Mit Hilfe von Strukturen ist es möglich, beinahe Objekt-orientiert zu programmieren. Dazu wird eine Struktur mit Namen global definiert und diverse Funktionen zur Verfügung gestellt, um Variablen (Instanzen) der Struktur zu manipulieren. Diese Funktionen werden dann allerdings nach dem Schema `Manip (UserA)` statt `UserA.Manip ()` aufgerufen. Mit Unions können auch Klassenerweiterungen imitiert werden.

5.1.6 Enumerations

Mit `enum` können Aufzählungstypen definiert werden, deren Werte wählbare Namen bekommen.

```
1  enum UserType {UT_admin, UT_staff, UT_student = 5, UT_guest};
2  printf ("%d %d %d %d\n", UT_admin, UT_staff, UT_student, UT_guest);
```

Hier wird Aufzählungstyp `UserType` mit vier Werten definiert. `enum`-Typen werden immer als `int` abgebildet, das heißt, jede Variable von diesem Typ ist einfach ein `int`. Die Namen für die Werte sind einfach `int`-Konstanten und überall als solche verfügbar und nicht auf den Einsatz mit dem Aufzählungstypen beschränkt. Daher funktioniert auch der `printf`-Befehl in Zeile 2. Die Ausgabe des Programms ist `0 1 5 6`, das heißt, dass `enum` bei 0 beginnt, die Namen durchzunummerieren. Der dritte Name wird explizit dem Wert `5` zugewiesen. Danach fährt `enum` fort, ab dieser Zahl weiterzunummerieren.

Die Tatsache, dass der `enum`-Typ in C nicht viel zu bieten hat im Vergleich zu Aufzählungstypen in anderen Sprachen, führt dazu, dass `enum` nicht sehr häufig verwendet wird. Vor allem die "Verseuchung des Namensraumes", d.h. dass jeder Name des Typs Variablennamen oder Funktionsnamen überlagern kann, weil er nicht auf den Typ beschränkt ist, ist ein großer Nachteil.

5.1.7 Typedef

Ein kleiner Nachteil bei benannten `struct`s, `union`s und `enum`s ist, dass bei der Deklaration von Variablen immer auch das Schlüsselwort `struct`, usw. mit angegeben werden muss. Aber auch Arrays mit bestimmter Länge oder andere Typkonstruktionen müssten bei Variablendeklarationen immer wiederholt werden.

C bietet daher die Möglichkeit, eigene Typen zu definieren. Dazu verwendet man das Schlüsselwort `typedef` gefolgt von der gleichen Syntax wie bei einer Variablendeklaration, nur dass statt der Variable ein neuer Typ deklariert wird.

```
1  typedef int int20[20];
2  typedef struct {int a; int b;} abt;
3  int20 arr;
4  abt ab;
5  arr[10] = 5;
6  ab.a = 7;
```

Hier werden zwei neue Typen erzeugt. `int20` ist ein Array von 20 Integers und `abt` ist eine (unbenannte) Struktur. Variablen können dann mit Hilfe der neuen Typen ganz einfach (Zeile 3–4) deklariert und in üblicher Weise benutzt werden (Zeile 5–6);

`typedef`s können auch benutzt werden, um Sourcecode leichter adaptierbar zu machen. So könnte z.B. ein Algorithmus, der mit `float` und `double` gleichermaßen funktioniert, einen eigenen Typ verwenden, z.B. `alg_float`. Dieser Typ kann dann an einer zentralen Stelle, z.B. im Header-File, als `typedef float alg_float;` oder als `typedef double alg_float;` definiert werden. Damit kann der Algorithmus leicht umkonfiguriert werden.

5.1.8 Globale und lokale Variablen

Dinge, die in C deklariert oder definiert werden, können im Prinzip *global* oder *lokal* sein. Lokal heißt, dass sie innerhalb einer Funktion oder eines anderen Konstruktes deklariert werden. Global heißt außerhalb.

Auf globale Variablen kann man überall im Programm zugreifen, solange die Deklaration sichtbar ist, d.h. vor der Stelle positioniert ist, an der auf die Variable zugegriffen wird. Lokale Variablen sind nur innerhalb der Funktion verfügbar, in der sie deklariert wurden.

Globale Variablen existieren nur einmal im Programm. Lokale Variablen können mehrfach und sogar gleichzeitig existieren, wenn die Funktion rekursiv ist. Dann existiert für jede Instanz der Funktion eine Instanz der lokalen Variable.

Globale Objekte dürfen beim Linken nur in *einem* Object-File vorkommen. Das bedeutet, dass globale Objekte in Header-File nur deklariert, nicht aber definiert werden sollten, weil sonst das globale Objekt in jedem Object-File enthalten ist, das unter Inkludierung des Header-Files erzeugt wurde. Funktionen werden rein deklariert, indem man den Funktionskörper weglässt und einen Strichpunkt anhängt. Dazu mehr in Abschnitt 5.4. Variablen werden rein deklariert, indem man das Keyword `extern` davorschreibt. Beispiel:

modus.h

```
1 extern int ProgrammModus;
2 void SetModus (int modus);
```

modus.c

```
1 int ProgrammModus;
2
3 void SetModus (int modus)
4 { ProgrammModus = modus; }
```

5.1.9 Statische Variablen

Statische Variablen sind irgendwo zwischen globalen und lokalen Variablen: sie existieren zwar nur einmal im Programm, sind aber nur von einer Funktion aus sichtbar.

Statische Variablen sind nützlich, wenn etwas nur einmal im Programm erledigt werden muss, eine Initialisierung oder ähnliches, oder wenn die Funktionsaufrufe mitprotokolliert werden sollen. Folgende Funktion berechnet die Fakultät einer Zahl und hinterlegt das zuletzt berechnete Ergebnis in einer statischen Variablen. Das Ergebnis kann wiederverwendet werden, wenn als nächstes eine höhere Fakultät berechnet werden soll.

```
1  int Faculty (int i)
2  {
3    static int par = 0;
4    static int val = 1;
5    int ret;
6    if (par == i) return val;
7    if   (i == 0)
8         ret = 1;
9    else ret = i * Faculty (i - 1);
10   par = i; val = ret;
11   return ret;
12 }
13
14 int main ()
15 {
16   int i;
17   for (i = 0; i < 13; i ++)  printf ("%d: %d\n", i, Faculty (i));
18 }
```

Die statische Variable `par` in Zeile 3 hält fest, für welche Zahl zuletzt die Fakultät berechnet worden ist, die Variable `val` hält das Ergebnis fest. Falls nun die aktuell zu berechnende Fakultät von `i` schon gespeichert ist, wird diese verwendet (Zeile 6), ansonsten wird die Fakultät in üblicher Weise rekursiv berechnet. Die Rekursion bricht ab, wenn `i==0` erreicht wird oder ein gespeichertes Ergebnis gefunden wird. Bevor das neue Ergebnis ausgegeben wird, wird es in Zeile 10 in den statischen Variablen gespeichert. In der Schleife in Zeile 17 wird dann in jeder Iteration für eine Fakultät nur *eine* Multiplikation benötigt.

5.1.10 Konstante Variablen (const)

Es ist möglich, in C Variablen als konstant zu definieren, indem man das Keyword `const` vor oder hinter den Typ schreibt.

```
1 const double pi = 3.14159;
2 int const answer = 42;
```

Solche Variablen müssen bei der Definition initialisiert werden und dürfen nicht mehr verändert werden. Das ermöglicht dem Compiler, Immediate-Werte im Maschinencode zu verwenden, wenn das schneller ist. Dadurch ist man eigentlich nicht auf die hässlichen Makros angewiesen.

Eine weitere, interessante Verwendung von `const` ist bei Pointern. Hier kann man wählen, ob der Pointer selbst unveränderbar sein soll oder das Datum, auf das er zeigt, oder beides. Dabei ist oft verwirrend, welches `const` für was steht. Man sollte dabei die Regel befolgen, `const` immer direkt *hinter* das zu schreiben, das unveränderbar sein soll.

```
1   int x;
2   const int *p = &x;
3   int const *q = &x;
4   int * const r = &x;
5   int const * const s = &x;
6   *p = 1;  // Fehler
7   p = r;   // ok
8   *q = 1;  // Fehler
9   q = r;   // ok
10  *r = 1;  // ok
11  r = p;   // Fehler
12  *s = 1;  // Fehler
13  s = p;   // Fehler
```

In Zeile 2 und 3 wird ein Pointer auf ein konstantes `int` deklariert. Zeile 3 ist dabei die konsistentere aber seltenere Schreibweise. Konstant ist immer das, was *links* von `const` steht, in diesem Fall das `int`. Deshalb sind Zeile 6 und 8 unerlaubt und Zeile 7 und 9 ok. In Zeile 4 wird ein konstanter Pointer auf ein veränderbares `int` deklariert. Deshalb ist Zeile 10 ok aber Zeile 11 nicht. In Zeile 5 ist sowohl Pointer als auch das referenzierte `int` konstant. Zeile 12 und 13 sind daher beide unerlaubt.

Dieses Konstrukt kann man benutzen, um zu verhindern, dass Funktionen, denen Arrays übergeben werden, diese Arrays verändern können. Arrays werden ja immer als Pointer und daher per Reference übergeben. Sie könnten daher immer verändert werden. Indem man den Pointer in der Parameterliste mit `const` modifiziert, kann das ausgeschlossen werden.

```
1 void Drucke (char const *str)
2 {
```

```
3   str[2] = 27;  // Fehler, ist nicht erlaubt
4 }
```

5.2 Operatoren

Die primitiven Datentypen können über mehrere Operatoren miteinander verknüpft werden. Dazu gehören natürlich die Grundrechenarten `+`, `-`, `*` und `/` mit den bekannten Bindungsstärken, sowie die Modulo-Operation `%`.

Der Increment-Operator `++` und der Decrement-Operator `--` erhöht bzw. vermindert eine Integer-Variable oder einen Pointer um "eins". Tritt der Operator innerhalb eines größeren Ausdrucks auf, wird die Variable bei `++a` *vor* und bei `a++` *nach* der Evaluation des Ausdrucks erhöht, bzw. bei `--` vermindert. Achtung: Zweimaliges Anwenden auf die gleiche Variable innerhalb eines Ausdrucks führt zu undefiniertem Verhalten (z.B. ist `b=a[i++]+a[i++]` inkorrekt).

Vergleiche von Zahlen können mit `<`, `>`, `<=`, `>=`, `==` (ist gleich) und `!=` (ungleich) gemacht werden. Achtung: Der `==` Vergleich ist nicht mit der Zuweisung `=` zu verwechseln. Da eine Zuweisung auch einen Wert liefert, kann in C auch eine Zuweisung als Wahrheitswert interpretiert werden und compiliert problemlos. Der Fehler fällt daher nicht so leicht auf. Lösung: alle möglichen Warning-Flags des Compilers einschalten.

Das Ergebnis dieser Operationen ist vom Typ `int`, weil es in C keinen Boolschen Typ gibt. Falsch entspricht einer `0`, jeder andere Wert gilt als wahr. Wahrheitswerte können mit `&&` (und) und `||` (oder) verknüpft und mit `!` negiert werden. Man beachte, dass laut C-Standard `&&` so definiert ist, dass die zweite Bedingung gar nicht mehr evaluiert wird, wenn die erste falsch ergibt, weil dann der Gesamtwert sowieso nicht mehr wahr werden kann. Bei `||` ist es ähnlich: die zweite Bedingung wird nicht evaluiert, wenn die erste wahr ist, weil damit der gesamte Ausdruck wahr ist. Eventuelle Increment-Operatoren in der zweiten Bedingung werden dann nicht ausgeführt (Buggefahr!).

Äquivalent dazu gibt es die Bit-Operatoren `&` (und), `|` (oder) und `~` (nicht). Im Gegensatz zu den logischen Operatoren wird die Verknüpfung hier bitweise durchgeführt, also jeweils das n-te Bit des einen Operanden mit dem n-ten Bit des anderen verknüpft. `3&6` ergibt also `2` (binär `011` und `110` ergibt `010`), während `3&&6` "wahr und wahr" entspricht und daher einfach "wahr" ergibt, also meist `1`. Es gibt hier auch noch den XOR-Operator `^`.

Die Bit-Shift-Operatoren `<<` und `>>` verschieben die Bits eines Integers nach links bzw. nach rechts. Das entspricht einer Multiplikation mit einer 2-er-Potenz. So ergibt `3<<2` den Wert $3 \cdot 2^2 = 12$ und `6>>2` den Wert $\lfloor 6 \cdot 2^{-2} \rfloor = 1$. Das Wegfallen der hinausgeschobenen Bits bei einem Rechts-Shift führt also zu Abrundung. Achtung: die Bindungsstärke der Shift-Operatoren ist sehr gering. So ergibt `1<<2+1<<3` nicht den erwarteten Wert $2^2+2^3 = 12$ sondern $2^{2+1} \cdot 2^3 = 64$. Es ist also oft günstig, nicht mit Klammern zu sparen. `(1<<2)+(1<<3)` liefert das gewünschte Ergebnis.

Zuweisungen können mit `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `<<`, `>>` verbunden werden, um eine verkürzte Schreibweise zu erreichen. Beispiel: `a+=2` entspricht der Anweisung `a=a+2`.

Zuweisungen liefern, wie bereits erwähnt, einen Ergebniswert, und zwar den zugewiesenen Wert. Dadurch werden Anweisungen wie `a=b=c=0` möglich, wo alle drei Variablen `a`, `b` und `c` auf `0` gesetzt werden.

Weitere Operatoren sind noch die Member-Operatoren `.` und `->`, sowie die Pointer-Operatoren `*` und `&`, die schon besprochen wurden.

Das Casten von Werten zählt auch zu den Operatoren. Es ist eines der hässlichsten Features von C. Oft ist es aber nicht vermeidbar. Mit einem Cast werden Werte als fremder Typ uminterpretiert.

```
1  double x = 2.1;
2  int *p = (int *) &x;
3  printf ("%f %d %d\n", x, (int) x, *p);
```

Hier werden die Gefahren verdeutlicht. Während `(int) x` die `double`-Variable `x` noch richtig nach `int` konvertiert (und rundet), wird in Zeile 2 nur der Pointer konvertiert, nicht jedoch das, worauf er zeigt. `*p` ergibt also einen korrupten Wert. Die Ausgabe des Programms ist:

```
1 2.100000 2 -858993459
```

Ein sehr selten benutzter Operator ist der Komma-Operator `,`. Er tut nichts, außer zuerst den Ausdruck vor und dann den nach dem Komma zu evaluieren. Dadurch können Anweisungen wie mit `;` hintereinandergereiht werden. Das braucht man nur dann, wenn C eine einzelne Anweisung erwartet aber mehrere ausgeführt werden sollen, was fast nur im Kontroll-Teil einer `for`-Schleife vorkommt. Will man z.B. in einer Schleife zwei Variablen gleichzeitig hochzählen, müsste man das normalerweise so machen:

```
1 j = 10;
2 for (i = 0; i < 15; i ++)
3 {
4   ...
5   j ++;
6 }
```

Das behandelt eine Schleifenvariable ganz anders als die andere und wirkt daher oft eigenartig. Mit dem Komma-Operator kann das so ausschauen:

```
1 for (i = 0, j = 10; i < 15; i ++, j ++)
2 { ... }
```

Ein letzter Operator ist fast schon eine Kontroll-Struktur. Es handelt sich um die Kombination von `?` und `:`. Durch diese Zeichen werden drei Teilausdrücke abgetrennt. Der erste ist eine Bedingung, der zweite wird ausgewertet, wenn die Bedingung wahr ist, der dritte wenn die Bedingung falsch ist. Damit können gewisse `if`-Konstrukte oft stark verkürzt werden. Allerdings gilt dieser Operator als schwer lesbar. Beispiel:

```
1  double x = -1.5;
2  // Variante 1 mit if
3  double y;
4  if   (x < 0)
5       y = -x;
6  else y = x;
7  y *= 2.0;
8  consume (y);
9  // Variante 2 mit ?:
10 consume ((x < 0 ? -x : x) * 2.0);
```

5.3 Kontroll-Strukturen

5.3.1 Selektion

Das normale `if`:

```
1  if (a == 2)
2    puts ("a ist 2);
3  else
4    puts ("a ist nicht 2);
```

Die Bedingung wird in Klammern gesetzt. Der `else`-Teil kann weggelassen werden. Will man mehr als einen Befehl im `if`- oder `else`-Teil unterbringen, muss man `{}`-Klammern setzen:

```
1  if (a == 2)
2  { puts ("a war 2");
3    a ++;
4  }
```

Eine Selektion nach mehreren Fällen kann mit dem `switch`-Konstrukt vorgenommen werden, wenn die Selektion nach einem Integer erfolgt. Floats, Strings oder Strukturen sind also nicht erlaubt.

```
1  int x = irgendwas ();
2  switch (x)
3  {
4     case 1: puts ("1"); break;
5     case 2: puts ("2");
6     case 3: puts ("2,3"); break;
7     case 4:
8     case 5: puts ("4,5"); break;
9     default: puts ("default");
10 }
```

Dieses Konstrukt ist am besten so zu interpretieren: Wird der Wert von `x` hinter einem `case` gefunden, wird dort hin gesprungen und alle Befehle ausgeführt, bis ein `break` auftritt. Danach wird das `switch`-Konstrukt verlassen. Wird der Wert nicht gefunden, wird zu `default` gesprungen. Gibt es das nicht, wird gar nichts gemacht.

Gefährlich ist dabei, auf das `break` am Ende jedes `case` zu vergessen, wie z.B. in Zeile 5. Falls `x=2` ist, wird dann nämlich nicht nur `2` ausgegeben, sondern auch `2,3`. C fährt also einfach fort, die Befehle des nächsten `case` auszuführen. Dieses Verhalten kann man auch ausnutzen, um eine Anweisungsfolge für zwei Fälle zu verwenden, wie in Zeile 7–8. So etwas wie in Zeile 5–6 bewusst zu machen, führt allerdings zu unleserlichen und fehleranfälligen Programmen.

5.3.2 Iteration

Die `while`-Schleife kommt in zwei Ausführungen, einmal mit Abbruchbedingung am Anfang:

```
1  while (x > 1)
2  {
3    l2 ++;
4    x /= 2;
5  }
```

und einmal mit Abbruchbedingung am Ende:

```
1  do
2  {
3    l2 ++;
4    x /= 2;
5  }
6  while (x > 1)
```

Am häufigsten wird allerdings die `for`-Schleife verwendet:

```
1  int i;
2  for (i = 2; i < 100; i ++)
3  {
4    a[i] = a[i-1] + a[i-2];
5    printf ("%d: %d\n", i, a[i]);
6  }
```

Die drei Elemente in der Klammer hinter `for`, die durch `;` getrennt sind, sind einfach Befehle, die zu bestimmten Zeitpunkten ausgeführt werden. Eine allgemeine `for`-Schleife ist auf folgende Weise äquivalent zu einer `while`-Schleife:

```
1  for (S, A, I)
2  {
3    B;
4  }
```

$\approx$

```
1  S;
2  while (A)
3  {
4    B;
5    I;
6  }
```

5.3.3 Sprünge

In C gibt es das böse `goto`. So kann man damit z.B. eine Schleife programmieren:

```
1  int i = 0;
2  loop:
3    printf ("%d\n", i);
4    i ++;
5  if (i < 10) goto loop;
```

Das Sprungziel ist also ein Label, das man einfach mit `:` dahinter festlegt, wie in Zeile 2.

Es gibt aber noch andere Sprungbefehle. Der häufigste ist `break`, den wir schon von `switch` her kennen. Mit diesem Befehl wird eine `while`- oder `for`-Schleife komplett abgebrochen.

```
1  int i;
2  for (i = 1; i < 100; i ++)
3  {
4    a[i] = a[i] - b[i-1];
5    if (a[i] == 0)  break;
6    b[i] = 1.0 / a[i];
7  }
8  printf ("Null bei i=%d\n", i);
```

In diesem Fall wird die Anweisung `b[i] = 1.0 / a[i]` nicht mehr ausgeführt, wenn `a[i]` null ist, und auch keine weiteren Iterationen.

Falls nur der Rest der aktuellen Iteration übersprungen werden soll, also ein Sprung ans Ende des Schleifenkörpers durchgeführt werden soll, kann der Befehl `continue` verwendet werden.

```
1  for (i = 0; i < 100; i ++)
2  {
3    a[i] = a[i] - b[i];
4    if (a[i] == 0) continue;
5    printf ("reziprok %d: %f\n", i, 1.0 / a[i]);
6  }
```

Hier wird der Reziprokwert nur ausgegeben, wenn `a[i]` nicht null ist. Die Schleife wird aber fortgesetzt.

Als letzter Sprungbefehl kann noch der `return`-Befehl gewertet werden, der eine Funktion (vorzeitig) beendet. Zu diesem Befehl siehe Abschnitt 5.4.

Man beachte, dass all diese Sprungbefehle als böse gelten, weil sie angeblich unüberschaubare Programme verursachen. Sie sollten daher nur in Performance-kritischen Codeteilen verwendet werden. Da aber heutige Compiler andere Konstrukte wahrscheinlich ebenso gut hinoptimieren können, ist auch dieser Anwendungsgrund fraglich.

5.4 Funktionen

Funktionen bestehen in C aus einem Return-Typ, einem Namen, einer Parameterliste und dem Funktionskörper in `{}`-Klammern, in dieser Reihenfolge.

```
1 double Power (double x, unsigned a)
2 {
3   double r = 1.0;
4   while (a > 0)
5   { r *= x; a --; }
6   return r;
7 }
```

Hier wird die Funktion `Power` definiert. Sie hat zwei Parameter, `x` vom Typ `double` und `a` vom Typ `unsigned int`. Die Parameter werden "per value" übergeben. Das heißt, dass im Funktionskörper der Wert von `a` verändert werden kann, wie in Zeile 5, ohne dabei den Wert des Parameters außerhalb der Funktion zu verändern. Wird die Funktion also z.B. mit

```
1  unsigned aa = 5;
2  Power (1.5, aa);
```

aufgerufen, dann wird der Wert von `aa` nicht verändert. Hier sieht man auch, dass man direkte Werte beim Funktionsaufruf angeben kann (`1.5`). Ebenfalls sieht man, dass man den zurückgegebenen Funktionswert nicht unbedingt verwerten muss. Bei Aufruf von `Power(1.5,aa)` anstatt z.B. `y=Power(1.5,aa)` wird der Rückgabewert einfach weggeschmissen.

Der Rückgabewert wird innerhalb der Funktion in Zeile 6 mittels `return` produziert. Ist der Rückgabetyp `void`, kann das `return`-Statement entfallen. Ist die Parameterliste leer, d.h. wenn die Funktion keine Parameter hat, dann schreibt man `(void)` oder einfach `()` als Parameterliste.

Funktionen können nur global sein. Das heißt, Funktionen kann man nicht schachteln.

Man kann Funktionen auch deklarieren, ohne sie zu definieren, d.h. ohne den Funktionskörper auszuführen. Das ist wichtig, wenn Funktionen sich gegenseitig aufrufen.

```
1  int fun_b (int n);
2
3  int fun_a (int n)
4  {
5    if (n == 0)
6      return 1;
7    else
8      return fun_b (n - 1) + 1;
9  }
10
11 int fun_b (int n)
12 {
13   if (n == 0)
14     return 0;
15   return n * fun_a (n - 1);
16 }
```

So etwas nennt man wechselseitige Rekursion (mutual recursion). Man wüsste hier nicht, welche Funktion man zuerst definieren soll, weil zur vollständigen Definition der einen die andere schon deklariert sein muss. Es reicht aber eine reine Deklaration oder *forward declaration* wie in Zeile 1. Danach kann `fun_a` ausprogrammiert werden, weil C weiß, was `fun_b` ist. `fun_b` kann/muss dann später ausprogrammiert werden.

Man beachte nebenbei das fehlende `else` zwischen Zeile 14 und 15. Das ist in diesem Fall korrekt, weil im *Ja*-Teil die Funktion mit `return` beendet wird und daher den *Nein*-Teil nicht aufgerufen wird. Ein richtig sauberes Programm würde im *Ja*- und im *Nein*-Teil eine Variable mit dem Return-Wert beschreiben und diesen immer erst in der letzten Zeile der Funktion zurückgeben. Das ist aber auch irgendwo wieder mühsam, nicht wahr?

Es gibt weitere Gründe, die reine Deklaration zu verwenden. Und zwar wenn Funktionen in einem Object-File oder einer Library anderen Programmen zur Verfügung gestellt werden. Dann wird die Deklaration in das Header-File geschrieben. Sowohl das `.c`-File, in dem die Funktion ausprogrammiert ist, als auch das Programm, das die Funktion benutzt, inkludiert das Header-File. Damit ist sichergestellt, dass beide die gleiche Funktions-Schnittstelle (Parameter und Rückgabewert) benutzen. Siehe auch Abschnitt 2.

Achtung: C beschwert sich komischerweise *nicht*, wenn eine Funktion benutzt wird, die nicht deklariert wurde. Es nimmt einfach an, eine solche Funktion existiert. Rückgabewert und Parameter werden default-mäßig als `int` angenommen. Beim Linken fällt eine etwaige Diskrepanz dann nicht mehr auf, weil in Object-Files keine Information über Parameter-Typ und -Anzahl enthalten ist. Das heißt, dass C möglicherweise stillschweigend `double`s als `int`s interpretiert und umgekehrt. Debugging und graue Haare sind oft die Folge.

Reicht ein einziger Rückgabewert nicht aus, muss die Funktion *Out*-Parameter zulassen. Das gibt es in C aber nicht. Alle Parameter werden per *value*, nicht per *reference* übergeben. Pointer schaffen hier die Abhilfe.

```
1  void ComplexUnit (double alpha, double *real, double *imag)
2  {
3    *real = cos (alpha);
4    *imag = sin (alpha);
5  }
```

```
6
7  int main (int argc, char *argv[])
8  {
9    double re, im;
10   ComplexUnit (atof (argv[1]), &re, &im);
11   printf ("%f + %f i\n", re, im);
12 }
```

Hier werden in Zeile 1 die Parameter `real` und `imag` als Pointer auf `double`-Werte übergeben. Die Funktion kann also die Variablen, auf die die Pointer zeigen und die sich außerhalb der Funktion befinden, in Zeile 3–4 verändern, indem die Pointer dereferenziert werden. Beim Aufruf der Funktion in Zeile 9 müssen allerdings die Variablen, in denen das Ergebnis stehen soll, mit dem `&`-Operator in Pointer verwandelt werden.

Zur allgemeinen Verwirrung braucht man das bei Arrays nicht zu machen. Der Grund ist, dass Arrays selbst schon Pointer sind (siehe Abschnitt 5.1.3). Hier ein Beispiel mit Strings, die ja `char`-Arrays sind.

```
1  void SetName (char str[])
2  {
3    strcpy (str, "abc");
4  }
5
6  int main ()
7  {
8    char s[] = "xyz";
9    SetName (s);
10   puts (s);
11 }
```

In Zeile 1 wird einfach ein Array (d.h. ein String) als Parameter definiert. Wie wir wissen, ist dieses Array im Prinzip ein Pointer auf das erste Element des Arrays und könnte daher auch als `char *str` geschrieben werden. (Das würde auch wirklich funktionieren.) Die Elemente des Arrays werden also *nicht* per *value* übergeben sondern sind durch Dereferenzieren des Pointers `str` veränderbar. Daher beschreibt `strcpy` in Zeile 3 wirklich das Array `s`, das in Zeile 8 definiert wurde und in Zeile 9 (ohne `&`) an die Funktion übergeben wurde.

Funktionen sind eigentlich auch nichts als Dinge, die irgendwo im Speicher stehen, quasi konstante globale Variablen vom Typ Funktion. Daher kann man auch Pointer auf Funktionen erzeugen und diese Pointer weiterreichen. So kann ein und derselbe Code verschiedene Aktivität entfalten, indem er eine Funktion, die als Pointer gegeben ist, aufruft. Zeigt der Pointer nämlich auf eine andere Funktion, macht der Code auch etwas anderes. Das ist eine Form der generischen Programmierung. Hier ein Beispiel:

```
1  int search (int x, int *y, void (*iterator) (int **))
2  {
3    int p = 1;
4    while (*y != x)
5    {
6      (*iterator) (&y);
7      p ++;
8    }
9    return p;
```

```
10 }
11
12 void increase (int **a) { (*a) ++; }
13 void decrease (int **a) { (*a) --; }
14
15 int main ()
16 {
17   int y[10] = {11,12,13,14,15,16,17,18,19,20};
18   printf ("von vorn: %d. Element\n", search (13, y, &increase));
19   printf ("von hint: %d. Element\n", search (13, y+9, &decrease));
20 }
```

`search` stellt einen einfachen Such-Algorithmus dar, der ein `int`-Array `y` nach einem Wert `x` durchsucht, und ausgibt, an welcher Position sich dieser Wert befindet. Dazu wird der Array-Pointer `y` solange iteriert (Zeile 6), bis er auf den richtigen Wert zeigt (Zeile 4). Statt des normalen Iterierens, d.h. Pointer erhöhen, wird dem Algorithmus eine Funktion übergeben, die genau das tut, nämlich die Funktion `increase` (Zeile 12). Dieser wird ein Pointer per Reference (also ein Doppelpointer) übergeben und der Pointer wird erhöht. Der Algorithmus wird dann (Zeile 18) aufgerufen unter Angabe des zu suchenden Elements `x`, des Arrays `y` und eines Pointers auf die Iterator-Funktion `increase`.

Wird nun (Zeile 19) nicht die `increase`-, sondern die `decrease`-Funktion übergeben, kann das Array mit dem selben Algorithmus von hinten her durchsucht werden. Dazu muss der übergebene Array-Pointer natürlich anfangs auf das *letzte* Element zeigen, daher `y+9`.

Man beachte, wie ein Funktionen-Pointer deklariert wird (Zeile 1). Es ist `(*iterator)` eine Funktion, weil der Return-Typ (`void`) davor steht und eine Parameterliste (`(int **a)`) dahinter. So werden Funktionen deklariert. Wenn aber `(*iterator)` eine Funktion ist, dann ist `iterator` ein Pointer auf eine Funktion. Ohne die Klammer würde einfach eine Funktion mit Return-Typ `void *` deklariert werden.

5.5 Memory Allocation

Wenn eine Variable definiert wird, wird für diese Variable natürlich Speicherplatz reserviert, damit die Variable auch Werte beinhalten kann. Dieser Speicherplatz existiert allerdings nur so lange, wie die Variable existiert. Ist die Variable *out-of-scope*, d.h. hat das Programm den Bereich verlassen, in dem die Variable deklariert wurde, dann wird der Speicherplatz, den die Variable beansprucht hat wiederverwendet. Allerdings kann noch immer ein Pointer existieren, der auf diesen Speicherbereich zeigt. Wird der Pointer dann schreibend oder lesend benutzt, passiert irgend etwas undefiniertes, meist unangenehmes wie ein Absturz des Programms.

```
1  int *PointerAbuse (int *p)
2  {
3    int x = *p;  // das ist ok
4    return &x;   // das ist böse aber noch nicht fatal
5  }
6
7  int main ()
8  {
9    int a = 123;
10   int *q;
11   q = PointerAbuse (&a);  // bei Funktionsaufruf bleibt a erhalten
```

```
12   *q = 321;               // das ist jetzt fatal: *q existiert nicht mehr
13 }
```

Dieses Beispiel zeigt, dass man zwar den Pointer auf eine Variable problemlos in eine Funktion hinein übergeben kann. Wird aber umgekehrt der Pointer auf eine in einer Funktion deklarierten Variable aus der Funktion herausgebracht wie in Zeile 4, dann erzeugt man damit einen Pointer auf eine nicht mehr existierende Variable. In Zeile 12 wird dann unter Verwendung eines solchen Pointers wild in den Speicher geschrieben. So etwas muss unter allen Umständen vermieden werden. Auch wenn das Programm zufällig dabei nicht abstürzt: der nächste Hacker benutzt den Bug, um seinen Code einzuschleusen.

Bei Arrays (und Strings) muss man auch aufpassen. Wird das Array unter Angabe seiner Länge deklariert, wird Speicherplatz für genau diese Anzahl von Elementen reserviert. Will man später zusätzliche Elemente einfügen, also das Array erweitern, hat man Pech gehabt. Daher muss man das Array von Anfang an groß genug wählen.

Wird das Array ohne Angabe der Länge, also nur als `int *arr;` deklariert, wird gar kein Speicher reserviert. Darum muss man sich selber kümmern. Dazu gibt es die Library-Funktionen `malloc` und `calloc`. Diese reservieren (= *allozieren*) Speicherplatz angegebener Größe und geben einen Pointer darauf zurück. Da – wie gehabt – Arrays eigentlich Pointer sind, kann dieser Pointer der Array-Variablen zugewiesen werden. Allerdings geben die Funktionen `void *`-Pointer zurück, weil sie nicht wissen, welcher Datentyp in die Arrays geschrieben werden soll. Der Pointer muss daher vor der Zuweisung noch entsprechend *gecastet* werden.

```
1   int i;
2   char *str;
3   int *arr;
4   str = (char *) malloc (100);
5   arr = (int *) calloc (100, sizeof (int));
6   strcpy (str, "hello world\n");
7   i = 0;
8   while (str[i] != 0)
9   {
10    arr[i] = str[i];
11    printf ("%d\n", arr[i]);
12    i ++;
13  }
14  free (str);
15  free (arr);
```

Hier werden zwei Arrays mit je hundert Elementen alloziert. `str` ist ein Array von `char`s, also ein String, und wird mit `malloc` auf eine Länge von 100 Bytes alloziert. Streng genommen ist das inkorrekt, weil ein `char` nicht unbedingt ein Byte lang sein muss. Den meisten Menschen kommt jedoch ein Fall, wo das nicht so ist, das ganze Leben lang nicht unter. Das `int`-Array `arr` wird mit `calloc` alloziert. Der erste Parameter ist die Länge des Arrays, der zweite Parameter die Länge eines Elements. Die Länge eines Elements ermittelt man korrekterweise mit der `sizeof`-Funktion. `calloc` multipliziert diese zwei Längen und alloziert dann so viele Bytes. Das ist äquivalent zu `malloc (100*sizeof(int))`. Man beachte die Casts `(char *)` und `(int *)`, die den Typ des Pointers richtig umwandeln.

Danach können die Arrays wie gewohnt bearbeitet werden (6–13). Der Vorteil bei diesem "händischen" Allozieren ist, dass der Speicherplatz nicht verloren geht, wenn man die Funktion verlässt. Der Nachteil dabei ist, dass man sich um die Freigabe des Speichers ebenfalls selbst kümmern muss. Das geschieht mit `free` wie z.B. in Zeile 14–15.

Zwei Arten von Bugs können bei der Verwendung von alloziertem Speicher entstehen. Der erste, kritischere, ist das verfrühte Freigeben von Speicher, der noch benutzt wird. Das kann leicht passieren, weil die Array-Variable auch nach dem `free` noch immer auf den selben (freigegebenen) Speicherplatz zeigt. Manche Programmierer setzen daher nach der Freigabe den Array-Pointer auf `NULL`. Das schützt zwar nicht vor Abstürzen, eher im Gegenteil. Aber es verhindert sogenannte "Heisenbugs", das sind Abstürze o.ä., die nicht immer auftreten und zwar meist z-u-f-ä-l-l-i-g genau dann nicht, wenn man sie "misst", d.h. zu debuggen versucht.

Die zweite Art von Bug heißt Memory-Leak und betrifft das Nicht-Freigeben von belegtem Speicher. Zwar wird beim Beenden eines Programms jeglicher belegter Speicher automatisch freigegeben. Wenn aber ein Programm länger läuft, wie z.B. ein Unix-Daemon monatelang aktiv sein kann, und dieses Programm einen Memory-Leak aufweist, dann kann es passieren, dass das Programm immer größer wird und sich mit im Grunde unbenutzten Daten im Hauptspeicher des Rechners ausbreitet. Es gibt Debugging-Tools wie z.B. *LeakTracer*, die in der Lage sind, die Programmstelle auszugeben an der nicht-freigegebener Speicher alloziert wurde.

Zum Schluss sei noch die Funktion `realloc` erwähnt, mit der man ein Array vergrößern oder verkleinern kann.

```
1  int *arr = calloc (100, sizeof (int));
2  for (i = 0; i < 100; i ++)  arr[i] = i + 50;
3  arr = realloc (arr, 200 * sizeof (int));
4  for (i = 100; i < 200; i ++)  arr[i] = i + 50;
5  for (i = 0; i < 200; i ++)  printf ("%d\n", arr[i]);
```

Dabei bleibt der Inhalt des Arrays erhalten, die Position im Speicher ändert sich aber. Pointer, die auf Elemente des Arrays zeigen, werden dadurch ungültig, daher ist hier höchste Vorsicht geboten.

Der Aufwand, der hier mit Allozieren und Freigeben von Speicher im Vergleich zu Java betrieben werden muss, und auch die Anfälligkeit für Bugs ergibt sich einfach daraus, dass Java einen Garbage-Collector verwendet, der von Zeit zu Zeit unbenutzten Speicher freigibt, und C eben nicht. Garbage-Collectors sind zwar mittlerweile hochoptimiert, machen aber bei zeitkritischen und Echtzeit-Anwendungen immer noch Probleme. Wie auch immer, das Thema explizites Memory-Management versus Garbage-Collection sorgt des öfteren für "Glaubenskriege".

6 Input/Output in C

6.1 Einfaches Text-I/O

Folgende Funktionen zur einfachen Ein-/Ausgabe von Text sind verfügbar:

- `putchar (int c)` gibt ein einzelnes Zeichen aus (`char` obwohl der Parameter `int` ist).
- `puts (char *s)` gibt einen String aus (Null-terminierter C-String).
- `printf (char *format, args...)` gibt Werte beliebigen Typs aus. Das Ausgabeformat kann mit dem Parameter `format` beeinflusst werden.
- `int getchar ()` liest ein einzelnes Zeichen ein.
- `char *gets (char *s)` liest eine ganze Zeile ein und schreibt sie in den String `s`. Ist der String zu kurz dimensioniert, gibt es Probleme. Diese Funktion ist also sehr problematisch.

- `scanf (char *format, args...)` liest Werte beliebigen Typs ein und schreibt sie in die Parameter, die als Pointer übergeben werden müssen.

Natürlich gibt es noch weitere aber das sind vorerst die wichtigsten. Das `format` bei `printf` kann jeden beliebigen String beinhalten, der im Prinzip einfach ausgegeben wird. Auszugebende Werte werden in diesem String mit `%` und einem Kennbuchstaben gekennzeichnet. `%d` steht für ein Integer im Dezimalformat, `%f` für eine Gleitkommazahl, `%s` für einen String. Weitere Möglichkeiten sind der Manual-Page zu entnehmen. Außerdem kann zwischen `%` und dem Kennbuchstaben noch Modifizierer eingebaut werden, die z.B. die Anzahl der auszugebenden Nachkommastellen festlegen.

`scanf` funktioniert genau umgekehrt. Es sind allerdings nur die `%`-Kombinationen im Format-String erlaubt. Sowohl bei `scanf` als auch bei `printf` muss für jedes `%` ein Parameter übergeben werden. Bei `scanf` müssen diese Parameter Pointer sein, damit `scanf` schreibenden Zugriff auf die referenzierten Variablen hat.

Die meisten dieser Funktionen geben einen `int`-Wert zurück. Wenn die Eingabe abbricht, weil die Datei, aus der gelesen wird, zu Ende ist, oder weil der Benutzer Strg-D gedrückt hat, das ASCII-Zeichen für End-of-File, dann nimmt der Rückgabewert den Wert `EOF` an. `EOF` ist ein Makro, das in `stdio.h` definiert wird und üblicherweise den Wert `-1` enthält. Deshalb ist auch der Rückgabewert von `getchar` vom Typ `int` und nicht `char`, `EOF` muss sich von jedem möglichen ASCII-Wert unterscheiden können. `scanf` gibt normalerweise die Anzahl der erfolgreich eingelesenen Werte an. Ist diese kleiner als die erwartete, dann ist ein Fehler aufgetreten. Wahrscheinlich hat der Benutzer Blödsinn eingegeben.

```
1  int i, r;
2  double d;
3  char s[100];
4  do
5  {
6    r = scanf ("%d %lf %s", &i, &d, s);
7    if (r == EOF)
8      puts ("ende");
9    else if (r < 3)
10   {
11     printf ("error: nur %d werte gelesen, restl. zeile gelöscht.\n", r);
12     while ((r = getchar ()) != EOF && r != '\n');
13   }
14   else
15     printf ("%s %8.3f %04d\n", s, d, i);
16 }
17 while (r != EOF);
```

In diesem Beispiel wird so lange versucht, ein `int` ein `double` und einen String einzulesen (Zeile 6), bis End-of-File eintritt (Zeile 17). Die drei Werte werden dann im Normalfall formatiert wieder ausgegeben (Zeile 15). Falls die Anzahl der korrekt eingelesenen Werte kleiner als 3 ist (Zeile 9), dann wird ein Eingabefehler erkannt und ausgegeben (Zeile 11). Außerdem muss dann die fehlerhafte Eingabe übersprungen werden. Dazu wird der Rest der Eingabezeile verworfen, indem so lange ein Zeichen mit `getchar` eingelesen wird, bis das Carriage-Return-Zeichen (`'\n'` = Zeilenende) auftaucht oder die Eingabe zu Ende ist (Zeile 12). Eine Session dieses Programms (`io`) könnte so aussehen:

```
1 $ io
2 1 2.3 hello
```

```
3 hello    2.300 0001
4 1 hello x$*%!
5 error: nur 1 werte gelesen, restl. zeile gelöscht.
6 1   2.345678 world 2 3.4
7 world    2.346 0001
8 rosenknospe
9 rosenknospe    3.400 0002
10 ende
```

Zeile 1 wird korrekt eingelesen und formatiert wieder in Zeile 2 ausgegeben. Zeile 3 enthält einen Eingabefehler, weil `hello` keine Gleitkommazahl ist. Zeile 5 enthält zwar zu viele Eingaben, das macht aber nichts. `scanf` arbeitet nicht zeilengebunden, d.h. die Eingaben `2` und `3.4` werden in der nächsten Iteration eingelesen. Dort ist dann die Eingabe aber noch nicht fertig, daher muss in Zeile 7 noch ein String eingegeben werden. Danach wird Strg-D gedrückt und die Eingabe ist zu Ende. Man beachte auch, dass die Anzahl der Leerzeichen und `\n` ("White-Spaces") zwischen den Eingaben keine Rolle spielt.

Die Tatsache, dass die Ausgabe nicht sofort nach der Eingabe von `world` passiert, ist darauf zurückzuführen, dass die Eingabe zeilengepuffert ist, d.h. jede Eingabe wird erst nach dem Drücken der Returntaste an das Programm übergeben. Um dieses Verhalten zu ändern, müsste man das Terminal-Device mittels `termios`-Funktionen manipulieren, aber das führt hier zu weit.

6.2 POSIX-File-Descriptors

Es gibt nicht nur die Standard-Eingabe und -Ausgabe. Man kann natürlich auch Dateien öffnen und auf diese zugreifen. Es gibt außerdem einen Standard-Error-Ausgabekanal, auf den jedes Programm zugreifen kann. Das hat den Sinn, dass Programme normale Ausgaben von Fehlermeldungen trennen können. Wenn ein Shell-Script die Ausgabe eines Programms umlenkt, z.B. mit `programm >ausgabe.txt`, dann ist es oft sinnvoll, dass eine Fehlermeldung nicht in die Datei `ausgabe.txt` geschrieben wird sondern am Terminal angezeigt wird. Benutzt das Programm den Standard-Error-Ausgabekanal für Fehlermeldungen, dann passiert genau das.

Auf Unix-Systemen sind diese Kanäle als sogenannte *Streams* ausgeführt. Der POSIX-Standard regelt den Zugriff auf diese Streams. Jeder Stream erhält eine für das Programm eindeutige Nummer, den *File-Descriptor*. Die Descriptoren für die Standard-Kanäle sind:

Descriptor	Stream
0	Standard-Eingabe
1	Standard-Ausgabe
2	Standard-Error

Zusätzliche Dateien können mit `open` geöffnet, mit `close` geschlossen und mit `read` und `write` gelesen und beschrieben werden. Mit `lseek` kann die aktuelle Schreib-/Leseposition in einem offenen File verändert werden. Pipes können mit `pipe` erzeugt werden. Und sogar Internet-Verbindungen können als File-Descriptoren behandelt werden (`socket`).

Allerdings gelten diese Funktionen als Low-Level-File-I/O. Außerdem sind sie Betriebssystem-spezifisch und nicht überall zu finden. Um in C plattformunabhängig auf Dateien zuzugreifen, gibt es die C-File-Handles.

6.3 C-File-Handles

Ein File-Handle ist eine Struktur, die in `stdio.h` definiert ist und `FILE` heißt. Die Library-Funktionen, die auf diese Handles zugreifen, fangen fast alle mit `f` an und verlangen einen Pointer auf eine `FILE`-Struktur. Für die Standard-Streams sind Handles vordefiniert, sie heißen `stdin`, `stdout` und `stderr`.

Erzeugt und geschlossen werden Files mit `fopen` und `fclose`. Gelesen und beschrieben mit `fread` und `fwrite`. Es gibt aber auch die Funktionen `fputc`, `fputs`, `fprintf`, `fgetc`, `fgets` und `fscanf`, die das gleiche machen, wie `putchar`, ..., `scanf`, nur dass man einen Stream angeben kann/muss.

```
1  FILE *F;
2  char c, state = 'a';
3
4  F = fopen (argv[1], "r+");
5  if (F == NULL)
6  {
7    fprintf (stderr, "could not open file %s\n", argv[1]);
8    exit (1);
9  }
10
11 c = fgetc (F);
12 while (!feof (F))
13 {
14   if (c == state)
15   {
16     if (state == 'c')
17     {
18       fseek (F, -3, SEEK_CUR);
19       fputs ("xyz", F);
20       state = 'a';
21     }
22     else state ++;
23   }
24   else
25   {
26     if   (c == 'a')
27          state = 'b';
28     else state = 'a';
29   }
30   c = fgetc (F);
31 }
32
33 fclose (F);
```

Dieses Beispiel ersetzt in einer Textdatei alle `abc` durch `xyz`. Als erstes wird in Zeile 1 ein `FILE`-Pointer angelegt. Dieser ist die Verbindung des Programms zur Datei, die in Zeile 4 mit `fopen` geöffnet wird. In Zeile 5–9 wird überprüft, ob das Öffnen des Files erfolgreich war. Wenn nicht, gibt `fopen` nämlich einen `NULL`-Pointer zurück. In diesem Fall wird eine Fehlermeldung ausgegeben, und zwar auf `stderr`, dem Stream, der zur Ausgabe von Fehlermeldungen vorgesehen ist. Außerdem wird das Programm brutal mit `exit (1)` beendet. Die `1` ermöglicht es einem Shell-Script, zu überprüfen, ob das Programm einen Fehler hatte.

Als nächstes wird das Textfile abgearbeitet, indem Zeichen für Zeichen mit `fgetc` eingelesen wird (Zeile 11 und 30), solange bis das Ende des Files erreicht ist (Zeile 12). Die Variable `state` protokolliert mit, wie weit schon ein `abc` erkannt wurde. Ist man z.B. beim `b` angelangt, springt die Variable zum `c` weiter (Zeile 22), was das nächste erwartete Zeichen ist. Kommt nicht das erwartete Zeichen (Zeile 14), fällt `state` wieder auf `a` zurück, außer das aktuelle gelesene Zeichen ist `a`, dann bekommt `state` den Wert `b` (Zeile 26–28).

Wird das `abc` vollständig erkannt (Zeile 16), dann wird an dessen Stelle `xyz` geschrieben. Dazu bewegt man die File-Position mit `fseek` um drei Zeichen zurück. `SEEK_CUR` heißt, dass die Zielposition relativ zur aktuellen Position ist. In Kombination mit `-3` heißt das drei Zeichen rückwärts. `SEEK_SET` stünde für eine absolute Position, d.h. relativ zum Dateianfang, `SEEK_END` wäre relativ zum Dateiende. Danach wird `xyz` mit `fputs` in die Datei geschrieben und `state` auf `a` gesetzt. Man beachte, dass `fputs` im Gegensatz zu `puts` kein Newline (`\n`) ans Ende hängt.

Dass man die Datei lesen *und* schreiben kann, liegt daran, dass die Datei mit dem Modus `"r+"` geöffnet wurde. Normalerweise heißt es entweder `"r"` für Lesen, `"w"` für Schreiben und `"a"` für Append. Hängt man ein `+` an, ist die Datei immer für Lesen und Schreiben geöffnet. Der Unterschied ist dann nur noch, ob der Stream nach dem Öffnen am Anfang oder am Ende der Datei positioniert ist, und ob ein eventuell existierendes File gelöscht (überschrieben) wird.

In Zeile 33 muss das File noch mit `fclose` geschlossen werden.

6.4 Binary I/O

Die Ausgabe von Ganzzahlen und Gleitkommazahlen als Strings in Dateien ist nicht unüblich. `fprintf` bietet alle Unterstützung, die Werte entsprechend einer File-Spezifikation zu formatieren und auszugeben. Für das Einlesen kann man `fscanf` verwenden. Hat man es allerdings mit großen Mengen an Daten zu tun, die möglichst speichereffizient ausgegeben werden sollen, dann ist es besser, die Daten in Binärkodierung auszugeben.

Mit `fread` und `fwrite` können ganze Speicherblöcke beliebiger Länge gelesen bzw. geschrieben werden. Die Kodierung der Daten wird so direkt vom Speicher übernommen und unverändert in die Datei geschrieben. Dabei ist aber Vorsicht geboten. Eigentlich ist das nämlich böse und unerlaubt.

Das Problem liegt darin, dass verschiedene Prozessoren und Plattformen Daten in unterschiedlicher Art kodieren. Schreibt man also eine `int`- oder `float`-Variable auf Computer *A* mit `fwrite` in eine Datei und liest diese auf Computer *B* mit `fread` wieder ein, ist nicht gewährleistet, dass das Ergebnis die selbe Zahl darstellt. Bei Gleitkommazahlen ist es sehr gut möglich, dass ein Computer die IEEE-Kodierung verwendet, der andere nicht.

Bei Ganzzahlen ist das Hauptproblem die *Byteorder*. Ein Integer besteht üblicherweise aus 4 Bytes. Auf manchen Plattformen steht das höchstwertige Byte als "erstes" im Speicher, d.h. an der niedrigsten Speicheradresse. Das hat den Vorteil, dass man im Debugger die Zahl richtig stehen sieht, wenn man die Bytes von links nach rechts aufsteigend ausgibt. Bei Intel ist das der Fall. Man nennt das "little endian". Andere Prozessoren stellen aber meistens das niederwertige Byte an die niedrigere Speicheradresse. Das nennt man "big endian".

Es gibt leider keine einfache Standardlösung für diese Problematik. Das Byteorderproblem könnte man selber lösen, indem man z.B. die Bytes vor dem Schreiben und nach dem Lesen händisch verdreht, falls die Byteorder des Prozessors nicht die gewünschte ist.

```
1 #include<endian.h>
2 ...
3   int x = -35007;
4   char t, *p;
5   if (__BYTE_ORDER == __BIG_ENDIAN)
```

```
6     x =   ((x >> 24) & 0xff) + ((x >> 8) & 0xff00)
7         + ((x & 0xff00) << 8) + ((x & 0xff) << 24);
8   fwrite (F, (char *) &x, 4);
```

Eingefleischten C-Fans dreht es dabei aber den Magen um. Die sind der Meinung, dass bei korrektem Umgang mit C die Byteorder unerheblich ist oder gar sein *muss*. Die würden das so machen:

```
1   int x = -35007, y;
2   FILE *F = fopen ("binary.dat", "w");
3   fputc (x, F);  fputc (x >> 8, F);  fputc (x >> 16, F);  fputc (x >> 24, F);
4   fclose (F);
5   F = fopen ("binary.dat", "r");
6   y = fgetc (F);  y += fgetc (F) << 8;
7   y += fgetc (F) << 16;  y += fgetc (F) << 24;
```

`fputc` speichert immer nur die untersten 8 Bit der Zahl (es verwendet einen Cast auf `unsigned char`). Mit `>>` schiebt man so nach und nach die oberen Bits in diesen Bereich und speichert alle Bits (Zeile 3). Beim Einlesen werden die Bytes in der selben Reihenfolge wieder eingelesen, die Bits an die richtige Stelle gerückt und summiert.

Bei Gleitkommazahlen wird es da allerdings schwieriger. Eine gute Möglichkeit ist die Verwendung der `xdr` Library-Routinen (`xdr` = eXternal Data Representation). Auf diese sei hier nur hingewiesen.

7 System-Calls

Abgesehen von den I/O-Funktionen, gibt es weitere Notwendigkeiten, mit dem Betriebssystem zu interagieren. GUI-Programmierung sei hier außen vorgelassen, weil es den Rahmen sprengen würde. Auf tieferer Ebene liegen die *System-Calls*. Sie stellen die Schnittstelle zwischen einer Anwendung und dem Betriebssystem-Kernel dar. C wurde fast ausschließlich durch Unix-Varianten beeinflusst, daher sind heute viele dieser System-Calls Teil des C-Standards.

Wichtige Gruppen von System-Calls sind: Datums- und Zeit-Funktionen, Prozess- und Threadsteuerung und -Kommunikation, Semaphoren, Zugriffsrechte und Netzwerk-Kommunikation. Threads kommen am Ende noch. Die Datumsfunktionen wollen wir uns hier kurz genauer anschauen, weil sie für Performance-Messungen wichtig sein können.

7.1 Datum und Zeit

Das Datum wird in Unix-Systemen meist in Sekunden seit dem 1. Januar 1970 0 Uhr gemessen. Dieser Wert wird mit `time()` ermittelt. Es gibt Funktionen, um diesen Wert in Jahre, Monate, Tage, usw. umzurechnen.

```
1   time_t Secs;
2   struct tm *Date;
3   Secs = time (NULL);
4   Date = gmtime (&Secs);
5   printf ("Wir schreiben das Jahr %d\n", Date->tm_year + 1900);
6   printf ("Greenwich-Zeit %d Uhr %d\n", Date->tm_hour, Date->tm_min);
7   Date = localtime (&Secs);
```

```
8   printf ("Lokale Zeit %d Uhr %d\n", Date->tm_hour, Date->tm_min);
9   printf ("Standard-Format: %s", asctime (Date));
```

Hier wird in Zeile 3 das aktuelle Datum ermittelt und in Zeile 4 in eine Struktur aufgeschlüsselt. Die Funktion `gmtime` hält sich dabei an die Greenwich-Zeit ohne Sommerzeit. Die `tm`-Struktur enthält mehrere Members wie z.B. `tm_year`, um das Jahr seit 1900 abzulesen. Will man nicht die Greenwich-Zeit sondern die lokale inklusive Sommerzeit, dann muss man wie in Zeile 7 `localtime` benutzen. Mit `asctime` kann man noch einen Datums-String im Standardformat erzeugen.

Ein Nachteil der Funktion `time` ist, dass die Zeit nicht besser als in Sekunden aufgelöst ist. Möchte man genauere Zeitmessungen machen, verwendet man besser `gettimeofday`. Diese Funktion gibt auch Mikrosekunden zurück, auch wenn die Auflösung meist gröber ist. Diese Funktion kann man gut zur Performance-Messung verwenden, indem man die Zeit am Anfang und am Ende misst und die Differenz ausrechnet. Wird das Programm allerdings durch das Betriebssystem oder ein anderes Programm aufgehalten, wird die Messung verfälscht. Dazu gibt es die Funktion `clock` oder noch besser `times`, welche die verbrauchte Prozessorzeit ausgibt, aufgeteilt in User-Zeit und System-Zeit, also der Zeit, die in System-Calls verbracht wird. Hier eine Demonstration:

```
1  #include <stdio.h>
2  #include <time.h>
3  #include <sys/time.h>
4  #include <sys/times.h>
5  #include <unistd.h>
6
7  int main ()
8  {
9    struct timeval T1, T2;
10   struct tms PT1, PT2;
11   int i, ct;
12   gettimeofday (&T1, NULL);
13   times (&PT1);
14   for (i = 0; i < 1000000000; i ++);  // tut irgendwas!
15   gettimeofday (&T2, NULL);
16   times (&PT2);
17   printf ("Echtzeit: %d ms\n", ((int) T2.tv_sec - (int)T1.tv_sec) * 1000
18     + ((int) T2.tv_usec - (int) T1.tv_usec) / 1000);
19   ct = sysconf (_SC_CLK_TCK);
20   printf ("Prozessor Userzeit: %d ms\n",
21     (PT2.tms_utime - PT1.tms_utime) * 1000 / ct);
22   printf ("Prozessor Systemzeit: %d ms\n",
23     (PT2.tms_stime - PT1.tms_stime) * 1000 / ct);
24 }
```

Hier wird in Zeile 14 etwas Zeit sinnlos verbraten. `gettimeofday` gibt die "Wallclock"-Zeit in einer Struktur `timeval` zurück. Diese enthält zwei Members, einen für Sekunden und einen für Mikrosekunden. In Zeile 17–18 wird die Differenz berechnet und die verbrauchte Zeit in Millisekunden ausgegeben. `times` benutzt die Struktur `tms`, welche Werte enthält, die in "Clock-Ticks" gemessen werden. Wieviele Clock-Ticks eine Sekunde hat, kann mit `sysconf(_SC_CLK_TCK)` in Zeile 19 ermittelt werden. Dadurch kann man diese Werte in Millisekunden umrechnen (Zeile 20–23). Member `tms_utime` gibt die Userzeit an, Member `tms_stime` die Systemzeit.

Man beachte übrigens, wieviele Header-Files man für die paar Funktionen inkludieren muss!

7.2 Errors

Viele der System-Calls verwenden eine einheitliche Methode zum Anzeigen von Fehlern. Das gilt auch und ganz besonders für die File-I/O-Funktionen. Auch wenn nicht die POSIX-Variante verwendet wird sondern C-Filehandles, werden intern doch die System-Calls verwendet und die selben Fehler erzeugt.

Tritt ein Fehler auf (z.B. Datei nicht vorhanden), dann gibt die jeweilige Funktion einen Wert aus, der anzeigt, dass die Funktion nicht erfolgreich war. Man erkennt daraus aber nicht den *Grund* für den Fehler. Dieser wird vom Betriebssystem in einer globalen Variable `errno` vom Typ `int` bekannt gegeben.

Zu jedem möglichen Wert von `errno` gibt es in einem globalen Array von Strings eine passende Meldung. Mittels `strerror` kann man diese Meldung ermitteln. Es gibt aber auch eine fixfertige Funktion, die die zu `errno` gehörige Meldung gleich auf `stderr` ausgibt.

Es ist zu beachten, dass jeder neue System-Call `errno` überschreiben kann. Nach einem `printf` könnte `errno` also bereits wieder gelöscht sein. Deshalb sollte man sich, gleich nachdem ein Fehler erkannt wurde, `errno` in einer anderen Variable sichern, falls der Wert noch gebraucht wird.

```
1  #include <stdio.h>
2  #include <errno.h>
3  #include <string.h>
4
5  int main ()
6  {
7    FILE *F = fopen ("/", "w");
8    int myerrno;
9    if (F == NULL)
10   {
11     myerrno = errno;
12     perror ("error");
13     printf ("errno = %d, message = %s\n", myerrno, strerror (myerrno));
14   }
15 }
```

In Zeile 7 wird versucht, ein Verzeichnis als Datei zu öffnen, was naturgemäß nicht funktionieren kann. In Zeile 11 wird `errno` gesichert. `perror` in Zeile 12 verwendet aber noch `errno` selbst. Das geht hier noch, weil es inzwischen noch keinen weiteren System-Call gegeben hat. Danach muss aber `myerrno` benutzt werden, wie in Zeile 13. Der String, den man `perror` übergibt wird einfach der Fehlermeldung vorangestellt und ein Doppelpunkt angehängt. Man könnte hier noch den Programmnamen oder den Namen des Files anführen, das zu öffnen versucht wurde.

Die Ausgabe des Programms ist:

```
1 error: Is a directory
2 errno = 21, message = Is a directory
```

8 C++-Syntax

C++ bringt eine ganze Reihe neuer Konstrukte mit. Schon beim Compilieren ändern sich ein paar Sachen. Das erste ist der Compiler selbst. Hier wird statt `cc` `c++` oder `g++` verwendet. Die Variable im Makefile, die den C++-Compiler enthält, heißt `CXX` (`CPP` ist der Preprozessor). Die Variable, die die Flags enthält, heißt `CXXFLAGS`. Object-Files, die mit `g++` erzeugt wurden, sind nicht so einfach kompatibel mit denen, die mit `cc` erzeugt wurden, daher muss man auch mit `g++` linken. Die Source-Files haben auch eine andere Endung: man kann hier wählen zwischen `.cc`, `.C`, `.cpp`, `.cxx`, `.c++`, `.cp` (mit absteigender Beliebtheit). Die Headerfiles enden aber immer noch mit `.h`.

Hier vorerst ein primitives C++-Programm, um die wichtigsten neuen Elemente vorzustellen.

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main (int argc, char *argv[])
6 {
7   for (unsigned i = 0; i < argc; i ++)
8     cout << i << ": " << argv[i] << endl;
9 }
```

Zeile 1 Statt `stdio.h` wird das `iostream`-Header-File der C-Standard-Library verwendet. Das muss man zwar nicht, aber es ist üblich. Die Header-Files der C++-Standard-Library haben komischerweise alle keine Dateiendung.

Zeile 3 Die Elemente der C-Standard-Library sind in einem Namespace "versteckt" (siehe Abschnitt 8.3). Mit dieser Anweisung werden sie "sichtbar" gemacht.

Zeile 5 Die Hauptprogramm-Schnittstelle ist unverändert.

Zeile 7 Die Variable `i` kann im Kontrollteil der `for`-Schleife deklariert werden. Das geht zwar in modernen C-Versionen auch schon, in C++ kann man Variablen aber (fast) überall deklarieren, zwischen Anweisungen oder gar mitten in Anweisungen. Deklarationen gelten in C++ nämlich selbst als Anweisungen.

Zeile 8 I/O funktioniert mit der C++-Standard-Library wirklich *ganz* anders als in C. `cout` ist ein Objekt, das den Standard-Ausgabekanal repräsentiert. In diesen werden mit dem zweckentfremdeten (überladenen) `<<`-Operator nacheinander Werte verschiedenen Typs "hineingeschoben". Der `<<`-Operator erkennt am Typ des übergebenen Werts, wie er ihn ausgeben muss (Polymorphie). Der Rückgabewert der `<<`-Operation ist wieder `cout` selbst. Deshalb sind solche Ketten von `<<`-Operationen möglich.

`endl` ist ein spezielles Objekt, das nur dazu dient, ein Newline auszugeben. Natürlich wäre auch `<<'\n'` möglich. `cin` ist der Standard-Eingabekanal. Mit `>>` liest man Werte ein.

Das Programm tut natürlich das gleiche wie das in Abschnitt 1.2.

8.1 Boolscher Typ

C++ hat einen neuen primitiven Typ für Boolsche Variablen, und zwar `bool`, der die Werte `true` und `false` annehmen kann.

```
1  bool a = false, b = true;
2  bool c = a || b;
3  cout << c << endl;
```

Dieses Programm gibt `1` aus, was für `true` steht.

8.2 Default-Parameter

Funktionen können Default-Parameter haben, die eingesetzt werden, wenn sie beim Aufruf weggelassen werden.

```
1 int power (int basis, unsigned exponent = 2)
2 {
3   if   (exponent == 0)
4        return 1;
5   else return basis * power (basis, exponent - 1);
6 }
```

Diese Funktion berechnet die Potenz von `basis`. Wenn der Exponent angegeben wird, z.B. mit `power (3, 3)`, wird dieser verwendet und es ergibt sich `27`; wird er nicht angegeben, z.B. `power (3)`, dann wird der Default-Wert 2 verwendet und es ergibt sich `9`.

8.3 Namespaces

Bei all den globalen Strukturen, Funktionen, Typennamen, etc., die in C auftreten, kann es bei größeren Projekten oder bei der Verwendung mehrerer Libraries leicht passieren, dass der gleiche Name an verschiedenen Stellen für verschiedene Dinge auftaucht. Dann gibt es einen Namenskonflikt.

Um das zu vermeiden, hat man in C++ *Namespaces* eingeführt. Damit kann man allen Namen, die innerhalb des Namespaces deklariert werden, den Namen des Namespaces als Präfix verpassen. Außerhalb des Namespaces müssen Namen aus dem Namespace unter Angabe des Präfixes und Verwendung des Namespace-Operators `::` referenziert werden. Man kann allerdings mit der `using namespace`-Anweisung C++ dazu veranlassen, Namen immer auch in einem bestimmten Namespace zu suchen, sodass man den Namespace-Präfix weglassen kann.

```
1  #include <iostream>
2
3  using namespace std;
4
5  namespace ns1
6  {
7  int x;
8  void setx (int px) { x = px; }
9  }
10
11 int main ()
12 {
13   ns1::setx (2);
14   using namespace ns1;
```

```
15   cout << x << endl;
16 }
```

In Zeile 4 wird ein eigener Namespace `ns1` eröffnet. Die Variable `x` und die Funktion `setx` sind dann Teil dieses Namespaces. In `setx` kann auf `x` ohne Angabe von `ns1::` referenziert werden, da sich beide im selben Namespace befinden. Die Funktion `main` befindet sich allerdings außerhalb des Namespaces, daher muss in Zeile 13 `setx` mit `ns1::setx` angesprochen werden. In Zeile 14 wird `ns1` mit `using` "inkludiert", daher kann in Zeile 15 `x` ohne `ns1::` angesprochen werden.

Die `using`-Anweisung in Zeile 3 bewirkt, dass `cout` und `endl` in Zeile 15 ohne `std::` angesprochen werden können, da beide aus der C++-Standard-Library stammen (deklariert im Standard-Header-File `iostream`). Die ganze C++-Standard-Library befindet sich nämlich im Namespace `std`. Die Zeile `using namespace std;` ist also für C++-Programme quasi obligatorisch.

Namespaces können auch geschachtelt werden. Namen werden dann z.B. mit `ns1::ns2::xyz` angesprochen. Wird etwas außerhalb von jeglichem Namespace deklariert, wie z.B. die Funktion `main`, dann ist sie im "Default"-Namespace und kann ohne Präfix angesprochen werden.

8.4 Klassen

Klassen sind in C++ prinzipiell nur Erweiterungen der `struct`-Struktur. Statt `struct` schreibt man `class`, der Rest ist gleich. Allerdings kommen noch einige Möglichkeiten dazu.

Neben Datenmembern kann man auch noch Member-Funktionen deklarieren. Innerhalb einer Member-Funktion kann man die Members der Klasse benutzen, ohne einen expliziten Pointer auf die Instanz der Klasse zu besitzen.

Members können als `public`, `private` oder `protected` deklariert werden, indem man eins dieser Keywords wie ein Label vor die Member-Deklarationen stellt. `private`-Members können nur von Member-Funktionen angesprochen werden, `public`-Members auch von außerhalb der Klasse. `protected` ist wie `private`, nur dass auch Member-Funktionen einer vererbten Klasse diese ansprechen können (dazu später).

```
1  class Konto
2  {
3  public:
4    Konto (char *pName)
5    {
6      Name = pName;
7      Guthaben = 0;
8    }
9
10   ~Konto () {}
11
12   void Einzahlen (int Betrag)  { Guthaben += Betrag; }
13   void Auszahlen (int Betrag)  { Guthaben -= Betrag; }
14
15   void Print ()  { cout << Name << ": " << Guthaben << endl; }
16
17 private:
18   char *Name;
19   int Guthaben;
20 };
```

```
21
22 int main ()
23 {
24   Konto K ("Börsl");
25   K.Print ();
26   K.Einzahlen (100);
27   K.Print ();
28 }
```

Hier wird in Zeile 1–20 eine Klasse Konto definiert. Gleich in Zeile 3 findet sich das `public`-Keyword. Auf das sollte man nicht vergessen, defaultmäßig ist nämlich in einer Klasse alles `private`. Dahinter kommt gleich der *Konstruktor*. Der Konstruktor ist eine Funktion, die den gleichen Namen wie die Klasse hat und die ganz ohne Return-Typ deklariert wird. Sie wird beim Erzeugen eines Objektes aufgerufen. Hier passiert das in Zeile 24. Die Klasse wird in der Variable `K` instanziert (ein Objekt!) und dem Konstruktor der String `"Börsl"` übergeben. Der Konstruktor speichert den String (naja, zumindest einen Pointer darauf) in der privaten Member-Variable `Name` und setzt das Guthaben – ebenfalls eine private Member-Variable – auf 0.

In Zeile 10 befindet sich der Destruktor. Er wird aufgerufen, bevor die Klasseninstanz gelöscht wird. Das passiert, wenn der Scope einer Variable vom Typ der Klasse endet, d.h. wenn z.B. eine Funktion beendet wird, in der die Variable definiert wurde, oder wenn die Klasseninstanz explizit mit `delete` gelöscht wird (dazu später).

Es gilt als gute Programmierpraxis, Member-Daten zu kapseln, d.h. in den `private`-Bereich zu geben und nur über passende Funktionen zugänglich zu machen. Dadurch wird die Implementierung einer Klasse von der Schnittstelle getrennt. In diesem Fall wird auf das `Guthaben` nur über `Einzahlen`, `Auszahlen` und `Print` zugegriffen. Auf die Member-Funktionen wird gleich wie auf Datenmembers mit dem Member-Operator `.` zugegriffen (bei Pointern mit `->`).

Die Ausgabe des Programms ist natürlich `Börsl: 0` und `Börsl: 100`. Man beachte übrigens auch den Strichpunkt am Ende der Klassendeklaration. Alle Deklarationen haben am Ende einen Strichpunkt. Einzig eine vollständig definierte Funktion braucht das nicht. Ohne diesen Strichpunkt kommen oft kryptische Fehlermeldungen.

Man kann die Implementierung von Member-Funktionen auch aus der Klassendeklaration auslagern. So könnte man in Zeile 4 den Konstruktor deklarieren mit

```
4   Konto (char *pName);
```

und die vollständige Definition an anderer Stelle ausführen:

```
1 Konto::Konto (char *pName)
2 {
3   Name = pName;
4   Guthaben = 0;
5 }
```

Hier taucht wieder der `::`-Operator auf, den wir von den Namespaces kennen. Die Members einer Klasse liegen nämlich in deren eigenem Namensraum, der den Namen der Klasse trägt. Bei der Definition der Member-Funktion außerhalb der Klasse muss man daher den vollen Namen der Funktion angeben, und der enthält eben hier den Namen der Klasse als Namespace-Präfix. Innerhalb der Funktion befindet man sich dann aber schon wieder im Namespace der Klasse. Also kann man wie gewohnt die Members ohne Präfix ansprechen.

Dieses Auslagern der Member-Funktionen kann man dazu benutzen, dass man die Klassendeklaration ohne Code in ein Header-File stellt und die Funktionsimplementierungen in ein `.cc`-File. Die Klassendeklaration reicht anderen Source-Files, um die Klasse zu verwenden. Die Funktions-Implementierungen kommen dann in das Object-File, wenn das `.cc`-File compiliert wird.

Es gibt auch die Möglichkeit, Member-Variablen zu initialisieren, und zwar abhängig von den Parametern des Konstruktors. Das ist vor allem dann notwendig, wenn eine Member-Variable als `const` deklariert wurde. Nehmen wir an, der Name eines Kontos darf sich nicht verändern. Dann könnten wir in Zeile 18 schreiben:

```
18  char const * const Name;
```

Dann würde aber die Anweisung in Zeile 6 (`Name = pName`) einen Fehler produzieren. Um dieses Problem zu lösen, gibt es eine eigene Syntax. Man schreibt den Konstruktor auf folgende Weise um:

```
4  Konto (char const *pName)
5    : Name (pName), Guthaben (0)
6  {}
```

Vor dem Funktionskörper kommt ein Doppelpunkt, gefolgt von einer Initialisierungsliste. In dieser werden alle zu initialisierenden Variablen angeführt und mit in Klammern stehenden Werten initialisiert, wie bei der Initialisierung von Objekten.

Der `this`-Pointer ist noch zu erwähnen, das ist ein Pointer, der immer auf das aktuelle Objekt zeigt. Manchmal ist dieser Pointer recht nützlich. Zum Beispiel ist oft gefordert, dass das Ergebnis einer Member-Funktion das (veränderte) Objekt selbst ist, damit mehrere Operationen hintereinandergehängt werden können. Folgende Modifikation der `Einzahlen`-Funktion

```
1  Konto *Einzahlen (int Betrag) { Guthaben += Betrag; return this; }
```

ermöglicht folgende Aufruf-Kette:

```
1  K.Einzahlen ()->Print ();
```

Das macht allerdings viel mehr Sinn, wenn References und überladene Operatoren verwendet werden. Dazu später.

8.5 Vererbung

Eine Klasse von einer anderen abzuleiten geht so:

```
1 class Tresor : public Konto
2 {
3 public:
4   Tresor (char *pName)
5   : Konto (pName)
6   {}
7
8   void Auszahlen (int Betrag) { Guthaben = max (0, Guthaben - Betrag); }
9 };
```

Die Klasse `Tresor` soll ein `Konto` darstellen, in dem das Guthaben nicht unter 0 fallen darf. Ansonsten soll alles gleich sein. In Zeile eins wird durch den Doppelpunkt und die Angabe der Parent-Klasse `Tresor` von `Konto` abgeleitet. `public` gibt an, dass alle Public-Members von `Konto` auch in Tresor public sind, ansonsten wären sie privat. Überladen werden die Funktionen, die sich ändern müssen, das sind der Konstruktor und `Auszahlen`.

Allerdings lässt sich diese Klasse noch nicht korrekt übersetzen. Der Grund ist, dass `Auszahlen` auf die Member-Variable `Guthaben` zugreift, die in `Konto` als `private` deklariert wurde. In `Tresor` ist sie daher nicht sichtbar. Um den Zugriff zu ermöglichen, müssen wir in `Konto` `Guthaben` unter `protected` stellen.

Quizfrage: welche Funktion wird hier aufgerufen?

```
1  Tresor T ("Kistl");
2  Konto *K = &T;
3  K->Auszahlen (100);  // Tresor::Auszahlen oder Konto::Auszahlen
```

In Zeile 2 wird ein `Konto`-Zeiger auf einen `Tresor` erzeugt. Das ist korrekt, weil ein `Tresor` hier ja ein `Konto` ist. Beim Aufruf in Zeile 3 weiß C++ allerdings nicht mehr, dass das Objekt, auf das der Pointer zeigt, eigentlich ein Tresor ist. Daher wird `Konto::Auszahlen` aufgerufen und nicht `Tresor::Auszahlen`. So kann es passieren, dass nun doch ein Tresor einen negativen Inhalt hat. Ein Bug!

Um sicherzustellen, dass auch in solchen Fällen die richtige Funktion aufgerufen wird, muss man die Funktion als `virtual` deklarieren. Und zwar in der Klasse `Konto`:

```
13  virtual void Auszahlen (int Betrag) { Guthaben -= Betrag; }
```

In der Klasse `Tresor` wird der `virtual`-Zusatz vererbt, er muss nicht mehr angegeben werden. Eine Klasse, die eine virtuelle Funktion enthält, ist eine virtuelle Klasse. Die Methode, mit der das Programm entscheidet, welche Funktion nun aufgerufen wird, heißt "late binding". Jede virtuelle Klasse erhält eine Dispatch-Tabelle mit Pointern zu den virtuellen Funktionen, jedes Objekt einen Pointer zu dieser Tabelle. Damit kann zur Laufzeit durch doppeltes Pointer-Auflösen die richtige Funktion gefunden werden. Das ist natürlich etwas langsamer als ein direkter Funktions-Call, moderne Prozessoren unterstützen diese Technik aber schon hervorragend.

Es ist möglich, eine virtuelle Funktion gezielt unimplementiert zu lassen. Dadurch wird es unmöglich, die Klasse zu instanzieren. Das kann aber durchaus erwünscht sein, wenn z.B. eine Überklasse nur als *Interface* fungiert, d.h. nur eine gemeinsame Schnittstelle aller abgeleiteten Klassen darstellt, ohne selbst eine eigenständige Klasse zu sein. So eine Funktion heißt dann *pure virtual*. Sie wird folgendermaßen definiert:

```
1  virtual void Auszahlen (int Betrag) = 0;
```

Eine Klasse mit einer pure-virtual-Funktion nennt man selbst pure-virtual.

In C++ ist auch multiple Vererbung möglich. Dazu gibt man einfach mehrere Eltern-Klassen durch Komma getrennt an. Dadurch ist es möglich, z.B. indirekt eine Klasse zweimal von der selben Klasse abzuleiten. Welche Komplikationen sich dadurch ergeben und wie sie gelöst werden können, wollen wir hier aber nicht besprechen, weil das den Rahmen sprengen würde.

8.6 Const-Member-Funktionen

In C kann man Variablen als `const` deklarieren, in C++ geht das natürlich auch. Was heißt das aber für Objekte, wenn man Member-Funktionen aufruft? Diese könnten ja das Objekt

verändern und man könnte so die `const`-Deklaration umgehen. Die Lösung in C++ ist, dass man nur jene Funktionen aufrufen darf, die das Objekt nicht verändern. Welche Funktionen das sind, weiß C++ aber nicht selbst, man muss ihm das mitteilen, indem man hinter die Parameterliste das Keyword `const` stellt. Solche `const`-Funktionen dürfen keine Member-Variablen verändern und auch keine anderen Nicht-`const`-Funktionen aufrufen.

Die Member-Funktion `Print` sollte also besser so aussehen:

```
1  void Print () const  { cout << Name << ": " << Guthaben << endl; }
```

Nun könnten wir z.B. eine Funktion schreiben, die ein Array von Konten auflistet, wobei die Konten `const`-Objekte sind.

```
1 void PrintKonten (unsigned Anzahl, Konto const *Kontos)
2 {
3   for (unsigned i = 0; i < Anzahl; i ++)
4     Kontos[i].Print ();
5 }
```

Dabei ist sichergestellt, dass `PrintKonten` keine Funktionen aufrufen darf, die ein Konto verändern. Wenn `const`-Funktionen Pointer (oder Referenzen, siehe später) auf Member-Variablen zurückgeben, müssen die Return-Typen auch als `const` deklariert werden. Zum Beispiel so:

```
1  int const *GetGuthabenPointer () const  { return &Guthaben; }
```

8.7 Friends

Andere Klassen und Nicht-Member-Funktionen können auf `private`- und `protected`-Members einer Klasse nicht zugreifen und das ist auch gut so. In seltenen Fällen hat man aber die Notwendigkeit, die Innereien einer Klasse auch einer klassenfremden Funktion zur Verfügung zu stellen. Eine solche Funktion muss man in der Klasse mit dem Keyword `friend` als zugriffsberechtigt deklarieren.

Als Beispiel wollen wir eine Funktion `Bankenaufsicht` implementieren, die überprüft, ob ein "Guthaben" unter `-100` gesunken ist.

```
1 void BankenAufsicht (Konto const *K)
2 {
3   if   (K->Guthaben >= -100)
4        cout << K->Name << " ist ok\n";
5   else cout << K->Name << " ist überzogen\n";
6 }
```

Mit dieser Funktion wollen wir nun ein Konto kontrollieren:

```
1  Konto K ("Bawag");
2  K.Auszahlen (1000000);
3  BankenAufsicht (&K);
```

Wir bekommen natürlich einen Übersetzungsfehler, weil `BankenAufsicht` auf `private`-Members von `Konto` zugreift. Wenn wir aber in die Klasse `Konto` folgende Zeile einbauen:

```
1  friend void BankenAufsicht (Konto const *K);
```

dann funktioniert es.

Man kann auch ganze Klassen als `friend` deklarieren. Alle Member-Funktionen dieser anderen Klasse können dann auf die `private`-Members zugreifen.

Es ist zu beachten, dass `friend`-Deklarationen die Kapselung einer Klasse zerstören. Daher sollten Friends nur in Ausnahmefällen benutzt werden.

8.8 Strings

C-Strings sind etwas unpraktisch, weil sie ihren Speicherplatz nicht selbst verwalten. Sollen z.B. zwei Strings aneinandergereiht werden, muss für den neuen String händisch neuer Speicher alloziert werden. Die C++-Library bietet daher eine eigene (self-contained) Klasse für Strings.

```
1  #include <iostream>
2  #include <string>
3
4  using namespace std;
5
6  int main ()
7  {
8    string a = "hello";
9    string b ("world");
10   a += ' ' + b;
11   cout << a << endl;
12   const char *c = a.c_str ();
13   cout << c << endl;
14 }
```

Dazu muss man zuerst das Header-File `string` inkludieren (Zeile 2). Strings werden deklariert und initialisiert wie in Zeile 8 oder 9. Dort wird ein normaler C-String in einen C++-String verwandelt. In Zeile 10 wird gezeigt, wie man Strings konkateniert bzw. einzelne Zeichen (`' '`) einfügt. Der C++-String erweitert dabei seinen Speicherbereich automatisch. Strings können (wie in Zeile 11) normal ausgegeben werden. C++-Strings können mit der Member-Funktion `c_str` wieder in C-Strings umgewandelt werden (Zeile 12).

8.9 References

Kleine Motivation: Es war eine wichtige Design-Entscheidung von C++, dass Initialisierungen der Art `Typ a = b` möglich sind, wo ein Objekt deklariert wird, das als Kopie eines zweiten Objekts gleicher Klasse konstruiert werden soll. Der Fachausdruck ist: Copy-Constructor. Obige Definition wird übersetzt in `Typ a (b)`, d.h. es würde der Konstruktor `Typ::Typ (Typ b)` aufgerufen. Wenn die Klasse `Typ` nun sehr groß ist, müssen dabei sehr viele Daten an eine Funktion übergeben werden, was langsam ist. Es wäre daher besser, `Typ a (&b)` aufzurufen, d.h. einen Konstruktor `Typ::Typ (Typ *b)` zu verwenden. Das mutet aber erst recht eigenartig an. Stattdessen hat man ein Mittelding aus Pointer und Nichtpointer eingeführt, das es zulässt, eine Funktion `func (b)` aufzurufen, ohne wirklich `b` als Ganzes an die Funktion zu übergeben. Das ist zwar auch eigenartig aber so ist es nun mal. Dieses Mittelding heißt *Reference* und entspricht ungefähr dem, was andere Programmiersprachen darunter verstehen und statt Pointer verwenden.

```
1    int a = 2;
2    int *b = &a;  // Pointer auf a
```

```
3   int &c = a;   // Referenz auf a
4   *b = 3;  cout << c << endl;  // a über b ändern und über c auslesen
5   c = 4;  cout << *b << endl;  // a über c ändern und über b auslesen
6   int d = 5;  c = d;  cout << a << endl;  // Zuweisung ändert Inhalt
```

Hier werden die zwei verschiedenen Arten vorgeführt, eine Variable zu referenzieren. In Zeile 2 wird ein altbekannter Pointer auf die Variable `a` definiert. In Zeile 3 wird eine Referenz auf `a` definiert. Die Referenz wird mit `&` (statt mit `*`) deklariert. Als Initial-Wert wird `a` direkt und ohne Modifikator verwendet. Trotzdem stellt `c` implizit einen Pointer zu `a` dar.

In Zeile 4 wird `a` auf übliche Weise über den Pointer `b` verändert. Danach wird `a` über die Referenz `c` ausgelesen. Die Ausgabe ist daher der neu zugewiesene Wert `3`. `c` kann also ohne Modifikator angesprochen werden und es wird der implizite Pointer dereferenziert. In Zeile 5 passiert das gleiche umgekehrt. Eine Zuweisung zu `c` verändert die Variable `a`, auf die `c` implizit zeigt. Daher hat `*b`, also `a`, den neuen Wert `4`. Zeile 6 zeigt noch, dass eine Zuweisung einer neuen Variable `d` (zu `c`) jene Variable verändert, auf die `c` implizit zeigt, und *nicht* den impliziten Pointer auf die neue Variable umlenkt. `a` wird also auf `5` verändert. Das zeigt, dass eine Referenz, einmal erzeugt, nicht mehr verändert werden kann. Das unterscheidet die C++-References von denen in anderen Sprachen.

Bei der Verwendung von References als Parameter einer Funktion kann man Out-Parameter erzeugen, die man ohne Pointer versorgen kann. So setzt man z.B. mittels folgender Funktion

```
1 void zero (int &x) { x = 0; }
```

durch Aufruf von `zero (a)` die Variable `a` auf 0.

Es ist damit auch möglich, Zuweisungen zu Member-Variablen zu simulieren und die Kapselung dabei halbwegs aufrecht zu erhalten. Dazu folgendes Beispiel:

```
1 class C
2 {
3 public:
4   C (int px) { x = px; }
5   int X () { return x; }
6 private:
7   int x;
8 }
```

Statt der Funktion `X()` zur einfachen Abfrage des Members `x` könnte man folgende Funktion bauen:

```
5   int &X () { return x; }
```

Diese Funktion kann gleich benutzt werden wie bisher. Allerdings ist damit auch folgende Zuweisung möglich:

```
1   C c;
2   c.X() = 42;
```

Dadurch erhält die Member-Variable `x` des Objekts `c` den Wert `42`.

8.10 Memory-Management

Die Funktionen `malloc` und `free` wurden in C++ durch die Operatoren `new` und `delete` ersetzt. Wie `malloc` liefert `new` einen Pointer auf neu allozierten Speicher. Außerdem hat dieser Pointer schon den richtigen Typ, weil man dem Operator den Typ des zu allozierenden Objekts geben muss. Eine Parameterliste in Klammer bedient den Konstruktor. Will man ein Array von Objekten allozieren, kann `[]` hinter den Typnamen gestellt werden.

```
1  int *p1 = new int;                 // Initialisierung nicht garantiert
2  int *p2 = new int();               // Initialisiert mit 0
3  int *p3 = new int(3);              // Initialisiert mit 3
4  Tresor *p4 = new Tresor ("XY");    // Allozierung einer Klasse
5  Tresor *p5 = new Tresor;           // Default-Konstruktor notwendig
6  int *p6 = new int[10];             // Array von 10 Integers
7  Tresor *p7 = new Tresor[10];       // Default-Konstruktor notwendig
8  Tresor *p8 = new Tresor[10]("XY"); // leider nur Gnu-Extension
9
10 p6[5] = *p3;
11
12 delete p1;  delete p2;  delete p3;  delete p4;  delete p5;
13 delete[] p6;  delete[] p7;  delete[] p8;
```

Ohne Angabe einer Parameter-Liste wie in Zeile 1 oder Zeile 5 wird entweder keine Initialisierung durchgeführt (zumindest nicht garantiert), das ist bei primitiven Typen der Fall, oder es wird der Default-Konstruktor aufgerufen, das ist der Konstruktor ohne Parameter. Gibt es diesen nicht, dann gibt es einen Fehler. Bei explizit leerer Parameterliste werden primitive Typen mit `0` initialisiert (Zeile 2). Ansonsten wird mit dem Wert in der Parameterliste initialisiert (Zeile 3), bzw. der zugehörige Konstruktor aufgerufen (Zeile 4).

In Zeile 6 wird ein Array der Länge 10 angelegt. Die Integers sind nicht unbedingt initialisiert. Bei einem Array von Klassenobjekten wird für jedes Arrayelement der Default-Konstruktor aufgerufen (Zeile 7). Eine Initialisierung mit vorgegebener Parameterliste ist leider nur eine Gnu-Extension und nicht Standard-konform.

Zeile 10 zeigt eine mögliche Benutzung der allozierten Objekte. Wenn die Objekte nicht mehr gebraucht werden, müssen sie mit `delete` wieder freigegeben werden (Zeile 12). Dabei wird der Destruktor (die Member-Funktion, die mit `~` anfängt) aufgerufen. Arrays müssen mit dem Operator `delete[]` freigegeben werden, weil sonst nur das erste Elemente freigegeben wird (Zeile 13).

Arrays zu erweitern (ähnlich zu `realloc`), ist mit `new` und `delete` nicht möglich. Da müsste man ein neues Array anlegen, alle Objekte kopieren und das alte Array löschen. Um Arrays dynamisch zu verwalten, verwendet man daher besser die Template-Klasse `vector<>` aus der Standard-Template-Library. Dazu mehr in Abschnitt 10.1.1.

8.11 Operator Overloading

Objekte können mit allen möglichen Operatoren verknüpft werden. Dabei stellt sich die Frage, was z.B. eine Multiplikation zweier Konten überhaupt bedeuten soll. Man hat die Möglichkeit, das festzulegen, indem man diese Operatoren wie Member-Funktionen von Parent-Klassen überlädt. Die Member-Funktion heißt intern `operator` plus dem jeweiligen Operator.

```
1 class Complex
2 {
```

```
3 public:
4   Complex ()  { real = 0.0; imag = 0.0; }
5   Complex (double preal, double pimag)  { real = preal; imag = pimag; }
6   double SqAbs () const  { return real*real + imag*imag; }
7   bool operator< (Complex const &z) const { return SqAbs () < z.SqAbs (); }
8 private:
9   double real;
10  double imag;
11 };
```

Hier wird eine Klasse `Complex` für komplexe Zahlen definiert. Eine komplexe Zahl soll dann kleiner sein als eine andere, wenn ihr Betrag kleiner ist als der der anderen. `SqAbs` in Zeile 6 berechnet den quadrierten Betrag, der für diese Zwecke auch ausreicht. In Zeile 7 wird nun der Operator `<` überladen, indem eine Member-Funktion `operator<` vom Typ `bool` definiert wird. Dadurch kann man nun zwei Objekte vergleichen:

```
1   Complex u (1, 2), v (2, 1.1);
2   cout << (u < v) << endl;
```

Das Ergebnis ist natürlich `1`. Allerdings sind dadurch die Operatoren `>`, `>=`, `<=` und `==` noch nicht definiert. Diese müssen alle extra ausprogrammiert werden.

Der Return-Typ eines Operators kann frei gewählt werden. Man kann z.B. eine Multiplikation definieren, die einen komplexen Wert als Ergebnis liefert:

```
1   Complex operator* (Complex const &z)
2   { return Complex ( real * z.real - imag * z.imag,
3                      real * z.imag + imag * z.real );
4   }
```

Für `operator++` gibt es ein kleines Problem. Es gibt ja zwei solche Operatoren, einen Präfix- (`++z`) und einen Postfix-Operator (`z++`). Um diese zwei zu unterscheiden, hat der Postfix-Operator einen Dummy-Operanden. Es reicht, wenn man irgendeinen Typ angibt, der Name kann entfallen.

```
1   Complex &operator++ ()  { real += 1.0; return *this; }
2   Complex &operator++ (int)  { real += 1.0; return *this; }
```

Zeile 1 überlädt den `++z`-Operator, Zeile 2 den `z++`-Operator.

Zuweisungen von Objekten gleichen Typs werden defaultmäßig so gehandhabt wie bei Strukturen, nämlich durch Kopieren der Member-Variablen. Wenn eine Member-Variable ein Pointer auf einen Speicherbereich ist, den das Objekt selbst verwaltet, dann entsteht eine Situation, wo zwei Objekte die gleichen Speicherbereiche verwenden. Sie können sich dabei gegenseitig die Inhalte überschreiben oder die Speicherbereiche freigeben, während das andere Objekt noch immer versucht, darauf zuzugreifen.

Es ist daher angebracht, den Zuweisungsoperator `=` zu überladen und dabei den Inhalt zu kopieren. Bei der Klasse `Complex` macht das zwar keinen Unterschied, trotzdem zur Demonstration:

```
1   Complex &operator= (Complex const &z)
2   { real = z.real; imag = z.imag;  return *this; }
```

Es werden aber nicht nur bei einer expliziten Verwendung des =-Operators Objekte zugewiesen. Zum Beispiel beim Aufruf einer Funktion, die einen `Complex`-Parameter hat, und zwar ohne Referenz, wird in der Funktion ein neues Objekt erzeugt, dem das übergebene Objekt zugewiesen wird. Im Objekt wird dabei ein Konstruktor aufgerufen, der ein Objekt gleichen Typs als Parameter hat.

```
1  Complex (Complex const &z)  { real = z.real; imag = z.imag; }
```

Dieser Konstruktor heißt Copy-Konstruktor. Er wird auch bei einer Variablen-Initialisierung mit = verwendet. Beispiel:

```
1  Complex u (1.0, 2.0);  // Initialisierung mit normalem Konstruktor
2  Complex v = u;         // Initialisierung mit Copy-Konstruktor
3  Complex w (u);         // bewirkt das gleiche
```

In Zeile 2 wird also *nicht* `operator=` aufgerufen.

Mit ähnlichen Konstruktoren kann man Typ-Konvertierungen implementieren. Eine reelle Zahl kann z.B. durch Zuweisung in eine komplexe Zahl konvertiert werden. Dazu reicht wieder ein einfacher Konstruktor:

```
1  Complex (double x)  { real = x; imag = 0.0; }
```

und man kann folgende Zuweisung tätigen

```
1  Complex z;
2  z = 3.0;
```

wobei hier *nicht* extra ein `operator= (double x)` implementiert werden muss. Falls doch, wird natürlich dieser verwendet.

Die umgekehrte Zuweisung

```
1  double x = z;
```

funktioniert aber nicht so einfach. Dazu müsste man `operator=` von `double` überladen, bzw. dort einen Konstruktor mit `Complex`-Parameter erzeugen. `double` ist aber gar keine Klasse. Man kann aber einen Konvertierungs-Operator zur Verfügung stellen. So ein Operator heißt `operator` plus Typnamen hintereinander geschrieben. Man muss den Typnamen nicht noch einmal als Return-Typ davorstellen.

```
1  operator double () const  { return real; }
```

Die Konvertierung einer komplexen Zahl in eine reelle ergibt nach diesem Operator also den Realteil. Obige Zuweisung `x = z` funktioniert jetzt.

Ganz wilde Sachen kann man machen, wenn man die `new`- und `delete`-Operatoren überlädt. Das wollen wir uns hier aber sparen. Es sein noch erwähnt, dass es auch möglich ist, freie Operatoren zu überladen, d.h. nicht innerhalb einer Klasse sondern so wie eine globale Funktion. Dazu gibt es in Abschnitt 9 noch Beispiele, wenn das Prinzip `cout << x` auf Objekte selbst definierter Klassen ausgedehnt wird.

8.12 Casts und Typ-Identifikation

Ein *Cast* ist in C die Aufforderung an den Compiler, den Typ einer Variable umzuwandeln. In C++ gibt es diesen Cast zwar noch, man soll ihn aber nicht verwenden. Ein ähnliches Konstrukt hat in C++ die Form `type (expression)`, also eine Typkonvertierung die die Form eines Funktionsaufrufs hat. Das ist eigentlich gar kein Cast sondern die Aufforderung an den Compiler, ein Objekt des angegebenen Typs `type` aus dem angegebenen Ausdruck `expression` zu erzeugen.

Es gibt aber auch tatsächliche Casts, die in C++ viel genauer unterschieden werden. Das erste ist der normale `static_cast`.

```
1  int a = 4, b = 3;
2  double d;
3  d = a / b;  cout << d << endl;
4  d = (double) a / b;  cout << d << endl;
5  d = static_cast<double> (a) / b;  cout << d << endl;
```

Hier sieht man, was passiert, wenn man zwei Integers dividiert: der Nachkomma-Teil wird abgeschnitten, obwohl das Ergebnis einer Gleitkommavariablen zugewiesen wird. Zeile 3 produziert als Ausgabe also `1`. Zeile 4 enthält den altmodischen Cast, der das richtige Ergebnis `1.33333` liefert. Sobald nämlich `a` in `double` umgewandelt wurde, wird die Division mit `double`-Werten durchgeführt, daher wird `b` automatisch ebenfalls in `double` verwandelt. Zeile 5 enthält die modernere Variante, die das gleiche tut.

Ein zweiter Cast-Operator betrifft den `const`-Modifikator. Mit `const_cast` kann man das `const` entfernen. Klarerweise ist das ein dubioser Vorgang. Es sei z.B. folgende Funktion definiert:

```
1 void printstring (char *str)  { cout << str << endl; }
```

Diese Funktion gibt einen String aus und verändert diesen nicht, trotzdem fehlt das `const` beim Parameter `str`. Wenn man nun einen `const`-String gegeben hat und mit dieser Funktion ausgeben will, gibt es einen Compilation-Error. Mit dem `const_cast` kann man sich behelfen:

```
1  char const *hallo = "hallo";
2  printstring (const_cast<char *>(hallo));  // so funktionierts
```

Der nächste Cast ist der ärgste. Damit kann man den Typ von Pointern uminterpretieren. Folgender Code nutzt den Cast um die Bytes einer `double`-Variable auszugeben.

```
1  double x = 1.5;
2  for (unsigned i = 0; i < sizeof (double); i ++)
3    cout << int (*(reinterpret_cast<unsigned char *>(&x) + i)) << endl;
```

Um Pointer auf Klassen in Pointer auf abgeleitete Klassen umzuwandeln, gibt es einen eigenen Cast. Dieser prüft zur Laufzeit, ob der Cast ok ist, d.h. ob das Objekt, auf das der Pointer zeigt, tatsächlich eine Instanz der abgeleiteten Klasse ist. Das funktioniert nur bei virtuellen Klassen. Betrachten wir also noch einmal die Klassen `Konto` und `Tresor`. Nehmen wir an, die Funktion `Auszahlen` wäre nicht virtuell. Damit die Klassen selbst aber virtuell sind, machen wir irgend eine andere Funktion virtuell, z.B. den Destruktor in `Konto` (`virtual ~Konto ()`). Folgende Funktion könnte benutzt werden, um auf umständliche Weise selbst dafür zu sorgen, dass die richtige `Auszahlen`-Funktion benutzt wird:

```
1 void KontoAuszahlen (Konto *K, int Betrag)
2 {
3   Tresor *T = dynamic_cast<Tresor *> (K);
4   if (T)
5     T->Auszahlen (Betrag);
6   else
7     K->Auszahlen (Betrag);
8 }
```

Der `dynamic_cast` in Zeile 3 liefert einen Pointer vom gewünschten abgeleiteten Typ `Tresor *`. Allerdings nur, wenn `*K` wirklich ein `Tresor` ist. Ansonsten gibt er einen Pointer mit Adresse `0` aus. Die `if`-Bedingung in Zeile 4 überprüft das und ruft je nach Ergebnis die richtige `Auszahlen`-Funktion durch Benutzung des neuen `T`-Pointers bzw. des alten `K`-Pointers auf.

Dieser Vorgang funktioniert auch mit References. Hier kann jedoch kein `0`-Pointer ausgegeben werden. Stattdessen wird eine Exception ausgelöst. Dazu kommen wir aber erst in Abschnitt 8.16.

Um die Klasse zu identifizieren, die ein Objekt hat, gibt es noch den `typeid`-Operator. Um diesen zu benutzen, muss man die Header-Datei `typeinfo` inkludieren. `typeid` liefert ein Objekt, das den Namen der Klasse kennt und das man außerdem mit dem `typeid`-Objekt eines anderen Objekts oder einer anderen Klasse vergleichen kann. Folgende Anweisung gibt den Namen der Klasse des Objekts aus, auf das `K` zeigt:

```
1   cout << typeid(*K).name () << endl;
```

`name ()` ist vom Typ `char *`. Mit der Möglichkeit, `typeid`s zu vergleichen, kann man die `if`-Bedingung in obiger Funktion umprogrammieren:

```
4   if (typeid (*K) == typeid (Tresor))
```

`typeid` funktioniert also auch mit einem Klassennamen als Operand. Auch hier müssen die Klassen aber virtuell sein, d.h. mindestens eine virtuelle Funktion enthalten.

8.13 Inlining

Eine wichtige Möglichkeit zur Programm-Optimierung durch Compiler ist *Inlining*. Dabei wird beim Aufruf einer Funktion statt des Funktions-Calls der ganze Code einer Funktion dort eingefügt, wo die Funktion aufgerufen wird. Dadurch wird der übersetzte Programmcode länger, da der Code der einen Funktion an mehreren Stellen auftauchen kann, aber man spart sich einen Funktionscall mit all dem Register-, PC- und Parameter-Speichern am Stack. Vor allem bei kurzen Funktionen ist das sinnvoll.

C++ bietet die Möglichkeit, dem Compiler Funktionen zum Inlining zur Verfügung zu stellen, und zwar mit dem Keyword `inline`, das der Funktionsdefinition vorangestellt wird. Inline-Funktionen werden nicht in das Object-File geschrieben und werden daher auch nicht gelinkt. Sie sollten daher in einem Header-File stehen, und zwar nicht nur die Deklaration sondern auch die Implementierung.

```
1 inline int max (int a, int b)  { return a > b ? a : b; }
```

Auch Member-Funktionen können `inline` sein. Werden Funktionen innerhalb einer Klassendeklaration definiert, sind sie automatisch `inline`, werden sie nur deklariert, nicht. Soll die Implementierung außerhalb der Klassendeklaration `inline` sein, so muss man das Keyword anführen.

```
1  class Auto
2  {
3  public:
4    Auto (double pPS) { PS = pPS; }
5    void SetPS (double pPS);
6  private:
7    double PS;
8  };
9
10 inline Auto::SetPS (double pPS) { PS = pPS; }
```

Diese Klasse `Auto` hat zwei Member-Funktionen und beide sind `inline`. Der Konstruktor, weil die Definition innerhalb der Klassendeklaration ist, die Funktion `SetPS`, weil sie zwar außerhalb, aber mit `inline` definiert wird.

Man muss allerdings dazusagen, dass `inline` nur eine "Aufforderung" an den Compiler ist, die Funktion inline zu machen. Der Compiler kann, muss sich aber nicht daran halten. Auf der anderen Seite kann der Compiler normale Funktionen inline machen, sofern die Funktion nicht aus einem Object-File oder einer Library dazugelinkt wird.

8.14 Static Members

So wie eine Funktion statische Variablen haben kann (siehe Abschnitt 5.1.9), kann auch eine Klasse statische Member-Variablen haben. Diese Variablen existieren nur einmal im Programm, jedes Objekt der Klasse greift aber auf dieselbe Variable zu. Das kann man zu allem möglichen gebrauchen, z.B. könnte man mitzählen, wieviele Objekte einer Klasse gleichzeitig existieren, indem jeder Konstruktor eine statische Variable erhöht und der Destruktor sie vermindert. Man könnte sogar eine Liste aller Objekte aufbauen.

Es gibt aber auch statische Member-Funktionen. Diese sind nicht mit einem Objekt assoziiert, haben also keinen `this`-Pointer und können nur auf statische Member-Variablen zugreifen. Sowohl statische Member-Variablen als auch -Funktionen können `public` oder `private` sein. Variablen sollten natürlich `private` sein.

Folgendes Beispiel implementiert die Objektzähl-Idee von oben:

```
1  class Klasse
2  {
3  public:
4    Klasse () { anz ++; }
5    ~Klasse () { anz --; }
6    static int Anzahl () { return anz; }
7  private:
8    static int anz;
9  };
10
11 int Klasse::anz = 0;
12
13 int main ()
```

```
14 {
15   Klasse a, b, c;
16   cout << Klasse::Anzahl () << endl;
17 }
```

Die statische Variable `anz` zählt also die Anzahl der Objekte, indem der Konstruktor sie erhöht und der Destruktor sie vermindert. Statische Variablen müssen außerhalb der Klassendeklaration noch definiert werden, weil sie sonst keine linkbare Adresse hätten. Dort können sie auch initialisiert werden (Zeile 11). Konstante statische Variablen (`const`) können auch in der Klassendeklaration initialisiert werden, sie haben dann aber keine Adresse und sind im Prinzip nur vordefinierte Werte, ähnlich einem Preprozessor-Makro.

Die statische Member-Funktion `Anzahl` (Zeile 6) gibt die Objekt-Anzahl aus. In Zeile 16 wird sie benutzt. Und zwar ohne Angabe eines Objekts, nur mit dem Klassen-Namespace-Präfix. Die Ausgabe ist `3`.

8.15 Templates

Des öfteren steht man vor dem Problem, den gleichen Code für verschiedene Typen oder Klassen schreiben zu müssen. Das ist erstens öd, zweitens muss dann jede Änderung mehrfach durchgeführt werden. Man kann versuchen, diese Unannehmlichkeit mit Mitteln wie künstlichen gemeinsamen Überklassen oder gar Preprozessor-Makros zu begegnen. Glücklicherweise aber bietet C++ die Möglichkeit der generischen Programmierung.

Die häufigste Variante ist, eine ganze Funktion mit einem Typ zu parametrisieren. Dazu setzt man vor die Funktionsdeklaration das `template<typename xyz>`, wobei für `xyz` der parametrisierende Typ einzusetzen ist.

```
1  template <typename T>
2  void sort (T *arr, unsigned len)
3  {
4    for (unsigned i = 0; i < len - 1; i ++)
5      for (unsigned j = i + 1; j < len; j ++)
6        if (arr[i] > arr[j])
7        { T tmp = arr[i];  arr[i] = arr[j];  arr[j] = tmp; }
8  };
9
10 int main ()
11 {
12   char txt[] = "hello world";
13   sort (txt, strlen (txt));
14   cout << txt << endl;
15   int ints[] = {5, 1, 7, 6, 2, 9, 3, 4, 8};
16   sort (ints, 9);
17   for (unsigned i = 0; i < 9; i ++)
18     cout << ints[i] << ' ';
19   cout << endl;
20 }
```

In Zeile 1 wird `T` als `template`-Typnamen-Parameter deklariert. Die Funktion `sort` kann `T` wie einen normalen Typ benutzen. Es muss nur gewährleistet sein, dass der `>`-Operator bei dem Typ funktioniert. Die `sort`-Funktion wird in Zeile 13 aufgerufen. Man müsste eigentlich das Template mittels `sort<char>` auf den gewünschten Typ konkretisieren, damit der Parameter

`arr` den Typ `char *` annimmt, der ja übergeben wird (`txt`). Aber der Parameter selbst bestimmt den Template-Typ schon genau, daher findet C++ den passenden Typ selbst. Die Ausgabe in Zeile 14 ergibt dann den sortierten String `" dehllloorw"`. Zeile 16 zeigt, dass `sort` ganz einfach auch auf ein Integer-Array angewendet werden kann.

Oft werden für solche generischen Algorithmen Funktionsobjekte übergeben. Das sind Objekte, die es erlauben, zwei Objekte des Template-Typs miteinander zu vergleichen. Solche Objekte beinhalten üblicherweise nichts als den überladenen `()`-Operator, der zwei Objekte als Parameter nimmt. Betrachten wir ein Beispiel:

```
1  #include <iostream>
2  #include <functional>
3
4  using namespace std;
5
6  template <typename T, typename compt>
7  void sort (T *arr, unsigned len, compt comp)
8  {
9    for (unsigned i = 0; i < len - 1; i ++)
10     for (unsigned j = i + 1; j < len; j ++)
11       if (comp (arr[i], arr[j]))
12       { T tmp = arr[i];  arr[i] = arr[j];  arr[j] = tmp; }
13 };
14
15 struct
16 { bool operator() (char a, char b)
17   { if (a == 'l') return false;
18     if (b == 'l') return true;
19     return a > b;
20   }
21 } mycomp;
22
23 int main ()
24 {
25   char txt[] = "hello world";
26   sort (txt, strlen (txt), mycomp);    cout << txt << endl;
27   greater<char> chargrt;
28   sort (txt, strlen (txt), chargrt);   cout << txt << endl;
29   less<char> charless;
30   sort (txt, strlen (txt), charless);  cout << txt << endl;
31 }
```

In Zeile 15–21 wird so ein Funktionsobjekt erzeugt. Es enthält den überladenen `operator()` mit zwei `char`s als Parameter. Diese Funktion hat das Ziel, das `l` zum kleinsten Buchstaben zu machen. `mycomp('l', b)` muss daher immer `false` ergeben (Zeile 17), `mycomp(a,'l')` immer `true`. Ansonsten wird die alphabetische Ordnung beibehalten (Zeile 19). Der generische Algorithmus wird nun so modifiziert, dass das Funktions-Objekt `mycomp` zum Vergleich der Elemente in `arr` verwendet wird (Zeile 11). Dazu muss das Funktions-Objekt an den Algorithmus übergeben werden. Es erhält als Parameter den Namen `comp` (Zeile 7) und muss einen eigenen Template-Typen `compt` bekommen (Zeile 6). Hier sieht man auch, wie man eine Funktion mit mehreren Template-Typen gleichzeitig parametrisieren kann. In Zeile 26 wird der Algorithmus aufgerufen und das Funktions-Objekt übergeben.

Die C++-Standard-Template-Library bietet im Header-File `functional` (Zeile 2) für genau diese Zwecke Template-parametrisierbare Funktions-Klassen. So ergibt `greater<char>` eine Klasse, mit der ein Funktions-Objekt instanziert werden kann, das einfach den `>`-Operator wiedergibt (Zeile 27). Wird dieses Objekt statt `mycomp` verwendet, wird ganz normal sortiert (Zeile 28). Wird stattdessen `less<char>` genommen, wird umgekehrt sortiert (Zeile 29–30). Die Ausgabe des Programms ist also:

```
1 lll dehoorw
2  dehllloorw
3 wroolllhed
```

Wie bereits erwähnt, gibt es nicht nur Template-Funktionen, sondern auch ganze Template-Klassen. Dazu stellt man `template<typename xyz>` vor die Klassendeklaration. Folgende Klasse implementiert einen Punkt in einem n-dimensionalen Raum, dessen Koordinaten beliebigen Typ haben können. Dieser Typ und n sind die Template-Parameter.

```
1  template <typename Koordinate, unsigned Dimension = 2>
2  class Punkt
3  {
4  public:
5    Punkt ()
6    { for (unsigned i = 0; i < Dimension; i ++) Koordinaten[i] = 0; }
7
8    Punkt (Punkt const &K)
9    { for (unsigned i = 0; i < Dimension; i ++)
10       Koordinaten[i] = K.Koordinaten[i];
11   }
12
13   Koordinate operator[] (unsigned d) const { return Koordinaten[d]; }
14   Koordinate &operator[] (unsigned d)  { return Koordinaten[d]; }
15
16   void Print ();
17
18 private:
19   Koordinate Koordinaten[Dimension];
20 };
21
22 template<typename Koordinate, unsigned Dimension>
23 void Punkt<Koordinate, Dimension>::Print ()
24 {
25   cout << '(';
26   for (unsigned i = 0; i < Dimension; i ++)
27   {
28     if (i > 0)  cout << ',';
29     cout << Koordinaten[i];
30   }
31   cout << ')';
32 }
```

Die Klasse `Punkt` hat also zwei Template-Parameter, einen Typnamen `Koordinate`, der Typ der Koordinaten, und ein `unsigned int Dimension`, die Dimension des Raums. Template-Parameter

sind also nicht auf Typnamen beschränkt. Außerdem hat der Parameter `Dimension` einen Default-Wert. Wird der zweite Parameter bei der Template-Instanzierung weggelassen, dann wird Dimension 2 angenommen.

Ein Punkt hat n Koordinaten, wie in Zeile 19 spezifiziert. Man beachte, dass diese Definition nur möglich ist, weil der Template-Parameter für die Klasse eine Konstante darstellt. Für jeden anderen Parameter-Wert wird quasi eine eigene Klasse angelegt.

Die Klasse enthält die üblichen Funktionen: einen Konstruktor, einen Copy-Konstruktor und einen `const`- und einen Non-`const`-Operator zum lesenden und schreibenden Zugriff auf die Koordinaten. Die Funktion `Print` wird außerhalb der Klassendeklaration definiert (Zeile 22–32). Dazu müssen alle Template-Parameter wieder angegeben werden und der Klassenname als Präfix vor dem Funktionsnamen entsprechen parametrisiert werden.

Die Klasse kann nun auf folgende Weise benutzt werden:

```
1  Punkt<double,2> K;
2  K[0] = 1.0;  K[1] = 2.2;
3  K.Print ();  cout << endl;
```

Nun könnte es sinnvoll sein, einen `double`-Punkt (d.h. einen mit Koordinaten vom Typ `double`) in einen `int`-Punkt zu kopieren und die Koordinaten entsprechend konvertieren zu lassen. Dazu müssen wir aber erst einen passenden Copy-Konstruktor definieren. Um jeden möglichen Fremd-Typ konvertieren zu können (sofern das möglich ist), führen wird diesen Konstruktor als "Template im Template" aus:

```
1  template <typename FremdKoordinate>
2  Punkt (Punkt<FremdKoordinate, Dimension> const &K)
3  { for (unsigned i = 0; i < Dimension; i ++)
4      Koordinaten[i] = Koordinate (K[i]);
5  }
```

Der Konstruktor hat also insgesamt drei Template-Parameter: `Koordinate`, `Dimension` und `FremdKoordinate`. In Zeile 4 werden die Fremd-Koordinaten in den eigenen Koordinaten-Typ umgewandelt und eingetragen. Nun ist folgende Anweisung möglich:

```
1  Punkt<int> L = K;
```

Dabei wird die zweite Koordinate von `K`, nämlich `2.2` auf `2` gerundet.

Templates kann man auch *spezialisieren*. Man könnte z.B. für Dimension 1 die Klammer in der Ausgabe des Punkts mit `Print` weglassen. Das ginge so:

```
1 template<>
2 void Punkt<double, 1>::Print ()
3 { cout << Koordinaten[0]; }
```

Jetzt würde die Anweisungsfolge

```
1  Punkt<double,1> M; M[0] = 1.7;
2  M.Print ();  cout << endl;
```

nur noch `1.7` statt `(1.7)` ausgeben. Diese Spezialisierung mit `template<>` ohne Parameter ist eine *vollständige* Spezialisierung, d.h. für alle Template-Parameter werden konkrete Werte bzw. Typen eingesetzt. Nützlich wäre es hier, die Funktion `Print` nur *partiell* zu spezialisieren, d.h. den Koordinaten-Typ als Template-Parameter beizubehalten. Dann wäre das Weglassen der Klammer auch automatisch bei allen anderen Typen erledigt. Also so:

```
1 template<typename Koordinate>
2 void Punkt<Koordinate, 1>::Print ()   // inkorrekt!
3 { cout << Koordinaten[0]; }
```

Leider geht das nicht. Aus irgendeinem Grund darf man in C++ nur ganze Template-Klassen partiell spezialisieren. Das ginge hier natürlich. Allerdings müsste man dann für Dimension 1 die ganze Klasse inklusive Konstruktoren und Member-Variablen neu implementieren.

8.16 Exceptions

Die Behandlung von Fehlern in Programmen ist immer recht kompliziert, wenn man alle Eventualitäten korrekt behandeln will. Das Problem ist, dass man einen aufgetretenen Fehler oft an ganz anderer Stelle behandeln muss als dort, wo er aufgetreten ist. Man könnte z.B. Gotos verwenden, um dieses Problem zu entschärfen, Gotos sind aber – wie gesagt – böse und können nicht über Funktionen hinweg angewendet werden, zum Glück auch in C und C++ nicht. In C++ gibt es daher *Exceptions*.

Sehen wir uns zuerst einmal an, was passiert, wenn C++ auf einen Fehler stößt.

```
1 int main ()
2 {
3   for (;;) new int[1000000];
4 }
```

Hier wird endlos Speicher alloziert, bis keiner mehr da ist. Dann löst der `new`-Operator eine Exception aus, die nicht abgefangen wird. Das Programm wird beendet und gibt

```
1 Aborted
```

aus. Nun kann man diese Exception aber auch abfangen und darauf reagieren. Als erstes machen wir es uns ganz einfach: wir erzeugen unsere eigene Exception und sehen, wie wir damit umgehen können.

```
1  int main ()
2  {
3    try
4    {
5      throw "hello error";
6      cout << "hello world\n"
7    }
8    catch (char const *error)
9    { cout << error << endl; }
10   cout << "bye-bye world\n";
11 }
```

Hier wird in Zeile 5 eine eigene Exception "geworfen". Diese hat den Typ `char *`, also ein normaler C-String. Anweisungen, die potentiell Exceptions erzeugen, kann man nun in einen `try`-Block klammern. Ein solcher Block beginnt mit dem Keyword `try` (Zeile 3), gefolgt von einem Block aus Anweisungen (Zeile 4–7) und einem oder mehreren `catch`-Blöcken. Ein `catch`-Block beginnt mit dem Keyword `catch`, gefolgt von einem Parameter in Klammer (Zeile 8) und einem weiteren Anweisungsblock (Zeile 9).

Wenn ein Catch-Block verfügbar ist, der genau den Typ der geworfenen Exception akzeptiert, dann wird dessen Anweisungsblock ausgeführt und das Exception-Objekt übergeben. Hier passt der Catch-Block zur Exception (`char *`) und die zugehörigen Anweisungen werden ausgeführt, d.h. der geworfene String wird ausgegeben. Danach fährt das Programm nach dem `try`-Block fort, sofern der Catch-Block das nicht verhindert. Die Ausgabe ist also:

```
1 hello error
2 bye-bye world
```

Sehen wir uns nun an, wie das mit der Allozierungs-Exception funktioniert.

```
1  int main ()
2  {
3    int *ptr[10000];
4    unsigned Anzahl;
5    try
6    {
7      for (Anzahl = 0; Anzahl < 10000; Anzahl ++)
8        ptr[Anzahl] = new int[1000000];
9    }
10   catch (exception &error)
11   {
12     cout << error.what () << endl;
13     for (unsigned i = 0; i < Anzahl; i ++)  delete[] ptr[i];
14   }
15 }
```

Hier werden die allozierten Speicherbereiche in einem Array `ptr` von Pointern verwaltet, um sie später wieder freigeben zu können. Die gefährlichen Statements werden in einen `try`-Block geklammert. Die Exception, die der `new`-Operator aufwirft, ist vom Typ `std::exception`. Der Catch-Block in Zeile 10–14 akzeptiert ein Objekt diesen Typs (natürlich wurde `using namespace std` gesetzt) und wird daher aufgerufen. Die Klasse `exception` enthält eine Funktion `what`, die einen C-String mit dem Namen der Exception enthält. Dieser wird hier zuerst ausgegeben (Zeile 12). Danach werden alle bisher allozierten Speicherbereiche wieder freigegeben (Zeile 13). Man beachte, dass `ptr` und `Anzahl` außerhalb des `try`-Blocks deklariert werden müssen, weil sie sonst im `catch`-Block out-of-scope sind und nicht mehr benutzt werden können.

Eine beliebte Methode ist, sich eigene Error-Klassen zu definieren, die so gestaltet sind, dass sie soweit wie möglich selbst wissen, was zu tun ist. Zum Beispiel so:

```
1  class Ausnahme
2  {
3  public:
4    Ausnahme (string const &pMeldung) : Meldung (pMeldung) {};
5    virtual void Handle () const = 0;
6    void PrintMeldung () const { cerr << Meldung << endl; }
7  private:
8    string const Meldung;
9  };
10
11 class Error : public Ausnahme
12 {
```

```
13 public:
14   Error (string const &pMeldung) : Ausnahme ("error: " + pMeldung) {}
15   void Handle () const  { PrintMeldung (); exit (1); }
16 };
17
18 int main ()
19 {
20   try
21   {
22     throw Error ("künstlicher Fehler");
23   }
24   catch (const Ausnahme &error)  { error.Handle (); }
25 }
```

Hier gibt es eine virtuelle Überklasse `Ausnahme`, die mit einer Fehlermeldung initialisiert wird. Die Klasse stellt die virtuelle Funktion `Handle` zur Verfügung, die tun soll, was im Fehlerfall zu tun ist. Davon kann man jetzt beliebige Klassen ableiten, wie z.B. die Klasse `Error` in Zeile 11-16. Diese implementiert die `Handle`-Funktion so, dass zuerst die Fehlermeldung auf `cerr` ausgegeben wird (unter Zuhilfenahme von `PrintMeldung`) und beendet das Programm. Im Konstruktor wird außerdem noch der String `"error: "` vor die Fehlermeldung gehängt.

Das Programm selbst wirft nun in Zeile 22 ein Objekt der Klasse `Error` als Exception. Im Catch-Block in Zeile 24 wird allerdings ein Parameter vom Typ `Ausnahme` abgefangen. Trotzdem wird der Catch-Block aufgerufen, weil ein `Error` ja eine `Ausnahme` ist, d.h. davon abgeleitet ist. Da die `Handle`-Funktion virtuell ist, wird die richtige Funktion aufgerufen und das Programm beendet sich mit der Meldung:

```
1 error: künstlicher Fehler
```

Dass Exceptions auch aus Funktionen “heraushüpfen” können, sieht man im nächsten Beispiel:

```
1  void TutIrgendwas ()
2  {
3    throw Error ("künstlicher Fehler");
4    cout << "Irgendwas" << endl;
5  }
6
7  int main ()
8  {
9    try
10   {
11     TutIrgendwas ();
12   }
13   catch (const Ausnahme &error)  { error.Handle (); }
14 }
```

Hier wird von Zeile 3 direkt zu Zeile 13 gesprungen und das Programm mit derselben Fehlermeldung wie oben beendet. `"Irgendwas"` wird natürlich nicht ausgegeben. Wenn man nun allerdings in der Funktion die Exception auffängt, wird der Catch-Block in `main` nicht aktiv:

```
1 void TutIrgendwas ()
2 {
```

```
3   try
4   {
5     throw Error ("künstlicher Fehler");
6     cout << "Irgendwas" << endl;
7   }
8   catch (Error const &error) {}
9 }
```

Hier wird zwar `"Irgendwas"` nicht ausgegeben, es wird aber auch keine Fehlermeldung ausgegeben. Das Programm fährt fort, als ob nichts passiert wäre, weil der Catch-Block in `TutIrgendwas` nichts unternimmt (`{}`). Man kann allerdings eine bereits geworfene und aufgefangene Exception erneut "weiterwerfen", indem man im Catch-Block `throw` ohne Parameter aufruft:

```
1 void TutIrgendwas ()
2 {
3   try
4   {
5     throw Error ("künstlicher Fehler");
6     cout << "Irgendwas" << endl;
7   }
8   catch (Error const &error)
9   { cerr << "In TutIrgendwas: ";
10    error.PrintMeldung ();
11    throw;
12  }
13 }
```

Hier wird in Zeile 12 die Exception weitergereicht, nachdem eine eigene Meldung ausgegeben wurde. Es werden also beide Catch-Blöcke aktiv.

Was es noch zu sagen gibt:

- Natürlich können mehrere Catch-Blöcke vorhanden sein, um alle möglichen Exceptions abzufangen.
- Es gibt einen *Default-Catcher*, der alle ansonsten nicht abgefangenen Exceptions fängt. Es sollte immer am Schluss stehen. Er wird geschrieben als `catch (...)` mit drei Punkten statt dem Exception-Parameter.
- C++ sollte immer wissen, welche Exceptions entstehen können. Wird aber eine Funktion dazugelinkt, weiß man nicht, welche Exceptions diese Funktion werfen kann. Um diese Information zur Verfügung zu stellen kann man bei der Funktionsdeklaration (am besten im Header-File) eine "function throw list" angeben. Zum Beispiel so:

  ```
  1 void TutIrgendwas ()  throw (Error, char *, std::exception);
  ```

  Eine Funktion darf nur Exceptions werfen, die in dieser Liste angeführt sind, sofern sie vorhanden ist.

9 Input/Output in C++

Das Prinzip der Streams zur Ein-/Ausgabe von Daten wurde auch in C++ beibehalten. Allerdings sind die Streams in C++ Klassen-basiert. Die Stream-Klassen heißen `istream`, `ostream`

und `iostream` und werden in den Header-Files `<istream>` und `<ostream>` deklariert. Davon werden einige andere Klassen und Objekte abgeleitet, die zur I/O verwendet werden. All diese Klassen sind von einer Basis-Klasse `ios` abgeleitet. Das muss man wissen, wenn diverse Flags eines Streams abgefragt werden. Diese sind nämlich alle in `ios` deklariert.

Die wichtigsten I/O-Objekte sind `cin`, `cout` und `cerr`. Sie stehen für die altbekannten I/O-Kanäle zur Eingabe, Ausgabe und Error-Ausgabe. Sie werden meistens mit dem überladenen `<<`-Operator beschrieben bzw. mit `>>` gelesen. Diese Objekte sind verfügbar, wenn man das `<iostream>`-Header-File inkludiert.

Zur Abfrage des Zustands eines Streams haben diese Klassen folgende Funktionen, die einen Boolschen Wert liefern:

- `eof ()` teilt mit, ob das Ende eines Files oder Streams erreicht wurde.
- `fail ()` teilt mit, ob ein Fehler aufgetreten ist. EOF ist übrigens noch kein Fehler, aber der Versuch, nach EOF zu lesen, *ist* ein Fehler.
- `good ()` gibt an, dass weder EOF noch irgendein Fehler aufgetreten ist.

Wenn so ein Fehlerflag in einem Stream gesetzt wurde, bleibt es üblicherweise erhalten. Falls die Situation aber gelöst werden kann und mit Lesen oder Schreiben fortgefahren werden soll, dann kann `clear ()` aufgerufen werden, um die Flag rückzusetzen.

Um das Format der Ein-/Ausgabe zu beeinflussen, gibt es mehrere Möglichkeiten. Zuerst gibt es die `setf()`-Funktion, der einige Flags übergeben werden können. So manipuliert man z.B. mit `cout.setf(ios::hex,ios::basefield)` den Ausgabekanal so, dass Zahlen in Hexadezimal-Format ausgegeben werden. Des weiteren gibt es Funktionen wie `precision()`, mit der man die Anzahl der Nachkommastellen bei der Ausgabe von Gleitkommazahlen abfragen oder verändern kann.

Dann gibt es noch Modifiers, das sind spezielle Objekte, die man in den `<<`-Operator stecken kann und die weitere Ausgabe beeinflussen können. Mit Modifiern kann man im Prinzip das gleiche machen wie mit `setf()` und den anderen Funktionen. Um sie benutzen zu können muss `<iomanip>` inkludiert werden. Sie sind im Namespace `std` deklariert und können daher ohne Präfix verwendet werden, wenn `using namespace std;` gesetzt ist. Zum Beispiel gibt `cout<<hex<<n;` die Zahl `n` hexadezimal aus. Der bekannteste Modifier ist `endl`, der ein Newline-Zeichen in den Stream einfügt.

Die Streams haben außerdem einen Konvertierungs-Operator zum Typ `bool` definiert. Das heißt, man kann einen Stream als Boolschen Wert interpretieren. Dieser Wert entspricht `!fail()`.

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main ()
6  {
7    while (cin)
8    {
9      double d;  int i;  string s;
10     cin >> d >> i >> s;
11     if (cin)
12     {
13       cout << s;
```

```
14        cout.setf (ios::fixed, ios::floatfield);
15        cout.precision (3);
16        cout.width (8);
17        cout << d << ' ' << i << endl;
18      }
19    }
20  }
```

Dieses Beispiel bildet das Beispiel aus Abschnitt 6.1 in C++ nach, außer dass nicht ermittelt wird, wieviele der Variablen im Fehlerfall korrekt gelesen wurden. In Zeile 1 wird der Header für `cin` und `cout` inkludiert. In Zeile 7 wird eine Schleife solange durchgeführt, bis der `bool`-Konverter `false` liefert, d.h. bis ein Fehler auftritt oder das Ende der Eingabe erreicht wurde. Danach werden drei Variablen eingelesen, ein Integer, ein Float und ein String. Falls das Einlesen erfolgreich war (Zeile 11), werden die Variablen wieder ausgegeben.

Dabei wird in Zeile 14–16 der Float-Wert `d` speziell formatiert. Wem das komplizierter erscheint als in C mit `printf`, der hat natürlich recht. Es geht aber auch kürzer, und zwar mit Modifiers:

```
1  #include <iomanip>
2  ...
13     cout << s << fixed << setprecision(3) << setw (8) << d << ' ' << i << endl;
```

Auf Dateien kann in C++ mit den abgeleiteten Klassen `ifstream`, `ofstream` und `fstream` zugegriffen werden (lesend, schreibend und beides, respektive). Folgendes Beispiel reimplementiert das Beispiel aus Abschnitt 6.3, das `abc` in einem File durch `xyz` ersetzt, in C++.

```
1  #include <iostream>
2  #include <fstream>
3
4  using namespace std;
5
6  int main (int argc, char *argv[])
7  {
8    fstream F (argv[1], ios::in | ios::out);
9    if (F.fail ()) { perror (argv[1]); exit (1); }
10
11   char c, state = 'a';
12   F >> c;
13   while (F)
14   {
15     if (c == state)
16     {
17       if (state == 'c')
18       {
19         F.seekp (-3, ios::cur);
20         F << "xyz";
21         state = 'a';
22       }
23       else state ++;
24     }
25     else
```

```
26      {
27        if   (c == 'a')
28             state = 'b';
29        else state = 'a';
30      }
31      F >> c;
32    }
33  }
```

Es wird `<fstream>` inkludiert und ein `fstream`-Objekt definiert (Zeile 8). Die Flags `ios::in | ios::out`, zusammengeodert, weil es sich um Bits eines Integer-Typs handelt, geben an, dass man lesen und schreiben will. Danach wird mit `fail()` überprüft, ob das Öffnen des Files erfolgreich war. Wenn nicht, wird die gute alte `perror`-Funktion aufgerufen, die den Grund für den Fehler auf `stderr` (bzw. hier `cerr`) ausgibt.

Zeichen werden mit dem `>>`-Operator gelesen (Zeile 12, 31). Die Positionierung im File funktioniert mit der Funktion `seekp` (Zeile 19). Es ist also trotz allem alles sehr ähnlich zu C.

Weitere wichtige Funktionen sind noch `getline`, mit der eine ganze Zeile aus einem Stream gelesen werden kann, und `read` und `write`, mit denen unformatierte binäre Daten gelesen und geschrieben werden können.

Eine häufig genutzte Technik ist, den `<<`-Operator zu überladen, um eigene Klassen ebenso elegant in einen Stream ausgeben zu können wie die primitiven Datentypen. Das geht so:

```
1 ostream &operator<< (ostream &s, Konto const &K)
2 { s << K.Name << ": " << K.Guthaben;
3   return s;
4 }
```

Dieser `operator<<` ist eine globale Funktion, also keine Member-Funktion von `ostream`. Der erste Parameter, also der Stream `s`, steht bei der Anwendung dann links vom `<<`, der zweite rechts. Der Operator muss den Stream returnieren, damit eine `<<`-Kette in üblicher Weise möglich ist. Jetzt könnte man diesen Operator z.B. auf folgende Weise anwenden:

```
1   Konto K ("Börsl");
2   cout << K << endl;
```

Allerdings bekommen wir jetzt noch einen Übersetzungsfehler, weil der `operator<<` auf `private`-Members zugreift. Dieses Problem kann man auf zwei Arten lösen. Einerseits können man den Ausgabe-Code in eine Member-Funktion verlagern, die einen `ostream` als Parameter akzeptiert (z.B. `Print(ostream &s)`), und von `operator<<` aus diese aufrufen. Andererseits könnte man einfach die `operator<<`-Funktion in `Konto` als `friend` deklarieren.

10 Standard Template Library

Wir erinnern uns an das Beispiel des Sortier-Algorithmus in Abschnitt 8.15. Der Algorithmus operierte auf einem Array aus Elementen eines Template-Typen. In einer Variante konnte auch die Sortierreihenfolge auf Basis eines Funktionsobjekts parametrisiert werden. Ein weiteres Beispiel hatten wir beim Such-Algorithmus in Abschnitt 5.4. Dort konnte eine Vorwärts- oder Rückwärts-Suchrichtung über eine Iterator-Funktion parametrisiert werden.

Damit hätten wir schon die wichtigsten Bestandteile der Standard-Template-Library beisammen:

Container sind Behälter für Daten wie Arrays, Lists, Stacks, Fifos, Sets, etc. ...

Iteratoren sind Zeiger in Container, die dazu benutzt werden, die Elemente des Containers zu iterieren.

Funktions-Objekte werden dazu verwendet, die Daten eines Containers zu verknüpfen, also sie z.B. zu addieren, multiplizieren, zu vergleichen, etc. ...

Algorithmen werden dazu verwendet, Container bzw. deren Elemente mit Hilfe von Iteratoren und Funktions-Objekten zu durchsuchen, zu modifizieren, zu sortieren, etc. ...

Der Typ der Daten in den Containern, sowie die zu verwendenden Iteratoren und Funktions-Objekte sind als Templates parametrisierbar. Die durchdachte und ineinandergreifende Organisation all dieser Elemente erzeugt ein Fülle von Methoden und Werkzeugen, um Problemstellungen aller Art schnell und optimal zu lösen.

Im Folgenden werden einige der wichtigsten Elemente vorgestellt.

10.1 Container und Iteratoren

In der STL werden Container danach unterschieden, was sie für den Benutzer zur Verfügung stellen und was sie verbieten. Die dahinterliegende Implementierung ist dabei nebensächlich, hat aber für die Vorstellung der involvierten Vorgänge und zur Abschätzung des Laufzeitverhaltens natürlich eine gewisse Bedeutung.

Kriterien zur Einteilung der Konzepte und Template-Klassen sind Darstellungspotential, Zugriffsmöglichkeiten und die Komplexität gewisser Operationen. Unter Darstellungspotential ist hier zu verstehen, ob z.B. die Elemente eine Reihenfolge besitzen oder nicht; Zugriffsmöglichkeiten betreffen z.B. den wahlfreien Zugriff gegenüber der bloßen Iterierbarkeit der Elemente; und die Komplexität z.B. des Einfügens eines neuen Elements ist meistens von logarithmischer Komplexität (bezüglich der Größe des Containers), manchmal ist aber sogar konstante Komplexität garantiert.

Alle Container verwalten den Speicher für ihre Elemente selbst, d.h. wenn Elemente eingefügt werden, wird – wenn nötig – automatisch Speicher alloziert. Die Container dürfen sogar mehr Speicher als notwendig allozieren. Das macht Sinn, wenn dadurch vermieden werden kann, dass bei jedem Einfügen ein ganzes Array neu alloziert werden muss, was die Komplexität der Einfügen-Operation von konstant auf linear steigen lassen würde.

Alle Container sind in eigenen Header-Files definiert. Diese heißen gleich wie die Container, und zwar komischerweise *ohne* den `.h`-Suffix. Wenn also ein `vector<xyz>` deklariert werden soll, muss man vorher `#include <vector>` machen.

10.1.1 Sequenzen

Sequenzen haben die charakteristische Eigenschaft, dass ihre Elemente eine Reihenfolge haben, die nicht aus den Elementen selbst entsteht, vergleichbar mit "geordneten n-Tupeln" aus der Mathematik.

Es gibt zwei Hauptkriterien, nach denen Sequenzen unterschieden werden. Das erste ist die Komplexität des Einfügens eines Elements am Anfang bzw. Ende der Sequenz. Während ein `vector` nur beim Einfügen am *Ende* lineare Komplexität garantiert, ist das bei `deque` und `list` auch am Anfang garantiert, bei `slist` wiederum *nur* am Anfang.

Das zweite Kriterium ist die Zugriffsmöglichkeit: `vector` und `deque` gewährleisten wahlfreien Zugriff, d.h. die Elemente können per Index abgerufen werden. `list` ist im Prinzip eine

doppelt verkettete Liste und kann nur per Iterator vorwärts und rückwärts iteriert werden. `slist` ist einfach verkettet, hier ist nur ein Vorwärts-Iterator möglich.

`vector` und `deque` stellen im Prinzip einfache eindimensionale Arrays dar. Als repräsentatives Beispiel wollen wir hier `vector` vorführen. Die anderen Sequenzen sind in der Bedienung ganz ähnlich.

```
1  vector<int> a;
2  for (unsigned i = 0; i < 10; i ++)
3    a.push_back (i+100);
4  a[5] = 123;
```

Hier wird in Zeile 1 ein Vektor `a` mit `int`s als Elementen definiert. Ohne Konstruktor-Parameter enthält `a` anfangs keine Daten. Mit der Funktion `push_back` werden dann in Zeile 3 einige (10) Werte jeweils am Ende des Vektors eingefügt. Auf die Elemente kann man per Index mit dem `[]`-Operator zugreifen, sowohl lesend als auch schreibend, wie in Zeile 4 angeführt.

Vektoren können auch leicht als Ganzes kopiert werden:

```
1  vector<int> b;
2  b = a;
3  cout << b.size () << ' ' << b[5] << endl;
```

Hier enthält *b* nach der Zuweisung die gleichen Daten wie `a`. `b` ist dann nicht wie bei normalen C-Arrays einfach ein Pointer auf den selben Speicherbereich sondern enthält Kopien der Daten. Mit der Funktion `size()` kann die Länge eines Vektors abgefragt werden.

Gibt man als Konstruktor-Parameter einen Wert an, dann wird der Vektor mit dieser Größe angelegt. Die Elemente werden mit dem Default-Konstruktor angelegt. Gibt man nach der Größe noch einen Wert vom Typ eines Elements an, dann werden die Elemente des Vektors auf diesen Wert initialisiert.

```
1  vector<int> c (10);
2  vector<int> d (10, 234);
```

In Zeile 1 wird ein Vektor `c` mit 10 Elementen angelegt, die auf 0 initialisiert werden. In Zeile 2 passiert das gleiche, nur werden die Werte alle auf 234 initialisiert.

Iteratoren haben Typen, die innerhalb der Template-Klasse des Containers deklariert sind. Man spricht diese Typen also mit `Container::iterator` an. Dadurch hat jeder Container-Typ einen eigenen passenden Iterator-Typ. Konkrete Iteratoren, die in einen Container hineinzeigen, erzeugt man meist mit den Member-Funktionen `begin()` und `end()`, die an den Beginn und ans Ende des Containers zeigen, wobei `end()` eigentlich gar nicht auf ein Element des Containers zeigt, sondern quasi "schon darüber hinaus". `end()` repräsentiert den Zustand, den ein Iterator hat, nachdem er über das letzte Element hinaus bewegt wurde.

```
1  for (vector<int>::iterator i = a.begin (); i != a.end (); i ++)
2    cout << *i << ' ';
3  cout << endl;
```

Hier sieht man eine einfache Iteration durch die Elemente des Vektors `a`. Zuerst wird der Iterator `i` vom Typ `vector<int>::iterator` erzeugt und auf den Beginn des Vektors gesetzt, er zeigt also anfangs auf das Element `a[0]`. Danach wird `i` durch die Operation `i++` immer um ein Element

weitergerückt, bis `i` den Wert `a.end()` hat, was anzeigt, dass der ganze Vektor verarbeitet wurde. Das Element, auf das i zeigt, erhält man mit `*i` (Zeile 2).

Ein weiteres Beispiel soll zeigen, wie man Teile eines Vektors herausnehmen und kopieren kann.

```
1  vector<int>::iterator B = a.begin () + 3;
2  vector<int>::iterator E = a.begin () + 7;
3  vector<int>::iterator P = c.begin () + 5;
4  c.insert (P, B, E);
```

Hier wird die Member-Funktion `insert` verwendet, um mit Hilfe von Iteratoren Elemente in einen Vektor `c` einzufügen. `insert` erhält als ersten Parameter einen Iterator `P`, der anzeigt, an welche Stelle die neuen Elemente eingefügt werden sollen. Als zweiten und dritten Parameter erhält es die Iteratoren `B` und `E`, die an den Beginn und das Ende eines Bereiches eines anderen Containers `a` zeigen. Dieser Bereich wird dann in den Vektor `c` eingefügt. Das heißt, die Elemente ab `P` müssen nach hinten geschoben werden. `a` muss übrigens kein `vector` sein, man könnte auch einen ganz anderen Container verwenden und auf diese Weise z.B. Teile eines geordneten `set`-Containers in einen `vector` einfügen.

Die Iteratoren `B`, `E` und `P` werden hier erzeugt, indem der Beginn-Iterator `begin()` abgefragt wird und dann eine Zahl addiert wird. Das bewirkt, dass der Iterator um diese Zahl an Elementen weitergerückt wird, analog zum `++`-Operator, der nur um ein Element weiterrückt. Im obigen Beispiel wurden benannte Iteratoren verwendet. Man könnte aber auch anonyme Iterator-Objekte verwenden, dann wird die Anweisung kürzer:

```
1  c.insert (c.begin () + 5, a.begin () + 3, a.begin () + 7);
```

Um die Unterscheidung zwischen veränderbaren und `const`-Objekten zu ermöglichen, gibt es `const_iterator`-Typen. Diese funktionieren gleich wie die normalen Iteratoren, lassen aber keinen schreibenden Zugriff auf den Container zu. Funktionen, die Container-Objekte nur lesend verarbeiten, sollten diese Container-Objekte also in der Parameter-Liste immer als `const` deklarieren und nur `const_iterator`-Iteratoren verwenden. Folgendes Beispiel ist eine Funktion zur Ausgabe eines Vektors (Elemente durch Komma getrennt und mit Klammern drumrum) auf einen `ostream` mittels überladenem `<<`-Operator (siehe Abschnitt 9).

```
1  template<typename T>
2  ostream &operator<< (ostream &s, vector<T> const &arr)
3  {
4    s << '(';
5    for (typename vector<T>::const_iterator i = arr.begin (); i != arr.end (); i ++)
6    {
7      if (i != arr.begin ())  s << ',';
8      s << *i;
9    }
10   s << ')';
11 }
```

Damit ist es möglich, einen Vektor so auszugeben:

```
1  cout << a << endl;
```

Es gibt auch noch einen `reverse_iterator`. Dieser iteriert die Elemente in umgekehrter Reihenfolge. Solche Iteratoren können mit `rbegin()` und `rend()` erzeugt werden, wobei diesmal `rbegin()` ans Ende des Vektors zeigt. Auch diese Iteratoren gibt es natürlich in einer `const`-Version.

Weitere Sequenzen sind `deque`, `list` und `slist`, deren Eigenschaften oben schon angeführt wurden. Die Bedienung ist ähnlich, mit ein paar Ausnahmen, dass z.B. bei List der `[]`-Operator nicht vorhanden ist aber eine Funktion `reverse()` existiert. Es gibt natürlich viele weitere Funktionen in all diesen Containern, diese können in geeigneten Quellen nachgeschlagen werden.

10.1.2 Assoziative Container

Assoziative Container sind dadurch gekennzeichnet, dass sie keine Reihenfolge der Elemente haben, bzw. die Reihenfolge sich aus den Elementen selbst ergibt und nicht aus der Reihenfolge, in der sie eingefügt wurden. Sie sind daher sinnvollerweise sortiert.

Es gibt zwei Kriterien, nach denen sich assoziative Container unterscheiden. Das erste betrifft den Schlüssel, nach dem die Elemente des Containers sortiert werden. Der Schlüssel kann das Element selbst sein, dann heißt der Container `set`. Ist der Schlüssel extra anzuführen, dann beinhaltet der Container Paare von Objekten. Ein Element eines solchen Paares ist das eigentlich zu speichernde Objekt, das andere ist der Schlüssel. So ein Container stellt also eine Zuordnung her zwischen einem Schlüssel und einem Objekt, und heißt daher `map`.

Das zweite Kriterium ist, ob zwei gleiche Elemente in einem Container enthalten sein dürfen. Falls nicht, entspricht das einer Menge, wie sie in der Mathematik verwendet wird. Falls doch, erhält der Name des Containers den Zusatz `multi`, also `multiset` und `multimap`. `set` und `multiset` sind in `<set>` definiert, `map` und `multimap` in `<map>`.

Hier ein Beispiel für `set`:

```
1  set<int> S;
2  S.insert (8);  S.insert (2);  S.insert (6);  S.insert (4);
3  for (set<int>::iterator i = S.begin (); i != S.end (); i ++)
4    cout << *i << ' ';
5  cout << endl;
```

Hier wird eine Menge aus *int*s deklariert und in Zeile 2 vier Werte eingefügt. Danach werden die Werte in einer Iterator-Schleife ausgegeben. Es zeigt sich, dass die Werte sortiert sind (`2 4 6 8`).

Die `find`-Funktion kann man verwenden, um ein Objekt in der Menge zu lokalisieren. Wird die Funktion nicht fündig, gibt sie den `end()`-Iterator aus.

```
1  for (;;)
2  {
3    int k;  cin >> k;
4    if (!cin) break;
5    if (S.find (k) != S.end ())
6      cout << "ist vorhanden\n";
7    else
8      cout << "ist nicht vorhanden\n";
9  }
```

Dieses Programmsegment lässt den Benutzer eine Zahl eingeben (Zeile 3) und diese wird dann in der Menge `S` gesucht (Zeile 5). Dann wird ausgegeben, ob die Zahl in `S` vorhanden ist oder nicht. Der große Vorteil des `set`-Containers ist nicht nur das Vorhandensein der `find`-Funktion,

sondern dass diese Funktion garantiert logarithmische Komplexität hat. Die Implementierung der Klasse basiert nämlich (zurzeit in der gcc-Version) auf der Methode der Rot-Schwarz-Bäume, eine Variante der selbst-balanzierenden binären Bäume.

Die `set`-Klasse hat auch noch einen zweiten Template-Parameter, der den Typ eines Funktions-Objekts angibt, das zum Vergleichen von Elementen benutzt werden kann. Dieser Parameter bestimmt also die Sortierordnung der Menge. Default ist die Kleiner-Relation. Auf folgende Weise kann man den Typ eines solchen Funktions-Objekts selber erzeugen:

```
1 struct intgt
2 { bool operator() (int a, int b)  { return a > b; } };
```

Das Funktions-Objekt ist im Prinzip eine Klasse (hier einfach ein `struct`), die nichts als den überladenen `()`-Operator enthält. Dieser soll zwei Parameter vom Typ der Mengen-Elemente akzeptieren und einen Boolschen Wert ausgeben, der den Wert des Vergleichs der zwei Elemente angibt. Hier ermitteln wir einfach `a>b`, also die Größer-Relation statt der Kleiner-Relation. Gibt man diesen Typ nun als Template-Parameter bei `set` an, wird der Container umgekehrt sortiert.

```
1   set<int, intgt> T;
2   T.insert (8);  T.insert (2);  T.insert (6);  T.insert (6);  T.insert (4);
```

`T` enthält die Zahlen nun in der Reihenfolge `8 6 4 2`. Des zweite `insert` der Zahl 6 bleibt wirkungslos, da jedes Objekt in `set` nur einmal vorkommen darf. Den Typ des obigen Funktions-Objekts kann man aber auch einfacher definieren, nämlich so:

```
1   set<int, greater<int> > T;
```

Für die gebräuchlichsten Vergleichs-Operatoren gibt es also fixfertige Templates, um passende Funktions-Objekte zu erzeugen. Default ist meistens `less<>`.

Um die Zahlen aus der Menge `S` zu übernehmen, könnte man folgende Initialisierung anwenden:

```
1   set<int, intgt> T (S.begin (), S.end ());
```

Der hier verwendete Konstruktor akzeptiert also zwei Iteratoren und trägt alle Objekte in sich ein, die sich in dem Bereich (range) zwischen diesen zwei Iteratoren befinden. Interessant ist dabei, dass simple Pointer auch als Iteratoren gelten. Hat man also ein Array, dann gilt der Array-Pointer, der ja auf das erste Element des Arrays zeigt, als `begin()`-Iterator. Der `end()`-Iterator sollte auf das fiktive Element zeigen, das dem letzten Element des Arrays folgt. Das erreicht man am leichtesten, indem man zum Array-Pointer die Array-Größe dazuzählt.

```
1   int Ints[] = {1,5,3,7};
2   set<int> T (Ints, Ints+4);
```

Nun zu `map`. `map` hat zwei Template-Parameter, einen für den Schlüssel und einen für das Objekt. Nehmen wir einmal an, beides wären `int`s. Dann könnte das so aussehen:

```
1   map<int, int> M;
2   M[1] = 8;  M[5] = 4;  M[3] = 2;
3   for (map<int,int>::iterator i = M.begin (); i != M.end (); i ++)
4     cout << (*i).first << "->" << (*i).second << "  ";
5   cout << endl;
```

`M` ist die `map`. In Zeile 2 werden mit dem `[]`-Operator Schlüssel und Werte in `M` eingetragen. Der `[]`-Operator sucht den Schlüssel in `M`, und wenn er ihn nicht findet, trägt er ein Paar bestehend aus dem Schlüssel und einem leeren Objekt in `M` ein. Danach gibt er eine Referenz auf das (leere) Objekt zurück. Mit dem `=`-Operator wird das Objekt dann beschrieben und damit ist ein Zuordnungs-Paar eingetragen. Falls der Schlüssel schon vorhanden war, wird das zugeordnete Objekt überschrieben.

In Zeile 3–4 werden dann in üblicher Weise mit einem Iterator die Elemente von `M` iteriert. Die Elemente sind `pair<int,int>`-Objekte. Die Klasse `pair` ist eine Hilfs-Template-Klasse der STL. Sie hat zwei `public`-Members, `first` und `second`. `first` enthält den Schlüssel und `second` das Objekt. Also bekommt man z.B. mit `(*i).second` das Objekt, auf das der Iterator `i` zeigt. Die Ausgabe des Programms ist:

```
1 1->8  3->2  5->4
```

Die Einträge sind also nach dem Schlüssel sortiert.

Man kann aber auch komplexere Daten sowohl für die Objekte als auch für den Schlüssel verwenden. Hier ein Beispiel mit Strings:

```
1  map<string, string> N;
2  N["Kutil"] = "Rade";  N["Uhl"] = "Andreas";  N["Vajtersic"] = "Marian";
3  for (;;)
4  {
5    cout << "Nachname: ";
6    string s;  cin >> s;
7    if (!cin) break;
8    map<string,string>::iterator i = N.find (s);
9    if (i != N.end ())
10     cout << "Vorname: " << (*i).second << endl;
11   else
12     cout << "nicht gefunden\n";
13 }
14 cout << endl;
```

`N` ist eine `map` von `string` zu `string` und ordnet Nachnamen ihren Vornamen zu. Eine Schleife fragt den Benutzer nach Nachnamen und sucht diese in `N`. Falls dieser gefunden wird, wird der zugehörige Vorname ausgegeben, ansonsten, dass nichts gefunden wurde. Man beachte, dass hier die Strings in die `map` hineinkopiert werden und dort verwaltet werden.

10.2 Algorithmen

Es gibt eine ganze Reihe von Algorithmen, die man auf Container anwenden kann. Durch die Parametrisierung mit Funktionsobjekten kann damit eine große Anzahl an Aufgabenstellungen bewältigt werden. Die Algorithmen sind allesamt im Header-File `<algorithm>` definiert.

Sehen wir uns zuerst ein einfaches Beispiel an. Mit dem `count_if`-Algorithmus wollen wir zählen, wie viele Elemente in einem Array größer als 10 sind. Dazu brauchen wir zuerst ein Funktions-Objekt `gt10`, das überprüft, ob ein Wert größer als 10 ist.

```
1 #include <iostream>
2 #include <vector>
3 #include <algorithm>
4
```

```
5  using namespace std;
6
7  struct gt10 { bool operator() (int a) const { return a > 10; } };
8
9  int main ()
10 {
11   int tmp[] = {13, 25, -5, 7, 15};
12   vector<int> a (tmp, tmp + 5);
13   cout << count_if (a.begin (), a.end (), gt10 ()) << endl;
14 }
```

Dieses Funktionsobjekt ist in Zeile 7 definiert, oder zumindest sein Typ `gt10`. In Zeile 11-12 definieren wir uns ein `int`-Array `a` mit fünf Elementen. Man beachte übrigens den Konstruktor von `vector`, der die zwei Pointer `tmp` und `tmp+5` als Iteratoren interpretiert und den damit definierten Range in sich aufsaugt. Nun zum eigentlichen: In Zeile 13 wird `count_if` aufgerufen mit zwei Iteratoren, die wiederum den Range definieren, in dem gezählt wird. Der dritte Parameter ist unser Funktions-Objekt. Mit dem `()`-Operator wird aus dem Typ `gt10` ein anonymes Objekt konstruiert. Das Ergebnis von `count_if` wird dann gleich ausgegeben, es ist `3`.

Aber es geht noch einfacher. Wir müssen uns das Funktions-Objekt gar nicht selber stricken, sondern können STL-Templates verwenden, um uns on-the-fly eines zu erzeugen. Das geht so:

```
13   cout << count_if (a.begin (), a.end (), bind2nd (greater<int>(), 10)) << endl;
```

Zuerst wird mit `greater<int>()` ein *binäres* Funktionsobjekt erzeugt, d.h. ein Funktionsobjekt mit zwei Parametern. Wir wollen aber ein *unäres* mit nur einem Parameter. Dazu müssen wir den zweiten Parameter des binären `greater` mit dem konkreten Wert `10` binden. Das geht mit `bind2nd`. Und schon haben wir einen unären Operator `>10` definiert, anonym instanziert und an `count_if` übergeben.

Etwas komplexer sind schon Mengen-Operatoren. Das folgende Beispiel implementiert den Durchschnitt zweier Mengen von `char`s.

```
1   char ca[] = "qawsedrf";  set<char> a (ca, ca + strlen (ca));
2   char cb[] = "aysxdcfv";  set<char> b (cb, cb + strlen (cb));
3   set<char> c;
4   set_intersection (a.begin (), a.end (), b.begin (), b.end (),
5                     insert_iterator<set<char> > (c, c.begin ()));
6   for (set<char>::iterator i = c.begin (); i != c.end (); i ++)  cout << *i;
7   cout << endl;
```

In Zeile 1–2 werden zwei Mengen definiert, die jeweils 8 Buchstaben beinhalten, in Zeile 3 noch eine, die den Durchschnitt erhalten soll. In Zeile 4–5 wird mit dem Template-Algorithmus `set_intersection` der Durchschnitt gebildet. Diese Funktion erwartet einen `begin()`- und einen `end()`-Iterator für die erste Menge und zwei ebensolche für die zweite Menge. Zur Ausgabe in die dritte Menge benötigt man auch einen Iterator, allerdings keinen normalen, sondern einen *Insert-Iterator*, mit dem man Elemente in einen Container einfügen kann. Dieser wird hier auch wieder von einer Template-Klasse instanziert, und zwar von `insert_iterator`. Der Template-Parameter dieser Klasse ist der Typ des Containers, in den eingefügt werden soll. Der erste normale Parameter ist der Container selbst, in den eingefügt werden soll, der zweite ist ein Iterator, der angibt, an welcher Stelle das passieren soll. Damit ist der Funktionsaufruf komplett und das Ergebnis der Mengen-Operation in *c* kann in Zeile 6 ausgegeben werden.

Es gibt auch eine kleine Abkürzung zur Erzeugung des Insert-Operators, nämlich die globale Template-Funktion `inserter`. Mit dieser schaut das dann so aus:

```
1  set_intersection (a.begin (), a.end (), b.begin (), b.end (),
2                    inserter (c, c.begin ()));
```

Der wichtigste Algorithmus ist aber natürlich noch immer das Sortieren. Assoziative Container muss man natürlich nicht sortieren, weil sie schon sortiert sind. Wir schauen uns daher an, wie man eine Sequenz sortiert. Es ist wirklich sehr einfach:

```
1  int aa[] = {6, 2, 4, 9, 3, 1, 5, 8, 7};  vector<int> a (aa, aa + 9);
2  sort (a.begin (), a.end ());
3  for (unsigned i = 0; i < 9; i ++) cout << a[i] << ' ';
4  cout << endl;
```

In Zeile 1 wird ein Array (`vector`) erzeugt, in Zeile 2 wird es sortiert, indem man der Funktion `sort` den `begin()`- und `end()`-Iterator übergibt, und in Zeile 3 wird es ausgegeben.

Der Trick mit den Pointern als Iteratoren funktioniert natürlich mit der `sort`-Funktion selbst auch, wir können also direkt ein normales C-Array sortieren:

```
1  int a[] = {6, 2, 4, 9, 3, 1, 5, 8, 7};
2  sort (a, a+9);
3  for (unsigned i = 0; i < 9; i ++) cout << a[i] << ' ';
4  cout << endl;
5  sort (a, a+9, greater<int> ());
6  for (unsigned i = 0; i < 9; i ++) cout << a[i] << ' ';
7  cout << endl;
```

Zusätzlich wird hier in Zeile 5 ein Funktionsobjekt angegeben, das die Sortier-Reihenfolge umkehrt, von default-mäßig `less` auf `greater`.

Man kann natürlich auch selbst eine Reihenfolge programmieren. Folgendes Funktions-Objekt sortiert zuerst die ungeraden Zahlen aufsteigend, danach die geraden Zahlen absteigend:

```
1 struct mylt
2 {
3   bool operator() (int a, int b) const
4   { if (a & 1)
5     { if (b & 1)  return a < b;  else return true; }
6     else
7     { if (b & 1)  return false;  else return a > b; }
8   }
9 };
```

Verwendet wird das Ding ganz einfach so:

```
1  sort (a, a+9, mylt ());
```

und das Ergebnis ist `1 3 5 7 9 8 6 4 2`.

Oben haben wir des öfteren einen ganzen Container ausgegeben. Dieses immer wiederkehrende Programmieren einer Schleife und dann das `cout<<` kann man sich aber ersparen, denn es gibt *Stream-Iteratoren*. Folgendes Beispiel verwendet einen `ostream_iterator`, um ein `set` auszugeben:

```
1  int aa[] = {9, 3, 18, 5, 9, 1, 6, 12, 9};
2  set<int> a (aa, aa + sizeof (aa) / sizeof (int));
3  copy (a.begin (), a.end (), ostream_iterator<int> (cout, " "));
4  cout << endl;
```

In Zeile 1–2 wird die Menge `a` erzeugt. In Zeile 3 wird sie ausgegeben, indem sie Element für Element mit dem Algorithmus `copy` in den `ostream_iterator` kopiert wird. Dieser macht nichts anderes, als dass er Objekte vom angegebenen Template-Typ empfängt, indem der `copy`-Algorithmus `*iterator=objekt`-Operationen durchführt, und dass er diese Objekte mittels `stream<<objekt` auf den Stream ausgibt. Der Stream ist der erste Parameter im Konstruktor des `ostream_iterator`, der zweite Parameter ist der gewünschte Trenn-String, der zwischen den Objekten eingefügt wird.

Für komplexere Klassen kann natürlich in üblicher Weise der globale `operator<< (ostream, Klasse)` überladen werden (siehe Abschnitt 9), dann funktioniert der `ostream_operator` klaglos auch mit solchen Klassen.

11 Threads

Threads sind Prozesse, die sich die Ressourcen teilen. Das heißt hauptsächlich, dass sie auf die selben globalen Variablen zugreifen. Daher muss man aufpassen, dass sich die Threads nicht in die Quere kommen. Programme bzw. Libraries gelten als *thread-safe*, wenn die Funktionen immer richtig funktionieren, auch wenn die selbe Funktion zweimal zur gleichen Zeit von zwei verschiedenen Threads aufgerufen wird. Das erreicht man am leichtesten, indem man einfach keine globalen Variablen verwendet. Aber das ist oft leichter gesagt als getan.

Es gibt mehrere Standards für Threads, die sich zwar alle ähneln, aber doch nicht so richtig kompatibel sind. Eine sehr häufig benutzte und auf den meisten Plattformen unterstützte Thread-Variante sind POSIX-Threads, auch bekannt unter *Pthreads*. Diese wollen wir hier genauer betrachten.

Pthreads werden in einem C-Header-File namens `pthreads.h` deklariert. Die Funktionen befinden sich in einer eigenen Library, die mit `-lpthread` gelinkt werden muss. Jeder Thread, der erzeugt werden soll, erhält Identifikations-Daten (meist einfach eine Id-Nummer), die in einer Variable vom Typ `pthread_t` abgelegt wird. Der Thread wird mit `pthread_create` gestartet. Hier ein Beispiel in C++.

```
1  #include <iostream>
2  #include <pthread.h>
3
4  using namespace std;
5
6  void *PrintThread (void *arg)
7  {
8    cout << reinterpret_cast<char *> (arg) << endl;
9    return 0;
10 }
11
12 int main ()
13 {
14   pthread_t th1, th2;
15   pthread_create (&th1, 0, PrintThread, const_cast<char *> ("hello"));
```

```
16   pthread_create (&th2, 0, PrintThread, const_cast<char *> ("world"));
17   void *r1, *r2;
18   pthread_join (th1, &r1);
19   pthread_join (th2, &r2);
20   cout << "done\n";
21 }
```

Hier werden zwei zusätzliche Threads gestartet, deren Id in `th1` und `th2` zu finden sind (Zeile 14). In Zeile 15–16 werden sie gestartet. Die Funktion `pthread_create` erwartet folgende Parameter: der erste ist ein Pointer auf die zugehörige Datenstruktur; der zweite stellt die gewünschten Thread-Attribute dar, 0 (oder in C `NULL`) gibt die Default-Werte an, welche in den meisten Fällen die passenden sind; der dritte Parameter ist eine Funktion, mit der der Thread gestartet werden soll, diese Funktion muss einen Pointer als Parameter nehmen und einen Pointer als Ergebnis liefern; der vierte Parameter ist der Pointer, der dem neuen Thread als Parameter übergeben wird.

Hier wird jeder neue Thread mit der Funktion `PrintThread` gestartet und ein Pointer auf eine Zeichenkette übergeben. Weil die Zeichenketten hier `const` sind, `pthreads_create` aber einen non-`const`-Pointer verlangt muss hier in Zeile 15–16 ein `const_cast` gesetzt werden. Die Funktion selbst (Zeile 6–10) bekommt einen `void *`-Pointer übergeben. Dieser muss in Zeile 8 zu einem `char *`-Pointer gecastet werden, damit die übergebene Zeichenkette mit `cout<<` richtig ausgegeben wird. Als Ergebnis wird ein Null-Pointer zurückgegeben.

Würde die Funktion `main` nach der Erzeugung der Threads einfach terminieren, dann würden wahrscheinlich die Threads abgebrochen werden, bevor sie irgendetwas ausgeben können (ausprobieren). Der Aufruf von `pthread_join` in Zeile 18–19 löst das Problem. Diese Funktion wartet so lange, bis der Thread terminiert. Der zurückgegebene Pointer des Threads landet dann in der Variable, die als zweiter Parameter angegeben wird.

Manchmal können Threads nicht ganz unabhängig voneinander arbeiten, sondern müssen auf die selbe Ressource, also meistens die selbe globale Variable, zugreifen. Der gleichzeitige Zugriff kann aber zu Inkonsistenzen führen. Daher muss gewährleistet sein, dass sich die Threads bei Zugriff gegenseitig ausschließen. Dazu braucht man einen *Mutex*, das ist eine Struktur, mit der man exklusiven Zugriff beanspruchen kann. Ein Mutex kann gelockt werden. Versucht ein weiterer Thread, das Mutex zu locken, muss er warten, bis der erste Thread ihn wieder freigegeben hat. Das folgende Beispiel veranschaulicht das anhand eines globalen Kontos, auf das zwei Threads mehrfach Buchungen durchführen.

```
1  int Konto = 0;
2
3  pthread_mutex_t mutex;
4
5  void *Buchen (void *arg)
6  {
7    int Betrag = *reinterpret_cast<int *> (arg);
8    for (unsigned i = 0; i < 1000; i ++)
9    {
10     pthread_mutex_lock (&mutex);
11     int tmp = Konto;
12     for (unsigned busy = 0; busy < 100; busy ++);
13     Konto = tmp + Betrag;
14     pthread_mutex_unlock (&mutex);
15   }
16   return 0;
```

```
17 }
18
19 int main ()
20 {
21   pthread_mutex_init (&mutex, 0);
22   pthread_t th1, th2;
23   int Betrag1 = 10, Betrag2 = -10;
24   pthread_create (&th1, 0, Buchen, &Betrag1);
25   pthread_create (&th2, 0, Buchen, &Betrag2);
26   void *r1, *r2;
27   pthread_join (th1, &r1);
28   pthread_join (th2, &r2);
29   pthread_mutex_destroy (&mutex);
30   cout << Konto << endl;
31 }
```

Die Threads führen die Funktion `Buchen` aus, die 1000 mal eine Buchung mit demselben Betrag durchführen, der ihnen als Parameter übergeben wird. Das Lesen und Schreiben des Kontos (Zeile 11 und 13) erfolgt (aus Demonstrationsgründen) zeitverzögert. Daher kann es sein, dass der zweite Thread liest, bevor der erste geschrieben hat. Dann ist das Ergebnis inkonsistent. Da ein Thread `+10` bucht und der andere `-10`, muss am Ende des Programms wieder `0` herauskommen, das ist aber nicht gewährleistet. Daher muss die Anweisungsfolge von Lesen bis Schreiben als *mutual exclusive* ausgeführt werden. Dazu wird ein Mutex verwendet, das ist eine Variable vom Typ `pthread_mutex_t`, die initialisiert werden muss (Zeile 21) und am Ende wieder zerstört werden muss (Zeile 29). Um die Buchung mutually exclusive zu machen, wird vorher (Zeile 10) das Mutex "gelockt" und nachher (Zeile 14) "ungelockt". Auf diese Weise funktioniert das Programm richtig.

Oft ist es wichtig, dass ein Thread darauf warten kann, dass ein gewisser Zustand eintritt. Dass ein solcher Zustand eingetreten ist, muss ihm ein anderer Thread mitteilen. Das wird mit sogenannten Condition-Variablen gemacht. Diese haben den Typ `pthread_cond_t`. Mit `pthread_cond_wait` wird auf ein Ereignis gewartet, mit `pthread_cond_signal` wird dem wartenden Thread signalisiert, dass das Ereignis eingetreten ist. Im folgenden Beispiel wartet der Haupt-Thread (die *main*-Funktion) darauf, dass das Konto überzogen wird, d.h. dass es von `0` auf einen negativen Wert überspringt.

```
1  int Konto = 0;
2  bool Ende = false;
3
4  pthread_mutex_t mutex;
5  pthread_cond_t ueberzogen;
6
7  void *Buchen (void *arg)
8  {
9    int Betrag = *reinterpret_cast<int *> (arg);
10   for (unsigned i = 0; i < 1000; i ++)
11   {
12     pthread_mutex_lock (&mutex);
13     int tmp = Konto;
14     for (unsigned busy = 0; busy < 100; busy ++);
15     Konto = tmp + Betrag;
16     if (tmp == 0 && Konto < 0)
```

```
17        pthread_cond_signal (&ueberzogen);
18      pthread_mutex_unlock (&mutex);
19    }
20    Ende = true;
21    pthread_cond_signal (&ueberzogen);
22    return 0;
23  }
24
25  int main ()
26  {
27    pthread_mutex_init (&mutex, 0);
28    pthread_cond_init (&ueberzogen, 0);
29    pthread_t th1, th2;
30    int Betrag1 = 10, Betrag2 = -10;
31    pthread_create (&th1, 0, Buchen, &Betrag1);
32    pthread_create (&th2, 0, Buchen, &Betrag2);
33
34    while (!Ende)
35    {
36      pthread_mutex_lock (&mutex);
37      pthread_cond_wait (&ueberzogen, &mutex);
38      if (!Ende)  cout << "überzogen!\n";
39      pthread_mutex_unlock (&mutex);
40    }
41
42    void *r1, *r2;
43    pthread_join (th1, &r1);
44    pthread_join (th2, &r2);
45    pthread_cond_destroy (&ueberzogen);
46    pthread_mutex_destroy (&mutex);
47    cout << Konto << endl;
48  }
```

Die Condition-Variable `ueberzogen` (Zeile 5) muss auch initialisiert (Zeile 28) und am Schluss zerstört werden (Zeile 45). In Zeile 37 wartet der Hauptthread auf das Ereignis. Diese Anweisung muss in einen mutual-exclusive-Block geklammert werden (Zeile 36–39) und das dazu verwendete Mutex wird an `pthread_cond_wait` übergeben. Dieses Mutex muss auch dasselbe sein, wie das, in das `pthread_cond_signal` geklammert ist (Zeile 12–18). Mit `pthread_cond_signal` wird in Zeile 17 signalisiert, dass das Konto überzogen wurde, falls es vorher auf `0` war und nun negativ ist. Der Hauptthread wartet so lange auf ein solches Ereignis, bis die globale Variable `Ende` anzeigt, dass ein Thread seine Arbeit beendet hat. Dieses Programm ist etwas unsauber, weil am Schluss in Zeile 21 noch einmal signalisiert wird, ohne das Mutex zu locken und außerdem der zweite Thread am Schluss signalisiert, ohne dass der Haupt-Thread darauf wartet. Aber es funktioniert: der Zufall bestimmt, ob das Konto einmal, mehrere Male oder gar nie überzogen wird.

Eine bekannte Methode, Prozesse zu synchronisieren, ist die Verwendung von Semaphoren. Semaphoren sind in `semphore.h` deklariert. Mit `sem_wait` wird eine Semaphore um eins vermindert, sofern sie einen Wert größer 0 hat, ansonsten wird gewartet, bis ein anderer Thread die Semaphore mit `sem_post` wieder freigibt. Folgendes Beispiel ruft 50 Instanzen der Funktion `Calc` auf, die eine zufällige Menge an (sinnloser) Arbeit verrichtet. Dabei sollen immer genau 10 Threads gleichzeitig arbeiten.

```
1  #include <iostream>
2  #include <pthread.h>
3  #include <stdlib.h>
4  #include <semaphore.h>
5
6  using namespace std;
7
8  sem_t sem;
9
10 void *Calc (void *)
11 {
12   unsigned cnt = rand () % 100000000;
13   for (unsigned i = 0; i < cnt; i ++);
14
15   sem_post (&sem);
16 }
17
18 int main ()
19 {
20    sem_init (&sem, 0, 10);
21    for (unsigned job = 0; job < 50; job ++)
22    {
23      sem_wait (&sem);
24      pthread_t *thread = new pthread_t;
25      pthread_create (thread, NULL, Calc, 0);
26      cout << '.'; cout.flush ();
27    }
28    for (unsigned i = 0; i < 10; i ++) sem_wait (&sem);
29    cout << endl;
30    sem_destroy (&sem);
31 }
```

Die Semaphore (Zeile 8) muss initialisiert (Zeile 20) und am Schluss zerstört werden (Zeile 30). Sie wird auf den Wert `10` initialisiert. Das heißt, die ersten zehn Aufrufe von `sem_wait` (Zeile 23) blockieren nicht und es wird jeweils ein Thread erzeugt (Zeile 25). Beim elften Mal wartet `sem_wait` darauf, dass ein Thread terminiert. In diesem Fall erhöht der Thread die Semaphore (Zeile 15) und weckt damit `sem_wait` wieder auf und es wird ein neuer Thread erzeugt. Wenn alle 50 Threads gestartet wurden, muss das Hauptprogramm noch darauf warten, dass die letzten zehn fertig werden. Dazu wird einfach in Zeile 28 zehn Mal `sem_wait` aufgerufen. Danach kann das Programm beendet werden. Das einzige unschöne an dem Programm ist, dass die `pthread_t`-Ids, die in Zeile 224 alloziert werden, nicht mehr freigegeben werden.

12 Wo finde ich Information?

Hier noch kurz eine kleine Auflistung, wo man zu welchem Thema am leichtesten an detailliertere Informationen kommt.

Thema	Information
`make`	`info make`
C-Syntax	(keine Empfehlung)
C-Library	`man`-Pages (`man 3 intro`)
System-Calls	`man`-Pages (`man 2 intro`)
C++-Syntax	C++-Annotations (`http://www.icce.rug.nl/documents/cpp.shtml`)
STL	`http://www.sgi.com/tech/stl/`
Pthreads	(keine Empfehlung)