Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Fachpraktikum 1 von 6: Systemprogrammierung in C

Communication Systems Group
ETH Zurich

Ariane Keller <ariane.keller@tik.ee.ethz.ch>
Daniel Borkmann <daniel.borkmann@tik.ee.ethz.ch>
Stephan Neuhaus <stephan.neuhaus@tik.ee.ethz.ch>

Inhaltsverzeichnis

1 Einführung

1.1 Motivation

If you get into serious programming, you will have to learn C, the core language of Unix.—Eric S. Raymond, “How To Become A Hacker”

I've been writing C code for over 20 years, and I still love it. This link is a good summary of why. ***It also controls the world, but people don't like to admit that****, it's C's dirty little secret ...*

> *The C Paradox: I don't think C gets enough credit. Sure, C doesn't love you. C isn't about love - C is about thrills. C hangs around in the bad part of town. C knows all the gang signs. C has a motorcycle, and wears the leathers everywhere, and never wears a helmet, because that would mess up C's punked-out hair. C likes to give cops the finger and grin and speed away. Mention that you'd like something, and C will pretend to ignore you; the next day, C will bring you one, no questions asked, and toss it to you with a you-know-you-want-me smirk that makes your heart race. Where did C get it? "It fell off a truck," C says, putting away the boltcutters. You start to feel like C doesn't know the meaning of "private" or "protected": what C wants, C takes. This excites you. C knows how to get you anything but safety. C will give you anything but commitment. In the end, you'll leave C, not because you want something better, but because you can't handle the intensity. C says "I'm gonna live fast, die young, and leave a good-looking corpse," but you know that C can never die, not so long as C is still the fastest thing on the road.* - Anonymous, `http://dis.4chan.org/read/prog/1312655446/4`

—Greg Kroah-Hartman, Linux Kernel Hacker

1.2 Über dieses Dokument

Dieses Dokument stellt eine Schnelleinführung in die Programmiersprache C dar. Als Zielplattform wird die GNU Compiler Collection (gcc, [1]) mit der GNU C Library (glibc [2]) unter Linux auf Intel-Plattformen benutzt. Grundlegende Erfahrung mit C++ oder Java wird vorausgesetzt, sodass der Fokus in diesem Dokument hauptsächlich auf den *Unterschieden* zwischen diesen Sprachen liegt.

C ist eine ziemlich systemnahe Sprache und vieles, an das man sich aus anderen Sprachen gewöhnt hat, muss man in C "von Hand zu Fuss" erledigen. Es gibt keine Sprachmittel für Objektorientierung, keine Garbage Collection, das Typsystem ist recht lose, und so weiter. Das hat Nachteile: Wenn man Objektorientierung will, muss man sie sich selbst bauen, Speicher muss man selbst managen, und bei den Typen muss man aufpassen, weil der Compiler das Aufpassen nur eingeschränkt übernimmt. Es hat aber auch Vorteile: Die Sprache ist vergleichsweise klein und lässt sich rasch erlernen (vergleichen Sie mal die Dicke eines guten vollständigen C-Buchs mit der eines guten vollständigen C++-Buchs) und die Abwesenheit von Aktivitäten "hinter den Kulissen", wie sie bei Operator-Überladungen oder Garbage Colelction leicht auftreten kann, erlaubt es einem, C-Code auch dann zu lesen, zu verstehen und zu ändern, wenn man nur einen Ausschnitt des gesamten Codes kennt.

2 Literaturverweise

2.1 Bücher, Referenzen

- "The C Book": Gutes C Buch, das unter `http://publications.gbdirect.co.uk/c_book/` frei verfügbar ist.
- "The C Programming Language." von Brian W. Kernighan und Dennis Ritchie (ausleihbar von der ETH Bibliothek)
- POSIX (*P*ortable *O*perating *S*ystem *I*nterface) Standard für Unix: `http://pubs.opengroup.org/onlinepubs/9699919799/`
- Sektionen 2 und 3 der manual pages: `man 2<system-call-name>`, `man 3 <function-name>`
- Der Sprachstandard selbst (nichts für Furchtsame Seelen): `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf`.
- Für Fortgeschrittene:
  - GCC C Erweiterungen: `http://gcc.gnu.org/onlinedocs/gcc/C-Extensions.html`
  - GCC Non-Bugs: `http://gcc.gnu.org/bugs/#nonbugs`

2.2 Programmierstil

- Eine gute Beschreibung von Linus Torvalds zum Programmierstil für den Linux Kernel, der auch für normale C-Programme verwendet werden kann: `http://lxr.linux.no/linux+v3.2.1/Documentation/CodingStyle`
- Die GNU Coding Standards werden in allen Projekten der Free Software Foundation verwendet: `http://www.gnu.org/prep/standards/`

2.3 Weitere Werkzeuge (Auswahl)

GNU make. *Das* Werkzeug zum automatisierten Übersetzen. Dokumentation zu finden mit `info make`.

GNU Debugger. *Das* Werkzeug zum fortgeschrittenen Debuggen von C Programmen. Dokumentation zu finden mit `man gdb`.

DDD. Der Data Display Debugger. Sehr praktisch um insbesondere auch komplexere Datenstrukturen zu analysieren. Dokumentation zu finden mit `info ddd`.

Valgrind. Ein gutes Werkzeug um "memory leaks", d.h. nicht freigegebene dynamische Speicherbereiche zu finden. Auch zum generellen Speicher-Profiling geeignet. Programme werden mit einer simulierten CPU ausgeführt. Dokumentation zu finden mit `man valgrind`.

2.4 Typographische Konventionen

Wir verwenden in diesem Dokument eine Reihe von Schriftarten, jede zu einem bestimmten Zweck. `Programmtext` schreiben wie üblicherweise in Schreibmaschinenschrift, genauso wie Befehle, die Sie per Shell an den Computer übermitteln. *Variablen* schreiben wir kursiv. Wir können so zwischen der Variable x und dem Programmtext `x` unterscheiden, was oft nützlich ist. **Schlüsselwörter** und **Typnamen** drucken wir fett (falls sie nicht im `Programmtext` vorkommen). *Dateinamen* schreiben wir kursiv.

3 Dokumentation

Es gibt verschiedene Quellen für Dokumentation zur Sprache C und zu den in verschiedenen Umgebungen verfügbaren Bibliotheksfunktionen. Die wichtigsten Quellen für gcc/glibc werden jetzt vorgestellt.

3.1 Man Pages

Ein Teil der C Bibliotheken ist unter Unix traditionell in "man-pages" dokumentiert. Man-pages sind schnell in jeder Shell zugreifbar über den "man" Befehl.

Man beachte, dass die man-pages unter Umständen mehrere Einträge mit dem gleichen Schlüsselwort enthalten. Hier muss gegebenenfalls die "Sektion" angegeben werden um die Dokumentation des C Befehls anzuzeigen. System Calls sind in Sektion 2 dokumentiert, Library Calls in Sektion 3. Weiterhin sind man-pages keine vollständige Dokumentation, sondern dienen in erster Linie als Kurzreferenz.

Beispiel 3.1. Die Dokumentation zu "man" kann so abgerufen werden:

```
man man
```

Beispiel 3.2. Die Dokumentation zur C-Funktionl *printf*:

```
man printf   # dokumentiert das Programm ``printf''
man 3 printf # dokumentiert die C Bibliotheksfunktion ``printf''
```

Beispiel 3.3. Die Dokumentation zur Linux-Funktion *open*:

```
man open    # dokumentiert das  Programm ``open''
man 2 open  # dokumentiert den C System Call ``open''
```

3.2 Info

Die offizielle Dokumentation für die gcc und die glibc finden sich in den "info" Seiten der entsprechenden Projekte. Die Dokumentation ist ebenfalls über die Websiten [1, 2] erreichbar.

Die info Seiten sind mittels dem Programmpaket "TeXInfo" erstelle Hyperlinkdokumente und können mittels des Programms "info" gelesen werden. In info sind Hyperlinks mittels vorgestelltem "*" gekennzeichnet. Ein info-Tutorial wird durch Eingabe von "h" nach Start von info zugänglich. Die Hilfeseite ist mittels "?" auswählbar.

Beispiel 3.4. Um die Dokumentation zu "info" zu erhalten:

```
info info
```

Beispiel 3.5. Die Dokumentation zur C Funktion “printf” mit info:

1. `info libc`
2. Zu “Function Index” scrollen und auswählen.
3. Zu “printf” scrollen und auswählen.

4 Tutorial

4.1 Das erste Programm

Die folgenden Schritte sind geeignet um ein erstes Programm zu erstellen und ablaufen zu lassen.

1. Geben Sie den folgenden Text mit einem Editor Ihrer Wahl ein. Geben Sie die Zeilennummern und die darauf folgenden Doppelpunkte *nicht* mit ein.

```
1: #include <stdio.h>
2:
3: int main(void)
4: {
5:         printf("Hello World\n");
6:         return 0;
7: }
```

2. Speichern Sie diesen Text als *hello.c*.
3. Übersetzen Sie den Code mittels des Befehls `gcc hello.c -o hello`. Der Compiler erzeugt ein ausführbares Programm mit den Dateinamen *hello*.
4. Starten Sie das Programm mit `./hello`

4.2 Analyse von `hello.c`

Die grösste Struktureinheit von C ist die Quelltext-Datei. Wenn es sich nicht um einen Basisdatentypen handelt (z.B. **int**, **char** oder ähnliche; siehe unten), sind innerhalb einer Datei nur Funktionen und Datentypen sichtbar, die dort deklariert sind.

Die erste Zeile

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

ist die Anweisung an den Compiler, hier die Deklarationen der Datentypen und Funktionen aus der Standard-Ein-Ausgabebibliothek *<stdio.h>* einzufügen und somit dem Programm verfügbar zu machen. Der von ‘standard I/O’ seltsam verkürzte Name ‘stdio’ hat historische Gründe.

Sie können auch eigene Typen und Funktionen deklarieren (wie das geht, erklären wir später). Typischerweise werden solche Funktionen in einer Datei mit der Endung *.h* zusammengefasst. Wenn Sie z.B. einen Datentyp “stack” entwickeln, gehören die Deklarationen dazu in eine Datei *stack.h* und die Implementation in eine Datei *stack.c*. Um anderen Programmteilen dann die Deklarationen aus *stack.h* zur Verfügung zu stellen, schreiben Sie in diesen Programmteil

```
#include "stack.h"
```

am besten ziemlich weit oben. Mehr zu `#include` finden Sie in Abschnitt 4.6

Zeile 3 enthält die Definition einer Funktion mit dem Namen *main*, dem Rückgabetyp **int** (vorzeichenbehaftete ganze Zahl) und keinen Argumenten (**void**). Der Name *main* ist besonders: Er markiert das Hauptprogramm, d.h. denjenigen Programmcode, der beim Programmstart als erstes ausgeführt werden soll. Der Rückgabewert wird dem aufrufenden Programm Ende des Programmlaufes übergeben. Dies geschieht in Zeile 6 mittels '**return** 0'. Mehr zu *main*, insbesondere zum Thema Parameterübergabe von der Kommandozeile, finden Sie in Abschnitt 7.

Zeile 5 enthält nun die einzige Anweisung in *hello.c*. Es ist ein Funktionsaufruf der Funktion *printf*, "print formatted". Beachten Sie, dass Semikolon am Ende der Zeile. Funktionsaufrufe werden in C mit einem Semikolon abgeschlossen.

Wenn Sie die Dokumentation zu *printf* lesen (man-page, info Seiten), sehen Sie dass *printf* in <*stdio.h*> deklariert ist. Das erklärt die erste Programmzeile.

Weiterhin hat *printf* einen Rückgabewert vom Typ **int**, der in unserem Beispiel ignoriert wird. (In C dürfen Rückgabewerte von Funktionen ohne weiteres ignoriert werden. Das ist keine gute Praxis, aber bei *printf* leider einfach Standard.) Weiterhin hat *printf* ein erstes Argument vom Typ '**const char** *', und eine variable Anzahl von weiteren Argumenten. Hier geben wir nur das erste Argument als konstante Zeichenkette direkt an. In dieser Form gibt *printf* die angegebene Zeichenkette auf der Konsole aus.

Die Funktion *printf* wird in Abschnitt 6.7 näher dargestellt. Für den Moment sei nur darauf hingewiesen, dass '`\n`' einen Zeilenumbruch verursacht (dies ist keine C Eigenheit, sondern Unix-typisch) und dass das Zeichen '`%`' im ersten Argument von *printf* eine besondere Bedeutung hat.

In Zeile 7 wird der Funktionskörper der Funktion *main* mit '`}`' abgeschlossen.

4.3 Formatierung, Syntaktische Konventionen

Die Formatierung des Quelltexts spielt für den C-Compiler keine Rolle und ist daher für die Korrektheit des Programms unerheblich. Bevor der Compiler den Quelltext bearbeitet, wird jedoch ein Präprozessor zur Anwendung gebracht. Dieser bearbeitet alle Anweisungen, die in einer Zeile mit einem '`#`' am Anfang stehen und entfernt die Kommentare. Daher ergibt sich eine gewisse Zeilenbindung.

Aber Programme werden nicht wie man denken könnte hauptsächlich für Compiler geschrieben, sondern für andere Menschen. Deshalb ist es schon wichtig, dass Sie lesbaren Code schreiben.

Es ist schon viel über Quellcodeformatierung geschrieben worden. Wir wollen uns dem nicht anschliessen. Wählen Sie eine Quellcodeformatierung, die zu Ihnen passt und bleiben Sie dabei. (Sollten Sie irgendwann professionell Code schreiben, müssen Sie sich unter Umständen an einen Ihnen fremden Stil gewöhnen. Das ist nicht schädlich.) Wenn Sie noch keine grosse Programmiererfahrung und daher keinen eigenen Einrückstil haben, können Sie sich bei den Meistern orientieren. Das wären die Konventionen aus dem C-Buch von Kernighan und Richie, die der Free Software Foundation, oder den Linux Coding Conventions; siehe dazu die Literaturliste.

Eine einheitliche Code-Formatierung nach solchen Standards sorgt dafür, dass Ihr Code für andere leichter lesbar ist und damit leicher im Code-Audit mögliche Fehler gefunden werden können, da man sich nur noch auf den Inhalt des Codes zu konzentrieren braucht statt auch auf den Stil.

Wenn Sie auf die Schnelle Hinweise zur Formatierung brauchen: hier ist ein kleines Programm, das die typischen Elemente beinhaltet, in einem Stil, den wir akzeptabel finden:

```
/* gcd.c -- compute gcd of two positive numbers.
 *
 * Compile with "gcc -O2 -Wall gcd.c -o gcd"
 *
 * Run as "./gcd m n", for example, "./gcd 111 33"
 */
#include <errno.h>
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>

/* First number for which to compute the gcd. */
static unsigned int m = 0;

/* Second number for which to compute the gcd. */
static unsigned int n = 0;

/* Converts a string to an unsigned int, base 10.  Makes
 * sure the string begins with an integer in base 10, the
 * number is positive and can fit into an unsigned int.
 * Exits if not. */
static unsigned int checked_strtou(const char* s) {
  errno = 0;
  long l = strtol(s, NULL, 10);
  if (errno == ERANGE) {
    fprintf(stderr, "%s not a number or too small or too big\n", s);
    exit(EXIT_FAILURE);
  } else if (l <= 0) {
    fprintf(stderr, "numbers must be positive (got %s)\n", s);
    exit(EXIT_FAILURE);
  } else if (l > UINT_MAX) {
    fprintf(stderr, "number %s too large", s);
    exit(EXIT_FAILURE);
  }

  return (unsigned int) l;
}

int main(int argc, const char* argv[]) {
  if (argc != 3) {
    fprintf(stderr, "usage: gcd m n\n");
    fprintf(stderr, "error: wrong number of arguments\n");
    exit(EXIT_FAILURE);
  }

  m = checked_strtou(argv[1]);
  unsigned int om = m; // save m for later

  n = checked_strtou(argv[2]);
  unsigned int on = n; // save n for later
```

```
  unsigned int r = m % n;

  while (r != 0) {
    m = n;
    n = r;
    r = m % n;
  }

  printf("gcd(%u, %u) = %u\n", om, on, n);

  return(EXIT_SUCCESS);
}
```

Dieses Programm weist folgende Elemente auf:

- Am Anfang steht ein kurzer Kommentar zum Programm, was es tut, wie es übersetzt wird und wie man es aufruft.
- Standard-Includedateien wie <*stdio.h*> kommen als Erstes. Wir führen sie in alphabetischer Reihenfolge an, um zu betonen, dass es unerheblich ist, in welcher Reihenfolge sie angeführt werden.
- Jede globale Variable hat einen kurzen(!) Kommentar, der erklärt, wozu sie dient.
- Jedes Objekt wird bei seiner Deklaration oder Definition mit der grösstmöglichen Anzahl an Typmodifikatoren ausgestattet (**const**, **static**), um dem Leser (und dem Compiler) die grösstmögliche Anzahl an Hinweisen zu diesem Objekt zu geben. (Variablen, die nur in der Datei sichtbar sein sollen, in der sie definiert sind, werden **static** definiert, Zeiger, die das Objekt auf das sie zeigen nicht ändern als **const**, usw.)
- Jedes Objekt enthält den am meisten eingeschränkten Typ: Eine Zahl wird nicht als **int** deklariert, wenn sie nur nichtnegative Werte annehmen kann (dafür gibt es **unsigned**), usw.
- Jede Funktion erhält einen kurzen(!) Kommentar, indem kurz dargelegt wird, was sie tun soll.
- Jeder Funktionsaufruf in die Standardbibliothek und jeder Systemaufruf wird auf Erfolg geprüft. Ausnahmen sind *printf* und Verwandte, *exit*, und *va_start* und Verwandte. Im Fehlerfall erfolgt eine Fehlermeldung und eventuell der Programmabbruch.
- Variablen werden so kurz wie möglich vor ihrer ersten Verwendung definiert. (In diesem Programm sind *m* und *n* globale Variablen, weil wir deren Verwendung illustrieren wollten.)

4.4 Fehlerausgaben

C-Compiler erzeugen ausführbaren Code, auch wenn der Quelltext zwar syntaktisch und semantisch korrekte, aber typischerweise so nicht gemeinte Konstruktionen enthält. Daher gibt es zwei Stufen von Diagnosen:

Warnungen sind Hinweise, dass potentielle Fehler existieren, aber dass lauffähiger Code erzeugt wurde.

Entfernen Sie das **return**-Statement und übersetzen Sie die Datei mit dem Befehl `gcc -Wall hello.c`. Betrachten Sie die Fehlerausgabe, die Sie erhalten. Der Compiler teilt Ihnen mit, das der Funktionsrückgabewert nicht gesetzt wurde; in einem solchen Fall ist der Rückgabewert nicht definiert.

Mit `-Wall` wird gcc angewiesen, alle zweifehlaften Konstruktionen anzumahnen. Ohne diese Compiler-Option erhalten Sie Warnungen nur für ernste Probleme. Wenn Sie solche Warnungen erhalten, sollten Sie die Ursache identifizieren und korrigieren. Sie sollten anstreben, Programme zu schreiben, die immer auch bei der Übersetzung mit `-Wall` keine Warnungen ausgeben.

Manche potentiellen Fehler, beispielsweise die Verwendung uninitialiserter Variablen, werden vom Compiler nur gefunden, wenn die Optimierung (und damit die Datenflussanalyse) eingeschaltet ist. Übersetzen Sie daher stets mit `-O2 -Wall`. (Die Option `-O2` schaltet die Optimierung auf Stufe 2.)

Mit `-pedantic` koennen die Warnungen weiter verschärft werden hinsichtlich Konformität zu ISO C. (Weitere Optionen sind unter `http://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html` verfügbar.)

Fehler werden bei der Übersetzung von Dateien ausgegeben, die syntaktische oder semantische Fehler enthalten, so dass kein ablauffähiges Programm mehr erzeugt werden konnte.

Hier ein Potpurri typischer Fehler und ihrer Ursachen. Als Illustration wählen wir wenn möglich das in Abschnitt 4.3 angegebene Programm zur Bestimmung des grössten gemeinsamen Teilers zweier Zahlen.

4.4.1 Vergessenes Semikolon

Hier fehlt das Semikolon nach der Zuweisung zu *errno*:

```
static unsigned int checked_strtou(const char* s) {
  errno = 0
  long l = strtol(s, NULL, 10);
  // ...
}
```

Dieser Code führt zu folgendem Fehler:

```
gcd.c: In function checked_strtou:
gcd.c:18: error: expected ; before long
gcd.c:22: error: l undeclared (first use in this function)
gcd.c:22: error: (Each undeclared identifier is reported only once
gcd.c:22: error: for each function it appears in.)
```

4.4.2 Vergessenes `#include`

Unterlässt man ein bestimmtes `#include`, dann sind die in diesem Header aufgeführten Datenstrukturen, Funktionen und Makros nicht verfügbar, werden also in der Regel dazu führen, dass der Compiler über unbekannte Bezeichner meckert.

Entfernen wir beispielsweise den Header <*limits.h*>, führt das zu:

```
gcd.c: In function checked_strtou:
gcd.c:25: error: UINT_MAX undeclared (first use in this function)
gcd.c:25: error: (Each undeclared identifier is reported only once
gcd.c:25: error: for each function it appears in.)
```

Entfernen wir hingegen <*stdio.h*>, führt das zu einer langen, langen Liste von Fehlern, angeführt von:

```
gcd.c: In function checked_strtou:
gcd.c:20: warning: implicit declaration of function fprintf
```

Sieht man so eine Fehlermeldung, ist sofort klar, dass der Header fehlt. Typischerweise ergibt es keinen Sinn, danach noch weitere Fehler zu suchen. Schneller geht es, wenn man den Header einfügt und dann den Übersetzungsvorgang erneut versucht.

4.4.3 Vergessene Variablendeklaration

Vergessen wir die Deklaration der Variablen *l* in *checked_strtou*, erhalten wir folgende Fehlermeldung, die recht klar auf die Ursache hinweist:

```
gcd.c: In function checked_strtou:
gcd.c:22: error: l undeclared (first use in this function)
gcd.c:22: error: (Each undeclared identifier is reported only once
gcd.c:22: error: for each function it appears in.)
```

4.4.4 Ausblenden von Variablen oder Parametern

Nehmen wir an, wir hätten die Variable *l* nicht *l* genannt, sondern *s*, wie den Parameter. Dann hätte der Bezeichner *s* plötzlich zwei Bedeutungen, was natürlich nicht geht und deshalb eine lange Fehlerkette produziert. Das ist auch dann ein Fehler, wenn *s* und *l* denselben Typ hätten.

```
gcd.c: In function checked_strtou:
gcd.c:18: error: s redeclared as different kind of symbol
gcd.c:16: note: previous definition of s was here
```

4.5 Quickstart: Syntax und Bibliothek

4.5.1 Bedingungen prüfen: if—else, switch—case

Allgemeines Prüfen von Bedingungen:

```
if (isdigit(*p))
  ret[k] = 10*ret[k] + (*p - '0');
else if (*p == '\n') {
  k++;
  if (k < napps)
    ret[k] = 0;
} else
  error("unexpected character `%c' (%02x)", *p, *p);
```

Prüfen, welchen Wert eine Variable hat:

```
char c;
unsigned int d = 0xff;
// ...
switch (c) {
case '0': d = 0; break;
case '1': d = 1; break;
case '2': d = 2; break;
case '3': d = 3; break;
case '4': d = 4; break;
case '5': d = 5; break;
case '6': d = 6; break;
case '7': d = 7; break;
case '8': d = 8; break;
case '9': d = 9; break;
default:
  error("`%c' is not a decimal digit", c);
  break;
}
```

4.5.2 Schleifen: for und while

Iterieren über ein Array:

```
unsigned int counts[UCHAR_MAX];
unsigned int i;

for (i = 0; i < UCHAR_MAX; i++)
  counts[i] = 0;
```

Ein Array a hat **sizeof** a/**sizeof** $a[0]$ Elemente, deshalb kann man auch schreiben:

```
unsigned int counts[UCHAR_MAX];
unsigned int i;

for (i = 0; i < sizeof(counts)/sizeof(counts[0]); i++)
  counts[i] = 0;
```

In der Version C99 des Sprachstandards kann man die Iteration fast schreiben wie in Java oder C++:

```
unsigned int counts[UCHAR_MAX];

for (unsigned int i = 0; i < sizeof(counts)/sizeof(counts[0]); i++)
  counts[i] = 0;
```

4.5.3 Strukturen und Zeiger: struct, ‘ . ’ und ‘–>’

Strukturen deklarieren:

```
struct _int_list_element_t {
  struct _int_list_element_t* next;
  int key;
};
typedef struct _int_list_element_t int_list_element_t;
```

```
// ...
int_list_element_t front;
int_list_element_t *rear;
```

Zugriff auf Strukturelemente:

```
front.next = NULL;     // Direct access is with '.'
front.key = -1;
rear->next = &front;   // Access via pointer is with '->'
rear->key = front.key;
```

4.5.4 Speicherverwaltung: *malloc* und *free*

Anfordern und Freigeben von Speicher. Prüfen Sie *immer* den Rückgabewert von *malloc* (so wie bei jeder anderen Bibliotheksfunktion auch). Geben Sie Speicher stets wieder frei, ausser, Sie sind kurz davor *exit* aufzurufen oder *main* mit **return** zu verlassen.

```
int_list_element_t* p
  = (int_list_element_t*) malloc(sizeof(int_list_element_t));
if (p == NULL) {
  error("can't allocate list element");
  exit(EXIT_FAILURE);
}
p->next = NULL;
p->key = rear->key + 1;
rear->next = p;
p = rear;
rear = rear->next;
free(p);
```

4.5.5 Logische Operatoren, bitweise Operatoren

Die logischen Operatoren dienen zur logischen Verknüpfung von Bedingungen. Ist beispielsweise in *y* eine Jahreszahl gespeichert, ist

```
(y % 4 == 0 && y % 100 != 0) || y % 400 == 0
```

die Bedingung für ein Schaltjahr. Die logischen Operatoren für Konjunktion (und) und Disjunktion (oder) sind dabei gedoppelt, (‘`&&`’, ‘`||`’), wie der Vergleich (‘`==`’). Es gibt auch Bitoperatoren, die auf den bits der jeweiligen Operanden arbeiten; diese bestehen aus nur jeweils einem Zeichen (‘`&`’, ‘`|`’); siehe Abschnitt 6.4.

4.6 Separate Übersetzung

C-Programme können zur besseren Handhabung in mehrere Dateien aufgeteilt werden. Das erlaubt es, funktional Verwandtes in einer Datei zusammenzufassen und von nicht Verwandtem zu trennen. So könnte der Quelltext für einem Editor aufgeteilt werden in Funktionen zur Textrepräsentation in einer Datei und Funktionen zum Suchen und Ersetzen in einer anderen Datei.

Nehmen wir nun an, dass die Funktion zum Suchen einer Zeichenkette im aktuellen Dokument im Pseudocode so aussieht:

```
/* Returns line number of first hit
 * -1 if no hit exists */
int findstring(const char* string_to_find,
               const document* doc) {
  int lineno = 1;
  const char* line = getline(doc, lineno);

  while (line != NULL) { /* NULL: Ende des Dokuments */
    if (match_exists(line, string_to_find))
      return lineno;
    lineno++;
    line = getline(doc, lineno);
  }

  return -1;
}
```

Nehmen wir weiter an, dieser Code sei in einer Datei *find.c* enthalten und der Code zur Dokumentenrepräsentation in einer Datei *document.c* Wie kann jetzt *find.c* auf Datentypen und Funktionen in *document.c* zugreifen?

C stellt dafür die nützliche Unterscheidung zwischen *deklarieren* und *definieren* zur Verfügung. Eine Variable oder Funktion entsteht durch ihre *Definition*. Variablen wird dabei Speicher zugeteilt und bei Funktionen wird der Funktionskörper kompiliert. Funktionen können nur auf dem äussersten Sichtbarkeitslevel einer Datei definiert werden; das Schachteln von Funktionen ist in C-Standard nicht vorgesehen.

Eine Funktionsdefinition kann dabei z.B. so aussehen:

```
/* Get LINENO'th line of document DOC, NULL if no such line. */
const char* getline(const document* doc, int lineno) {
  if (lineno > 0 && lineno < doc->nlines)
    return doc->lines[lineno];
  else
    return NULL;
}
```

Eine Variablendefinition könnte z.B. so aussehen:

```
document* current_document;
```

Definitionen dürfen nur an genau einem Ort erscheinen; das ist in unserem Fall die Datei *document.c*.

Um einem anderen programmteil, der in einer anderen Datei steht, nun Zugriff auf die Variablen, Typen und Funktionen zu ermöglichen, die in einer anderen Datei definiert sind, werden sie in der verwendenden Datei *deklariert*. Durch die Deklaration wird dreist behauptet, dass es in einer anderen Datei die zu dieser Deklaration passende Definition gibt. Die zu *document.c* gehörenden Deklarationen werden üblicherweise in einer Datei "*document.h*" zusammengefasst. Hier bedeutet das Schlüsselwort `extern`, dass es sich um Deklarationen handelt.

```
/* document.h */
typedef struct { /* Data structure for documents */
  ...            /* This syntax should be known from C++ */
} document;
```

```
/* The curent document. */
extern document* current_document;

/* Get LINENO'th line of document DOC, NULL if no such line. */
extern const char* getline(const document* doc, int lineno);
```

Diese Deklarationen werden dann in *find.c* mit der `#include`-Präprozessoranweisung bekannt gemacht

```
/* find.c */
#include "document.h"

/* From here on, data types and functions defined
 * in "document.h" are known in find.c and can be
 * used. */
int findstring(const char* string_to_find,
               const document* doc) {
  ...
}
```

Beide Dateien können nun vom C-Compiler übersetzt werden. Im Gegensatz zu unserem ersten Beispiel *hello.c* soll hier aber aus der Übersetzung zunächst noch kein ausführbares Programm hervorgehen. Dies macht man dem C-Compiler mit der Option '`-c`' deutlich:

```
gcc -O -Wall -pedantic -c document.c -o document.o
gcc -O -Wall -pedantic -c find.c -o find.o
```

Gehen wir weiter davon aus, dass das Hauptprogramm in einer Datei *editor.c* zu finden ist, können wir das fertige ausführbare Programm durch zwei weitere Schritte erzeugen:

```
gcc -O -Wall -pedantic -c editor.c -o editor.o
gcc -o editor editor.o document.o find.o
```

Der letzte Schritt sorgt dann dafür, dass die verwendeten Deklarationen (z.B. von `findstring` in *find.c*) zu den passenden Definitionen zugeordnet werden. Das nennt man "binden" oder auf Denglisch auch "linken" und das Programm dazu den "Linker".

5 Referenz: Unterschiede zu Java oder C++

5.1 Klassen für Arme mit struct

C ist keine objektorientierte Sprache. Trotzdem sind die Prinzipien der objektorientierung natürlich nicht erst mit C++, Java, Ruby, Python oder anderen OO-Sprachen bekannt, und C bietet einiges an, das man als rudimentäre Basisunterstürzung für Objektorientierung begreifen könnte. Ein Buchautor, der uns leider nicht mehr geläufig ist, hat mal sinngemäss gesagt, dass man auch in Assembler objektorientiert programmieren kann, es sei halt bloss mehr Arbeit. Recht hat der Mann.

5.1.1 Grundlagen

Werte können in einem Objekt mit dem Schlüsselwort **struct** zusammengefasst werden:

```
struct quotient_and_remainder_t {
  unsigned int quotient;
  unsigned int remainder;
};
```

Sie können dann ein Objekt dieses Typs so erzeugen:

```
struct quotient_and_remainder_t a = {
    .quotient = 3,
    .remainder = 4,
  };
```

Ja, da ist tatsächlich ein Komma vor der abschliessenden geschweiften Klammer. Das geht sowohl bei **struct**- als auch bei Array-Initialisierungen. Der Grund ist wohl, dass es dann einfacher ist, syntaktisch korrekten C-Code maschinell zu erzeugen. (Sonst muss man aufpassen, dass man nach dem letzten Wert kein Komma mehr ausgibt.)

Ein **struct** ist vergleichbar mit einer Klasse in C++, bei der alle Member die Sichtbarkeit **public** haben. In C gibt es weder **private** noch **protected**.

Es ist guter Stil, dafür zu sorgen, dass Funktionen nicht mit falschen Argumenten gefüttert werden können. Wenn Sie also eine Funktion haben, die ein Objekt vom Typ **quotient_and_remainder_t** ausgibt, nenne Sie diese Funktion nicht einfach *print*, sondern vielleicht *qr_print* und geben Sie allen Funktionen, die sich mit diesem Datentyp befassen, dasselbe Präfix '*qr_*'.

5.1.2 Virtuelle Funktionen mit struct

Man mag es kaum glauben, aber man kann tatsächlich so etwas wie virtuelle Funktionen in C schreiben. Virtuelle Funktionen sind Funktionen, die in abgeleiteten Klassen anders implementiert werden können als in der Basisklasse. In C++ gibt es dafür die Syntax:

```
class Shape {
public:
  virtual draw() const;
};
```

Wird diese Klasse dann abgeleitet, z.B, in eine Klasse **Triangle**, kann die abgeleitete Klasse ihre eigene Version von *draw* implementieren.

In C kann man so etwas auch machen, und das geht mit Zeigern auf Funktionen:

```
struct shape {
  void (*draw)(const struct shape*);
}
```

Jedem Objekt vom Typ **shape** hängt nun ein Zeiger auf die entsprechende *draw*-Funktion an. Das kann man nun so nutzen (zu den syntaktischen Feinheiten siehe Abschnitt 5.4.1).

```
void draw_basic_shape(const struct shape* s) { /* ... */ }

struct shape* init_basic_shape() {
  struct shape* ret = ...;
  ret->draw = draw_basic_shape;
  return ret;
}

void draw_triangle(const struct shape* s) { /* ... */ }

struct shape* init_triangle() {
  struct shape* ret = ...;
  ret->draw = draw_triangle;
  return ret;
}

int main() {
  struct shape* s = ...;

  s->draw();
  ...
}
```

5.2 Vererbung für Arme mit union und Bitfelder

Nehmen Sie an, Sie haben ein Netzwerkprotokoll, das nach dem Typ/Wert-Schema funktioniert: Zunächst kommt eine Zahl, die Ihnen Sagt, was für ein Datentype danach folgen wird, dann kommt der Datentyp selbst. In unserem Beispiel sei die Typkodierung so vorgenommen:

Erstes Byte ist…	Es folgt eine…
0	Kurze Ganzzahl (2 Bytes)
1	Lange Ganzzahl (4 Bytes)
2	Kurze Fliesskommazahl (4 Bytes)
3	Lange Fliesskommazahl (8 Bytes)

Sie brauchen jetzt eine Datenstruktur, in der Sie alle diese Datentypen vereinen können. In Java oder C++ würden Sie sowas mit Vererbung lösen. In C haben Sie dafür nur eine Krücke, die sich an dem aus Pascal bekannten Konzept des variant records orientiert:

```
#include <stdint.h>

typedef enum { short_int, long_int, short_float, long_float } var;
typedef struct {
  var variant;
  union {
    int16_t short_i;
    int32_t long_i;
    float short_f;  // Assumes 32-bit floats (IEEE 754)
    double long_f;  // Assumes 64-bit doubles (IEEE 754)
  } value;
} packet_t;
```

Hier wurde der Standard-Header *<stdint.h>* ausgewählt, um Zugriff auf Integer-Typen fester Grösse zu bekommen.

In einem Wert vom Typ **union** können Sie alle Komponenten speichern. Im Gegensatz zu einem entsprechenden **struct**-Wert können Sie jedoch nicht alle Komponenten *gleichzeitig* dort ablegen, sondern immer nur einen. Wenn Sie ein **int16_t** dort speichern, dürfen Sie auch nur einen **int16_t** wieder herauslesen. Um anzuzeigen, was Sie lesen dürfen, dient die Angabe der verwendeten Variante in *variant*.

Das würden Sie typischerweise so verwenden:

```
packet_t packet;
/* Initialise packet */

switch (packet.variant) {
case short_int: do_short_i(packet.value.short_i); break;
case long_int: do_long_i(packet.value.long_i); break;
case short_float: do_something_short_f(packet.value.short_f); break;
case long_float: do_long_f(packet.value.long_f); break;
}
```

Beachten Sie hier die Abwesenheit eines **default**-Zweiges in der **case**-Anweisung. Das ist Absicht. Sollten neue Werte zum **enum** *var* hinzukommen, wird einen der Compiler dann bei den richtigen Einstellungen auf das Fehlen einer Behandlung hinweisen. Mit anderen Worten erhält man die Fehlermeldung schon beim Übersetzen und nicht erst beim Ausführen.

Eine andere Anwendung (manche sagen auch: Missbrauch) von **union** ist es, sich die aufwändige Bearbeitung von Werten mit Bitfeldern zu vereinfachen. Nehmen wir als Beispiel das Format von Gleitkommazahlen nach IEEE 754. Dort ist ein Wert mit einfacher Genauigkeit (single-precision floating-point) so aufgebaut:

Bit No.	Länge	Bedeutung
31	1	Vorzeichen
30–23	8	Exponent
22–0	23	Bruchteil (fraction)

Hat man nun einen 32-bit **float**, kann man selbstverständlich mittels bitweiser Operationen die Komponenten extrahieren und verändern; einfacher wäre es jedoch, wenn man auf die Bitfelder wie auf **struct**-Komponenten zugreifen könnte. Und das geht auch, *vorausgesetzt, man weiss, wie der Compiler genau arbeitet*. Hier handelt es sich nicht mehr um machinenunabhängigen Code!

```
#include <stdio.h>

typedef union {
  float float_val;
  struct {
    unsigned int f : 23; // 23 bits fraction
    unsigned int e : 8;  // eight bits exponent
    unsigned int s : 1;  // one bit sign
  } ieee_val;
} float_components;

float_components f;
```

```
f.float_val = 3.1415926536F;
// prints "s = 0; e = 128; f = 4788187" on intel
printf("s = %u; e = %u; f = %u\n",
       f.ieee_val.s, f.ieee_val.e, f.ieee_val.f);
```

5.3 Typ-Synonyme: typedef

Mit dem Schlüsselwort **typedef** gibt man einem Datentypen einen neuen Namen.

```
struct my_int_list_element_t {
  struct my_int_list_element_t* next;
  int key;
};

typedef struct my_int_list_element_t my_int_list_element;
```

Jetzt können Sie Variablen dieses Typs auf drei Arten definieren:

```
my_int_list_element kind_1;          // First way, shortest way
struct my_int_list_element_t kind_2; // Slightly longer way
struct {
  struct my_int_list_element_t* next;
  int key;
} kind_3;                           // Most surprising way
```

Das überraschende ist wohl die Art Nr. 3: Durch genaue Wiederholung der **struct**-Defintion wird kein neuer Typ erzeugt. Genausowenig wird durch **typedef** ein neuer Typ erzeugt. Es wird lediglich einem bereits bekannten Typ ein neuer Name gegeben.

5.4 Pointer und Arrays

In Java gibt es Objektreferenzen. Das ist ein bestimmter Datentyp, dessen Instanzen auf Objekttypen verweisen können. Die Klasse, die ein Element einer einfach verkettete Liste von **int**-Werten beschreibt, können Sie deshalb in Java so schreiben:

```
class IntLinkedListItem {
  IntLinkedListItem next;
  int key;
}
```

Hier enthält eine Instanz von *IntLinkedListItem* eine Referenz auf das nächste Listenelement.

5.4.1 Pointer

Wenn Sie das in C wie folgt versuchen:

```
typedef struct int_linked_list_item_t {
  struct int_linked_list_item_t next;  // error
  int key;
} IntLinkedListItem;
```

erhalten Sie eine Fehlermeldung, weil die Bedeutung dessen, was Sie da geschrieben haben bedeutet, dass Sie versuchen, eine Instanz von (und nicht einen *Verweis* auf eine Instanz von) **struct** *int_linked_list_item_t* in einer anderen Instanz desselben Typs unterzubringen. Das können Sie in der Realen Welt machen, z.B. indem Sie eine Tasche in eine andere Tasche stecken, aber in C geht das nicht.

In C müssen Sie Verweise selber verwalten. Das sähe dann so aus:

```
typedef struct int_linked_list_item_t {
  struct int_linked_list_item_t *next;
  int key;
} IntLinkedListItem;
```

Können Sie den Unterschied sehen? Es ist das Zeichen '`*`', das bedeutet, dass hier nicht der **struct** selbst in einem anderen untergebracht ist, sondern nur ein *Verweis* auf einen **struct**. Verweise heissen im C-Sprachgebrauch Zeiger oder *pointer*.

Pointer sind Repräsentationen der Adresse von Objekten, auf die sie zeigen. Es gibt eine bestimmte Pointer-Konstante, die anzeigt, dass der Pointer auf kein Objekt zeigt. Diese Konstante heisst *Nullzeiger* oder *null pointer* wird mit '0' bezeichnet. Um sie von der Ganzzahlkonstante '0' zu unterscheiden, die ja genau gleich geschrieben wird, gibt es den symbolischen Namen `NULL`, der in <*stdlib.h*>, <*stdio.h*> und einigen anderen Standard-Headern definiert ist.

Wenn Sie ein Objekt vom Typ *T* und einen Zeiger vom Typ *T** haben, können Sie mit dem address-of-Operator '`&`' die Adresse des Objekts dem Zeiger zuweisen und mit dem Dereferenzierungsoperator '`*`' auf das Objekt, auf das der Zeiger zeigt, zugreifen:

```
int_linked_list_item front;
/* Fill list here. */
int_linked_list_item* p = &front; // Assign address to pointer

while (p != NULL) {          // While p points to valid object
  printf("%d\n", (*p).key);  // Print list item
  p = (*p).next;             // Go to next element
}
```

Der Ausdruck `(*p).key` bedeutet, dass zunächst mittels `(*p)` auf das Objekt, auf das *p* zeigt, zugegriffen wird und dort dann mittels `.key` auf die Komponente *key*. Der Zugriff auf **struct**-Objekte mittels Zeigern geschieht in C so häufig, dass es eine besondere Notation dafür gibt: die Schreibweise `p->key` ist identisch mit `(*p).key`:

```
int_linked_list_item front;
/* Fill list here. */
int_linked_list_item* p = &front; // Assign address to pointer

while (p != NULL) {         // While p points to valid object
  printf("%d\n", p->key);  // Print list item, easy way
  p = p->next;             // Go to next element, easy way
}
```

Es ist immer ein Fehler, einen Nullzeiger zu dereferenzieren. Mit anderen Worten., wenn Sie einen Zeiger *p* haben und der ist 0, dann ist der Ausdruck $*p$ fehlerhaft und führt zu undefiniertem Verhalten (das wird in Linux ein Programmabsturz sein).

5.4.2 Arrays und Zeigerarithmetik

Wie in Java und C++ sind Felder, auch *arrays* genannt, zusammenhängende Speicherblöcke, die Objekte desselben Typs enthalten. Felder werden aber anders definiert und verwendet. In Java schreibt man z.B.:

```
SomeClass[] classArray = new SomeClass[100];
```

Das Objekt *classArray* liegt dabei in demselben Speicherbereich, in dem alle dynamisch erzeugten Objekte liegen (dem *Heap*). Das ist auch die *einzige* Art, ein Java Array zu erzeugen. In C können Arrays auch anders erzeugt werden, nämlich z.B. so:

```
some_class class_array[100];
```

Je nachdem, wo dieser Ausdruck nun steht, kann *class_array* auf dem Stack liegen (wo auch andere lokale Variablen abgelegt werden), oder im sogenannten Daten-Segment, in dem sich globale Variablen befinden. Globale Variablen gibt es in Java nicht (obwohl man denselben Effekt natürlich erreichen kann).

Der Effekt dieses Ausrucks ist es, dass ein Array von einhundert *some_class*-Objekten erzeugt (aber im Unterschied zu C++ *nicht* initialisiert) wird. Der Name *class_array* ist dann ein Synonym für die Adresse des ersten Objekts:

```
class_array == &class_array[0];  // This is always  true
```

Da das so ist, kann man einem Zeiger einen Array-Namen zuweisen:

```
some_class* p = class_array;  // p points to first element
```

Wäre es nicht praktisch, wenn man auf syntaktisch einfache Art auf ein beliebiges Array-Element zugreifen könntenn, wenn man einen Zeiger auf das erste Element hat? Es wäre und man kann. Hat man einen Zeiger p auf Element i eines Arrays, so bezeichnet der Ausdruck $p + k$ einen Zeiger auf das Element $i + k$ des Arrays, vorausgesetzt, dass der Index $i + k$ innerhalb des Arrays liegt oder maximal ein Element ausserhalb. (Warum es legal und auch angenehm ist, einen Zeiger auf ein nichtexistentes Arrayelement zu haben, zeigen wir Ihnen später. Natürlich darf man so einen Zeiger nicht dereferenzieren, er dient nur zum Vergleich.) Der Ausdruck *class_array*`[`i`]` ist so *definiert*, dass er äquivalent ist zum Ausdruck $*$`(`*class_array* $+$ i`)` Mit anderen Worten,

```
some_class* p = class_array;  // p points to first element
int i = ...;                  // some legal index
p + i == &class_array[i];     // This is always true
```

5.4.3 Array-Initialisierung

Was glauben Sie, was passiert, wenn Sie folgenden Code ausführen?

```
char* s = "abcd";
s[1] = 'x';        // error
printf("%s\n", s);
```

Der String `"abcd"` ist ja offenbar ein **char**-Array mit 5 Elementen, also sollte ja wohl ‘`axcd`’ ausgegeben werden, richtig? Leider erhalten Sie auf einem normalen Linux-System bei Ausführung dieses Codes gar keine Ausgabe, sondern eine System-Fehlermeldung (‘Segmentation Fault’). Was ist da passiert?

Klären wir zunächst auf, was die erste Zeile macht. Dort wird einem Zeiger eine Zeichenketten-Konstante zugewiesen, das heisst, dass *s* auf das erste Zeichen der Zeichenkette. Das Problem ist nun, dass diese Konstante möglicherweise nicht in einem schreibbaren Bereich liegt, obwohl die den Typ 'Array von **char**' hat und der Compiler daher selbst bei Verwendung von '`-Wall`' keine Warnung ausgibt. Um das zu verhindern, kann man gcc anweisen, mit '`-Wwrite-strings`' Zeichenketten-Konstanten den Typ 'Array von **const char**' zu geben. Dann wird auch eine Warnung erzeugt. (Daran sieht man, dass '`-Wall`' durchaus *nicht* alle Warnungen einschaltet.)

Was aber nun, wenn ich ein echtes Array erzeugen möchte, eins, in das ich auch nach Belieben schreiben kann? In dem Fall nimmt man diese Initialisierung:

```
char s[] = "abcd";  // Instead of char* s = ...;
s[1] = 'x';         // ok
printf("%s\n", s);
```

5.4.4 Array-Iteration

Wie wir oben erwähnt haben, ist ein Zeiger, der auf ein (nicht existentes) Element so gerade eben ausserhalb eines Arrays verweist, legal. Der Grund ist, dass man mit Zeigern leicht über Arrays iterieren kann, wenn so eine Konstruktion möglich ist. Normalerweise iterieren Sie so über ein Array:

```
typedef struct { int count; } counter;
counter counters[100];

for (int i = 0; i < 100; i++)
  counters[i].count = 0;
```

Der letzte Array-Eintrag hat dabei den Index 99. Manchmal ist es bequemer, mittels eines Zeigers zu iterieren, und dank der Regel mit dem Element ausserhalb des Arrays funktioniert das auch:

```
typedef struct { int count; } counter;
counter counters[100];
counter* q = counters + 100; // One element beyond array

for (counter* p = counters; p < q; p++)
  p->count = 0;
```

Gäbe es die Regel nicht, d.h. müsste ein Zeiger immer auf ein Element innerhalb des Arrays verweisen, wäre das ungleich komplizierter:

```
typedef struct { int count; } counter;
counter counters[100];
counter* q = counters + 100 - 1; // Last element in array
counter* p = counters;

while (p <= q) {
  p->count = 0;
  if (p == q)
    break;
  p++;
}
```

5.4.5 Zeiger und Zeichenketten

In C sind Zeichenketten (String) kein elementarer Datentyp wie z.B. in Java oder (eingeschränkt) in C++. Stattdessen wird ein String repräsentiert als ein Array von **char**, dessen letztes Element ein Nullzeichen ist. Dieses terminierende Zeichen wird `'\0'` geschrieben. Mit unserem frisch erworbenen Wissen über Arrays und Pointer können wir gleich diesen Code verstehen:

```
char* my_name = "Stephan"; // Needs 8 characters
unsigned int i = 0;

for (char* p = my_name; *p != '\0'; p++)
  i++;

// i now has the value 7.
```

5.4.6 Zeiger (C) und Referenzen (C++)

In C gibt es keine Referenzen so wie in C++. Manche Leute sagen, das sei etwas Gutes.

5.4.7 Eingeschränkte Verwendung: restrict

Ein grosses Problem bei der Optimisierung von Code ist das sogenannte Pointer-Aliasing. Nehmen wir als Beispiel diese C-Funktion (Beispiel aus Wikipedia[1]):

```
void updatePtrs(size_t *ptrA, size_t *ptrB, size_t *val) {
  *ptrA += *val;
  *ptrB += *val;
}
```

Der Compiler kann hier nicht ausschliessen, dass *ptrA*, *ptrA* und *val*, alle auf denselben **size_t** zeigen. Deshalb darf der Compiler nicht davon ausgehen, dass sich **val* nach der ersten Zeile nicht geändert hat. Der Compiler muss also den Wert von **val* nach der ersten Zeile erneut bestimmen. Deklariert man diese Zeiger als **restrict**, darf der Compiler annehmen, dass all diese Zeiger auf verschiedene Speicherstellen verweisen:

```
void updatePtrs(size_t *restrict ptrA,
                size_t *restrict ptrB,
                size_t *restrict val) {
  *ptrA += *val;
  *ptrB += *val;
}
```

Die genaue formale Definition ist sehr kompliziert, läuft aber im wesentlichen darauf hinaus, das alle Zugriffe auf das hinter diesem Zeiger stehenden Objekt direkt oder indirekt durch diesen Zeiger erfolgen. Mit anderen Worten, ist ein Pointer Funktionsparameter und wird dieser als **restrict** deklariert, verspricht der Aufrufer, dass dieser Pointer zum Aufrufzeitpunkt die einzige Möglichkeit ist, an das entsprechende Objekt zu gelangen.

Beispielsweise existieren in C zwei Funktionen, um Speicherbereiche zu kopieren: *memcpy* und *memmove*. Dabei nimmt *memcpy* an, dass sich Quelle und Ziel nicht

[1] `http://en.wikipedia.org/wiki/Restrict`

überlappen. Bei *memmove* darf das passieren. Konsequenterweise sind diese beiden Funktionen verschieden deklariert:

```
extern void* memcpy(void* restrict s1,
                    const void* restrict s2,
                    size_t n);
extern void* memmove(void* s1, const void* s2, size_t n);
```

5.5 Konstanten, Zeiger auf Konstanten: const

Ab C99 können konstante Werte werden verwendet wie in jeder anderen Programiersprache auch. In C++ heisst das Schlüsselwort ebenfalls **const** (was nicht überraschen sollte, ist doch ein gültiges C-Programm immer auch ein gültiges C++-Programm) und in Java **final**.

```
const int n_elements = 100;
some_class class_array[n_elements];
```

Hat man C99 nicht zur Verfügung, muss man entweder den Umweg über enumerations gehen:

```
enum { n_elements = 100 };
some_class class_array[n_elements];
```

oder über den Präprozessor (siehe Abschnitt 5.12.2).

Viel häufiger ist die Verwendung von Zeigern auf Konstanten. Damit zeigt man dem Compiler an, dass man das Objekt, auf das der Zeiger zeigt, nicht verändern wird. Der Compiler kann daraufhin bestimmte Optimierungen vornehmen, die ohne das Schlüsselwort **const** nicht möglich wären.

Wenn Sie sich die Deklaration der Funktion *strlen* ansehen, stellen Sie fest, dass diese ein Argument vom Typ **const char*** erhält. Damit teilt die Funktion dem Übersetzer mit, dass sie gedenkt, den Parameter nur anzuschauen, aber nicht zu verändern. Mit unserem Wissen über Zeichenketten können wir die Funktion *strlen* nun selbst implementieren:

```
size_t strlen(const char* s) {
  const char* t;

  for (t = s; *t != '\0'; t++)
    ; // empty

  return t - s;
}
```

Das funktioniert aber nicht immer richtig, weil der Typ von `t - s` leider nicht **size_t** sondern **ptrdiff_t** ist. Das wäre nicht weiter schlimm, aber leider ist **size_t** vorzeichenlos, **ptrdiff_t** aber nicht. Mit anderen Worten, ist die Zeichenkette so lang, dass die Differenz $t - s$ zwar noch in einen Wert vom Typ **size_t** passt, aber nicht mehr in einen Wert vom Typ **ptrdiff_t**, gibt die Funktion den falschen Wert zurück. Deshalb lösen wir das jetzt so:

```
size_t strlen(const char* s) {
  const char* t;
  size_t ret = 0;

  for (t = s; *t != '\0'; t++)
    ret++;

  return ret;
}
```

C ist eine äussert kompakte Sprache. Denselben Effekt erhalten Sie mit diesem Code, der in C schon fast idiomatisch verwendet wird:

```
size_t strlen(const char* s) {
  size_t ret = 0;

  while (*s++)
    ret++;

  return ret;
}
```

5.6 Speicher dynamisch anfordern

Bislang haben wir in diesem Tutorial nur globale oder lokale Variablen verwendet. Nun wollen wir uns mit dynamischem Speicher befassen. In C++ und Java wird dynamischer Speicher mit dem Schlüsselwort **new** angefordert (wenn auch mit leicht anderer Syntax):

```
string* string_array = new string[100]; // C++
String[] stringArray = new String[100]; // Java
```

In C geht es etwas hemdsärmeliger zur Sache; dort finden wir nur die Funktion *malloc* und verwandte Funktionen. Die Funktion *malloc* bekommt als Argument die Grösse des benötigten Speicherbereichs und liefert im Erfolgsfall einen Zeiger auf einen Speicherbereich, der ein Objekt der angeforderten Grösse aufnehmen kann. Dieser Zeiger hat den Typ **void***; diesen wird man also durch eine Typumwandlung anpassen müssen. Im Fehlerfall wird ein Nullzeiger zurückgegeben.

Wie ermittelt man nun den Platzbedarf, den man an *malloc* übermitteln muss? Ganz einfach. Im folgenden bezeichnet *T* irgendeinen Datentypen.

- Möchte man ein einzelnes Element vom Typ *T*, ist der Platzbedarf gegeben durch **sizeof** *T*.
- Möchte man ein Array mit *n* Elementen vom Typ *T*, ist der Platzbedarf gegeben durch *n*(**sizeof** *T*).

Wenn man dann noch weiss, dass **sizeof char** als 1 definiert ist, kann man z.B. Strings so allozieren:

```
char* make_string_filled_with_a(size_t length) {
  char* ret = (char*) malloc(length + 1);
  if (ret != 0) {
```

```
    for (size_t i = 0; i < length; i++)
      ret[i] = 'a';
    ret[length] = '\0';
  }
  return ret;
}
```

Möchten Sie **struct**-Werte anfordern, geht das ganz ähnlich:

```
int_list_element* make_int_list_element() {
  int_list_element* ret
    = (int_list_element*) malloc(sizeof(int_list_element));
  if (ret != 0) {
    ret->next = 0;
    ret->key = 0;
  }
  return ret;
}
```

Beim GCC muss man allerdings den Typcast vor **malloc** nicht durchführen, da der Compiler am *lvalue* (hier: `ret`) erkennt, welchen Typ dieser besitzt, sodass der Cast von `void *` implizit geschieht. Bei Strukturen kann man auch bei **sizeof** den Pointertyp dereferenzieren, da der Compiler sich hierbei lediglich die zugrunde liegende Datenstruktur ansieht, z.B.:

```
int_list_element* make_int_list_element() {
  int_list_element* ret = malloc(sizeof(*ret));
  if (ret != 0) {
    ret->next = 0;
    ret->key = 0;
  }
  return ret;
}
```

Falls sich nun der Typ von `ret` ändert, so muss man nicht doppelt den Typen ändern, d.h. man kann dann auf die Änderung im `sizeof` verzichten.

Eine solche Dereferenz funktioniert sogar mit Komponenten der Struktur, d.h. man könnte in begründeten Fällen durchaus soetwas schreiben:

```
sizeof(((int_list_element *)(0))->key)
```

Einmal angeforderten Speicher müssen Sie wieder freigeben; es gibt in C keine garbage collection wie in Java. Die Funktion dazu heisst *free* und ihr übergeben Sie einen Pointer, den Sie zuvor von **malloc** erhalten haben, oder *NULL*. In letzterem Fall passiert gar nichts.:

```
void free_int_list_element(int_list_element_t* p) {
  free(p);
}
```

Noch eine kleine Kopfnuss zum Abschluss. Was gibt folgendes Programm aus?

```
int main() {
  char s[] = "abcd";
  char* t = "abcd";

  printf("s = %s, t = %s\n", s, t);
  printf("sizeof s = %zd, sizeof t = %z\n", sizeof s, sizeof t);
}
```

Wenn doch die Werte von *s* und *t* gleich sind, wie kann da **sizeof** verschiedene Werte für *s* und *t* liefern? Und wieso ist die Syntax '**sizeof** *s*' legal, gehören um das Argument von **sizeof** nicht Klammern wie bei jedem Funktionsaufruf auch?

5.7 Wenn sich Werte auf magische Art ändern: volatile

Schauen Sie sich folgenden maschinennahen Code an:

```
char* device_register = (char*) 0xfffffabc;

char read_value() {
  return *device_register;
}
```

Die Idee ist, dass *device_register* die Adresse eines Registers des Controllers irgendeines I/O-Geräts ist und mit *read_value* dieser Registerwert ausgelesen wird.

Wenn Sie solchen Code selbst implementieren, können Sie Ihr blaues Wunder erleben, nämlich dann, wenn Sie die Optimierung einschalten. Wird nämlich *device_register* sonst nirgendwo verwendet, und insbesondere niemals geschrieben, könnte der Compiler zu dem Schluss kommen, dass sich der einmal aus *device_register* gelesene Wert nie mehr ändern kann: Der Compiler weiss ja nicht, dass hinter der Adresse `0xfffffabc` ein Ein-Ausgabe-Gerät steckt.

Um dem Compiler zu verdeutlichen, dass sich der Wert einer Variablen auch durch Einwirkung von ausserhalb des Programms ändern kann, definieren Sie diese Variable als **volatile**:

```
volatile char* device_register = (char*) 0xfffffabc;

char read_value() {
  return *device_register;
}
```

Werden Sie das Register nur lesen und nie schreiben, können Sie es sogar als **volatile const char*** deklarieren.

5.8 Signale

Wenn Sie ein in der Shell ablaufendes Programm mit Eingabe von Control-C abbrechen, erhält das ablaufende Programm ein *Signal*. Ein Signal ist eine Nachricht, die jederzeit eintreffen kann, auch ohne, dass man darauf vorbereitet ist oder das Signal angefordert hat. Das ist vergleichbar mit einem Telefon, das jederzeit klingeln kann. In Unix wird ein Signal entweder *ignoriert* oder *behandelt*. Wird das Signal ignoriert, geschieht beim Eintreffen des Signals nichts; wird es behandelt, wird der Code ausgeführt, der zum Behandeln des Signals dient. Dieser Code heisst *signal handler*.

Ein signal handler kann die Beendigung des Programms veranlassen. Wenn aber der handler normal zurückkehrt, und wenn das Signal anzeigt, dass etwas mit dem ausführenden Programm nicht stimmt (es wurde versucht, eine Instruktion auszuführen, die nicht im Befehlssatz des Prozessors ist; es gab eine Floating-Point exception; es wurde versucht, auf nichtexistenten Speicher zuzugreifen etc), ist undefiniert, was danach geschieht. Sonst wird das Programm an der Stelle fortgesetzt, an dem es sich befand, als das Signal eintraf.

5.8.1 Signale laut C-Standard

Folgendes Programm fängt ein Control-C ab und verwendet es, um das Programm geordnet zu terminieren:

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

volatile int do_quit = 0;  // Not quite right, see Section 5.8.2

void handler(int sig) {
  if (sig == SIGINT)
    do_quit = 1;
}

int main() {
  if (signal(SIGINT, handler) == SIG_ERR) {
    perror("signal");
    exit(1);
  }

  unsigned int i = 0;
  while (!do_quit) {
    i = i + 1;
  }
  printf("Quit after i = %u\n", i);
  return 0;
}
```

Wenn Sie dieses Programm übersetzen und ablaufen lassen, gerät es bald in die **while**-Schleife. Ein Control-C aus der Shell sendet an das Programm das Signal `SIGINT`. Dieses wird abgefangen und führt dazu, dass der Kontrollfluss aus der Schleife heraus in die Funktion *handler* übergeht, wo die Variable *do_quit* auf 1 gesetzt wird. Das Programm wird dann in der Schleife fortgesetzt, wo es dann bei der nächten Iteration zum Abbruch kommt.

Die Funktion *signal* hat als ersten Parameter den Wert des Signals, das Sie behandeln wollen. Unser Wert hier ist `SIGINT`, was laut Dokumentation (`man 7 signal`) für “interrupt from keyboard” steht. Das zweite Argument ist die Adresse einer Funktion, die keinen Rückgabewert hat und als einziges Argument eine Signalnummer bekommt. Diese Funktion wird dann aufgerufen, sobald dieses Signal eintrifft. (Weil man einen handler auch zur Bearbeitung mehrerer Signale verwenden kann, mussman dem handler die Nummer des eingegangenen Signals mitgeben.)

Wenn Sie Abschnitt 5.7 gelesen haben, wissen Sie bereits, warum die Variable *do_quit* als **volatile** deklariert werden muss. Wenn nicht, lesen Sie ihne jetzt und über-

setzen Sie das Programm ohne **volatile** aber mit `-O2`. Jetzt führ ein Control-C (auf meinem System, ein 64-bit AMD Linux mit Linux 2.6.35) nicht mehr zum Abbruch!

Während Ihr Programm einen signal handler ausführt, ist der normale Programmfluss unterbrochen. Tritt bei der Ausführung eines handlers ein weiters Signal auf, kann es schnell zu unbeherrschbaren Zuständen führen. Deshalb gilt die Regel: *In einem signal handler soll man nur flags setzen und alles andere—Berechnungen, Aufrufe von System- oder eigenen Funktionen, etc—unterlassen!* Es gibt zwar Regeln für den Signal-sicheren Aufruf von Systemfunktionen aus signal handlern heraus, aber die sind so kompliziert, dass wir niemandem zutrauen, diese Regeln vollständig zu beherrschen und *uns* nicht zutrauen, einen vorhandenen signal handler korrekt daraufhin zu untersuchen, ob er Signal-sicher ist.

5.8.2 Signale und sig_atomic_t

Signale können an allen möglichen Zeitpunkten auftreten. Sicher ist Ihnen aus der obigen Diskussion klar geworden, dass Signale während der Bearbeitung eines C-Ausdrucks auftreten können. Was Ihnen aber möglicherweise noch nicht klar ist: Signale können auch die Ausführung einer scheinbar einfachen Zuweisung unterbrechen!

Nehmen wir als Beispiel unseren signal handler von oben, der den Wert 1 in die Variable *do_quit* speichert. Nehmen wir an, wir würden jetzt *do_quit* nicht auf 1, sondern auf `0x100000001` setzen wollen. Dazu würden wir ein **long long** benötigen, der unter aktuellen Linux-Implementationen 64 bit lang ist:

```
volatile long long do_quit = 0;

void handler(int sig) {
  if (sig == SIGINT)
    do_quit = 0x100000001LL;
}
```

Übersetzt man diesen Code auf einem 32-bit Intel Linux, erhält man:

```
handler:
        pushl   %ebp
        movl    %esp, %ebp
        movl    $1, do_quit
        popl    %ebp
        movl    $1, do_quit+4
        ret
```

Mit anderen Worten, da nicht ein 64bit-Wert geschrieben werden kann, müssen zwei 32bit-Werte geschrieben werden. Zwischen diesen beiden `movl`-Instruktionen kann jetzt ein Signal eintreffen, so dass *do_quit* womöglich *weder* den Wert 0 noch den Wert `0x100000001` hat! in unserem Fall ist das nicht so wichtig; man könnte z.B. jeden von Null verschiedenen Wert als Abbruchkriterium nehmen, es kann aber Fälle geben, wo der genaue Wert des Flags wichtig ist.

Für solche Fälle sieht der C-Standard den Datentyp **sig_atomic_t** vor. Zuweisungen an Variablen mit diesem Typ sind garantiert atomar und können nicht unterbrochen werden. Wir hätten also unseren Original-signal handler von oben richtig so schreiben müssen:

```
volatile sig_atomic_t do_quit = 0;
```

```
void handler(int sig) {
  if (sig == SIGINT)
    do_quit = 1;
}
```

Der Wertebereich von **sig_atomic_t** umfasst dabei mindestens die Ganzzahlen von -127 bis $+127$, die genauen Minimal- und Maximalwerte stehen in <*stdint.h*> (und nicht in <*limits.h*>, wie man annehmen könnte).

Beachten Sie, dass zwar die Zuweisung mit Literalen garantiert atomar ist, nicht aber eine Berechnung. Wenn Sie nicht `do_quit = 1`, sondern `do_quit++` schreiben, verlieren Sie die Garantie der atomaren Änderung.

5.8.3 Signale unter Linux

Unter Linux gibt es für alle bekannten Signale eine von vier *Default-Behandlungen*. Diese werden vorgenommen, wenn nicht mit *signal* eine explizite Behandlung vorgenommen wird. Diese Default-Behandlungen sind nicht im C-Sprachstandard verankert (aber natürlich mit diesem kompatibel). Andere Betriebssysteme können andere Default-Behandlungen vorsehen. Unter Linux sind das:

Terminate (Term). Das ablaufende Programm wird beendet.

Core dump (Core). Das ablaufende Programm wird beendet und ein Speicherabzug (core dump) wird geschrieben.

Ignore (Ign). Das Signal wird ignoriert.

Cont. Das Programm war vorher gestoppt (z.B. mit Control-Z) und wird nun fortgesetzt (z.B. mit `fg`)

Stop. Das ablaufende Programm wird angehalten.

Welche Signale welche Art der Default-Behandlung nach sich ziehen, kann man in den Manpages oder den info pages über Signale nachlesen. Sie können für ein Signal dessen Default-Behandlung wiederherstellen, indem Sie beim Aufruf der *signal*-Funktion als zweiten Parameter `SIG_IGN` übergeben. Wollen Sie ein Signal ignorieren, geben Sie `SIG_IGN` an.

Das Signal `SIGKILL` kann man nicht ignorieren oder anderweitig behandeln lassen. Es führt immer zur Beendigung des Programms. Dieses Signal trägt die Nummer 9, und wenn Sie also aus der Shell heraus ein ablaufendes Programm mit `kill -9` "abschiessen", tun Sie nichts anders, als diesem Programm ein Signal zu senden, das es nicht ignorieren oder sonstwie abfangen kann. Es ist daher oft das letzte Mittel, ein aus dem Ruder gelaufenes Programm noch zu beenden.

5.9 Atomarer Zugriff

Schauen Sie sich folgenden Code aus einer idealisierten Systemanwendung an. Die Idee ist, dass die Anzahl aller insgesamt gesendeten Pakete protokolliert wird.

```
unsigned int sent = 0;

void record_sent_packets(unsigned int n_packets) {
  sent += n_packets;
}
```

#	Erster Thread	Zweiter Thread	Wert
1	`pushl %ebp`		0
2	`movl %esp, %ebp`		0
3	`movl sent, %eax`		0
4	`addl 8(%ebp), %eax`		0
5		`pushl %ebp`	0
6		`movl %esp, %ebp`	0
7		`movl sent, %eax`	0
8		`addl 8(%ebp), %eax`	0
9		`movl %eax, sent`	200
10		`popl %ebp`	200
11		`ret`	200
12	`movl %eax, sent`		100
13	`popl %ebp`		100
14	`ret`		100

Tabelle 1: Zwei Threads greifen gleichzeitig auf eine Variable zu.

Das funktioniert solange gut, wie es im Programm nur einen Kontrollfluss (Thread) gibt. Gibt es mehrere Threads, kann diese Art des Variablenzugriffs schnell falsche Ergebnisse produzieren. Dazu schauen wir uns an, was gcc mit diesem Code macht. Der zugehörige Assembler-Code für 32-bit Intel-Architekturen sieht im Kern so aus (bei nicht eingeschalteter Optimierung):

```
record_sent_packets:
        pushl   %ebp
        movl    %esp, %ebp
        movl    sent, %eax
        addl    8(%ebp), %eax
        movl    %eax, sent
        popl    %ebp
        ret
```

Die ersten zwei Instruktionen erzeugen einen neuen Aktivierungsrahmen (stack frame). Danach wird der aktuelle Wert von `packets_sent` in das Register `eax` geladen, der Wert des übergebenen Parameters dazu addiert (dieser liegt 8 bytes über dem Wert von `ebp`, daher ist dessen Adresse `8(%ebp)`) und dann zurückgeschrieben. Wenn jetzt zwei Kontrollflüsse gleichzeitig diesen Code ausführen, darf es laut C-Standard zu der in Tabelle 1 geschilderten Situation kommen. Nehmen wir an, dass der erste Thread *record_sent_packets* mit dem Parameter 100 aufruft und der zweite diese Funktion mit 200. Insgesamt sollten also 300 Pakete gesendet worden sein, aber nach Ausführung des Codes enthält die Variable *sent* bloss den Wert 100!

In Java gibt es dafür das Schlüsselwort **synchronized**, aber in C gibt es keine Sprachmittel, um gleichzeitigen Zugriff zu synchronisieren. Sie werden in diesen Fachpraktika jedoch einige Möglichkeiten kennenlernen, um diese Aufgabe im Linux-Kern zu lösen.

5.10 Parameterübergabe

In C werden Parameter grundsätzlich *by value* übergeben, C kennt keine andere Art der Parameterübergabe, wie z.B. *by reference* wie in C++. Das bedeutet, dass Sie Rückgabewerte entweder direkt als Rückgabewert der Funktion deklarieren, oder Zeigerparameter verwenden müssen.

Nehmen wir als Beispiel eine Funktion, die es erlaubt, aus zwei **unsigned**-Werten m und n in einem Zug sowohl den ganzzahligen Quotienten $\lfloor m/n \rfloor$ als auch den ganzzahligen Rest m mod n zu bestimmen. Hier haben Sie zwei Rückgabewerte. Das können Sie nun so lösen:

```
void quotient_and_remainder(unsigned* q, unsigned* r,
                            unsigned m, unsigned n) {
  *q = /* quotient */;
  *r = /* remainder */;
}
```

Dann muss der Aufrufer Ihre Schnittstelle so bedienen:

```
unsigned q, r, m = 113, n = 37;

quotient_and_remainder(&q, &r, m, n);
/* At this point, q == 3, r == 2 */
```

Typischerweise werden in einem solchen Fall *alle* Rückgabewerte über Pointer-Parameter zurückgegeben und die Funktion hat dann entweder den Rückgabetyp **void** oder **int**, falls noch Fehler auftreten können und diese Fehlerbedingung dem Aufrufer mitgeteilt werden soll (hier wäre das z.B. möglich, um den Fall $n = 0$ abzufangen).

Eine andere mögliche Lösung ist es, sich einen **struct**-Wert zu definieren, der zwei **int**-Werte enthält:

```
typedef struct _q_and_r_t {
  unsigned q;
  unsigned r;
} q_and_r_t;

q_and_r_t quotient_and_remainder(unsigned m, unsigned n) {
  q_and_r_t ret;

  ret.q = /* quotient */;
  ret.r = /* remainder */;

  return ret;
}
```

Dann muss der Aufrufer Ihre Schnittstelle so bedienen:

```
q_and_r_t qr;

qr = quotient_and_remainder(m, n);
/* At this point, qr.q == 3, qr.r == 2 */
```

Wir raten eindringlich zur ersten Fassung, weil es dem Aufrufer nicht Ihre **struct**-Definition aufdrängt.

5.11 Sichtbarkeitsregeln

5.12 Präprozessor

C-Dateien werden gar nicht direkt übersetzt. Vor der Übersetzung folgt die Bearbeitung der Datei durch einen Präprozessor. Dieser Präprozessor hat zwei Aufgaben, nämlich die Entfernung von Kommentaren, die Bearbeitung von Präprozessor-Anweisungen. Eine Präprozessor-Anweisung (in Weiteren nur noch "Anweisung" genannt) haben wir schon kennengelernt: `#include` und die anderen behandeln wir weiter unten. Das Entfernen von Kommentaren müssen wir hier nicht beschreiben und die Bearbeitung von `#include`-Answeisungen wird in Abschnitt 4.6 genauer beschrieben.

Leider ist die Arbeitsweise des Präprozessors von der Arbeitsweise des restlichen C-Compilers grundlegend verschieden und leider kann der Präprozessor so missbraucht werden, dass der eigentliche Programmtext schwer verständlich wird. Wir werden uns bemühen, Ihnen zu zeigen, wie man einigermassen sinnvoll mit dem Präprozessor umgeht.

5.12.1 Grundlegendes zu Anweisungen

Eine *Anweisung* ist eine Zeile in einem C-Programm, die mit einem Hash-Zeichen '`#`' beginnt und am Zeilenede endet. Vor dem beginnenden Hash-Zeichen dürfen noch Leerzeichen stehen. Nach dem Hash dürfen nur die folgenden Zeichenfolgen stehen (eventuell mit Leerzeichen vom Hash-Zeichen abgesetzt): `if`, `ifdef`, `ifndef` `elif`, `else`, `endif`, `include`, `define`, `undef`, `line`, `error`, oder `pragma`. Wir behandeln nun diese Anweisungen (mit Ausnahme von `include`).

5.12.2 Makrodefinition: `#define` und `#undef`

Ein *Makro* ist ein Bezeichner, der bei der Bearbeitung durch den Ersetzungstext ersetzt wird. Makros werden mit `#define` definiert und erhalten sofort nach der Definition ihre Bedeutung. Ein Beispiel:

```
#define BUFFER_SIZE 10240
char buffer[BUFFER_SIZE];
```

Der Bezeichner *BUFFER_SIZE* wird nach der Definition durch 10240 ersetzt, so dass dieser C-Text äquivalent zur Definition '`char buffer[10240];`' ist. Diese Verwendung des Präprozessors ist Standard. Ab der Version C99 bietet C jedoch **const** an, so dass man unter Umgehung des Präprozessors schreiben kann:

```
const unsigned buffer_size = 10240;
char buffer[buffer_size];
```

Wenn man C99 zur Verfügung hat, sollte man diese Alternative nehmen, weil in diesem Fall der C-Compiler Typprüfungen vornehmen kann.

Wären Makros nur auf die Ersetzung von Bezeichnern mit Konstanten beschränkt, wäre der Präprozessor nicht so schlimm. Einer der Autoren dieses Handbuchs (SN) hat jedoch mit Pascal angefangen zu programmieren und ihm war die Syntax von C mit den geschweiften Klammern suspekt. Folglich erstellte er eine Header-Datei "*pascal.h*", die unter Anderem Folgendes enthielt:

```
#define IF if (
#define THEN )
#define BEGIN {
#define END }
```

So konnte er dann folgendes in einem “Cascal”-Dialekt schreiben:

```
IF i == 2 THEN BEGIN   // if (i == 2) {
  printf("i is 2\n");  //   ...
END                    // }
```

Das ist Missbrauch des Präprozessors. Ändern Sie niemals syntaktische Elemente von C mit dem Präprozessor ändern.

Es ist verboten, ein bereits definiertes Makro so neu zu definieren, dass der Ersetzungstext anders ist als vorher. (Die genauen Regeln dazu sind kompliziert.) Wollen Sie ein Makro neu definieren, können Sie es mit `#undef` vorher löschen. Es ist dabei in Ordnung, ein nicht definiertes Makro zu löschen.

```
#define BUFFE_SIZE 10240
char buffer_1[BUFFER_SIZE];
#undef BUFFER_SIZE
#define BUFFE_SIZE 20480
char buffer_2[BUFFER_SIZE];
```

Makros können Parameter haben. Das diente ursprünglich dazu, funktionsähnliche Strukturen zu bekommen, die “inline” expandiert werden, also ohne Overhead für einen Funktionsaufruf. Zum Beispiel dieses Funktionsähnliche Makro zum Aufrunden von x auf das nächste Vielfache von y:

```
#define NEXT_MULTIPLE_OF(x, y) ((y)*(((x) + (y) - 1) / (y)))
```

Verwendet man nun dieses Makro so: ‘`NEXT_MULTIPLE_OF(2000,256)`’, so erhält der C-Compiler ‘`((256)*(((3000) + (256) - 1) / (256)))`’. Die Klammerung ist dabei durchaus nicht übertrieben. Das liegt daran, weil die Parameterübergabe nicht zur Laufzeit erfolgt, sondern bereits zur Übersetzungszeit, und weil der Präprozessor nicht wirklich Ahnung von C hat. Nehmen wir an, wir hätten das Makro fälschlich so definiert:

```
#define NEXT_MULTIPLE_OF(x, y) (y*(x + y - 1) / y)
```

und wir rufen es als ‘`NEXT_MULTIPLE_OF(b & c, sizeof(int))`’ auf, dann wird das zu ‘`(sizeof(int)*(b & c + sizeof(int) - 1) / sizeof(int))`’, was nach den Präzedenzregeln für C nicht das tut, was wir wollen, weil bitweises Und stärker bindet als Addition: Der Ausdruck nach der Makroersetzung ist also äquivalent zu ‘`(sizeof(int)*(b & (c + sizeof(int)) - 1) / sizeof(int))`’.

Trotzdem ist diese Makrodefinition nicht gut. Das liegt daran, dass im Ersetzungstext der Parameter y mehrfach auftritt. Wenn der tatsächliche Parameter nur ein Ganzzahl-Literal oder ein einfacher Ausdruck ist, ist das kein Problem. Was aber, wenn der Ausdruck Seiteneffekte hat? Nehmen wir an, wir rufen das Makro auf und übergeben als zweiten Parameter ‘`x++`’. Dann steht plötzlich im Ersetzungstext drei Mal ‘`x++`’, mit unabsehbaren Folgen.

Nehmen Sie in so einem Fall lieber eine Funktion, die Sie **static inline** deklarieren (**inline** gibt es nicht im Sprachstandard, sondern nur in gcc):

```
static inline unsigned
next_multiple_of(unsigned x, unsigned y) {
  return y * ((x + y - 1) / y);
}
```

5.12.3 Bedingte Übersetzung: `#if` und Verwandte

Will man Programmtext einmal für mehrere Umgebungen (z.B. Linux mit gcc und Windows mit Visual C) schreiben, dann hat man in der Regel ein Problem. Beispielsweise existiert unter Linux der Header *<windows.h>* nicht. Man müsste also dafür sorgen, dass dieser Header nur eingefügt wird, wenn man unter Windows übersetzt. Diese Entscheidung kann man nun aber nicht zur Laufzeit treffen, also kommt man mit normalen **if**-Anweisungen nicht weiter. Hier hilft die bedingte Übersetzung.

C-Präprozessoren definieren eine Reihe von Makros vor. Beispielsweise wird ein C-Präprozessor das Makro `__STDC__` definieren, wenn er ANSI-C-konform ist. Aber der Präprozessor definiert auch andere hilfreiche Makros. So definiert der Präprozessor von Visual C beispielsweise das Makro `_WIN32_WINNT_WIN7`, wenn man für Windows 7 übersetzt und das Makro `WINVER` ist definiert unt enthält einen von 0 verschiedenen Wert. Will man also die Header-Datei *<windows.h>* nur bei der Übersetzung für Windows einfügen, und *<sys/system.h>* bei der Übersetzung für Linux, kann man das so schreiben:

```
#if defined(WINVER)
#  include <windows.h>
#elif defined(__linux__)
#  include <sys/system.h>
#endif
```

Man kann mit `#if` auch einfache Vergleiche anstellen:

```
#if WINVER >= 0x0600
   // We're on Vista or higher
   ...
#endif
```

Es gibt noch die Kurzform `#ifdef` – `#else` – `#endif`, falls nur auf Existenz eines Makros getestet werden soll und nur maximal zwei Zweige zu behandeln sind. Der `#else`-Zweig ist wie üblich optional.

5.12.4 Sonstiges: `#line`, `#error`, `#pragma`

5.13 Schutz vor Mehrfachem Einfügen

Selbstverständlich ist es möglich, mit `#include`-Anweisungen Zykel zu erzeugen und damit einen nicht endenden Präprozessorlauf. Das kann man mit Makros und bedingter Übersetzung verhindern. Dazu kapseln Sie jede Header-Datei in ihre eigenen sogenannten *include guard* ein:

```
/* This is file editor/text/util.h */
#ifndef _EDITOR_TEXT_UTIL_H_
#  define _EDITOR_TEXT_UTIL_H_

   /* Contents of include file goes here */
```

```
#endif /* _EDITOR_TEXT_UTIL_H_ */
```

Wenn jetzt durch eine Verkettung irgendwelcher Umstände diese Datei ein zweites Mal angefordert werden sollte, ist das Makro `_EDITOR_TEXT_UTIL_H_` bereits definiert und der Inhalt der Datei wird nicht erneut eingefügt.

5.14 Ein- und Ausgabe

C hat *im Sprachumfang* keine Ein- oder Ausgabebefehle! Es stehen lediglich Bibliotheksfunktionen zur Verfügung, die entsprechende Funktionen realisieren.

5.14.1 Streams

Das Ein-Ausgabemodell von C fusst auf dem Konzept eines *streams*. Ein stream ist dabei eine Folge von Zeichen, die aus Sicht der C-Bibliothek ganz ohne innere Struktur ist. Es ist also z.B. nicht Aufgabe der C-Bibliothek, zu prüfen, dass eine Datei namens *mypic.jpg* tatsächlich ein korrekt formatiertes JPEG-Bilt enthält, das muss die Anwendung selbst machen.

Streams werden üblicherweise entweder gelesen oder geschrieben, aber selten beides zusammen. Manche streams unterstützen das Konzept einer stream-Position, die sich beliebig verändern lässt. Streams, die auf Festplatten liegen gehören dazu, streams, die von der Tastatur gelesen werden, nicht.

Wenn ein C-Programm startet, sind zum Zeitpunkt des Aufrufs von *main* drei streams geöffnet: der stream *stdin* bezeichnet die “Standard-Eingabe” und ist oft mit der Tastatur verbunden; *stdout* bezeichnet die “Standard-Ausgabe” und ist oft mit einem Terminal-Emulator verbunden; *stderr* bezeichnet die “Standard-Fehlerausgabe”. Dieser Stream ist oft ebenfalls mit dem Terminal-Emulator verbunden.

Diese Streams können Sie in der Shell nach Belieben umlenken. Nehmen wir an, Sie haben ein Programm *intoout*, dass Eingaben von *stdin* liest und nach *stdout* kopiert. Wenn Sie jetzt eine Datei *quelle* und diese in eine Datei *ziel* kopieren möchten, können Sie in der Shell folgendes schreiben:

```
./inttoout <quelle >ziel
```

Diese Syntax sorgt dafür, dass *stdin* nun aus der Datei *quelle* kommt und *stdout* nun in die Datei *quelle* geht. Fehlermeldungen erhalten Sie immer noch auf die Konsole. Wollen Sie Fehlermeldungen z.B. in die Datei *errs* speichern, geht das so:

```
./inttoout <quelle >ziel 2>errs
```

Das geht deshalb, weil *stderr* die Nummer 2 trägt. Der stream *stdin* hat die Nummer 0 und *stdout* die Nummer 1 (warum das so ist, würde an dieser Stelle zu weit führen).

5.14.2 Zeichenweise Ein/Ausgabe: *getc*, *putc*

Hier sollen nur die in <*stdio.h*> deklarierten Funktionen *getc* und *putc* kurz beschrieben werden, die jeweils ein einzelnes Zeichen von *stdin* lesen, bzw. nach *stdout* ausgeben. Diese Funktionen können dann benutzt werden, um komplexere Ein- und Ausgaben zu realisieren. Mit diesen beiden Funktionen können wir nun das Programm *inttoout* schreiben:

```
#include <stdio.h>

int main() {
  int c;
  while ((c = getc(stdin)) != EOF)
    putc(c, stdout);
  return 0;
}
```

Die Variable *c* ist als **int** definiert, weil *getc* neben allen möglichen **char**-Werten noch einen besonderen Wert zum Anzeigen des Dateiendes zurückgibt (*EOF*, end of file). Also muss *getc* mehr Werte zurückgeben können, als **char** aufnehmen kann.

Die verwendete Syntax für die Abbruchbedingung der **while**-Schleife ist ziemlich gewöhnungsbedürftig, aber idiomatisch und Standard. Hier zeigt sich, dass eine Zuweisung in C (und auch in C++) keine Anweisung ist, sondern ein Ausdruck, der einen Wert hat, nämlich dessen rechte Seite. Durch die Klammerug wird erreicht, dass zunächst *getc*(*stdin*) aufgerufen und das Resultat an *c* zugewiesen wird. Dann wird dieser Wert mit *EOF* verglichen.

Für *getc*(*stdin*) gibt es die um zwei Zeichen kürzere Schreibweise *getchar*() und für *putc*(*c*, *stdout*) existiert auch *putchar*(*c*).

Die Funktion *getc*, wie auch die meisten anderen Eingabefunktionen, bietet keine Möglichkeit vor Aufruf festzustellen, ob ein Zeichen vorliegt und blockiert den Programmablauf solange, bis ein Zeichen vorhanden ist. Eine Möglichkeit, das Vorhandensein von Eingabedaten festzustellen, ist die Benutzung der Funktion *select*; siehe dazu die einschlägige Dokumentation.

5.14.3 Formatierte Ausgabe, *fprintf*

Die Funktion *fprintf*, deklariert in <*stdio.h*> dient der formatierten Ausgabe auf einen stream. Dazu wird der Funktion als Erstes der stream übergeben, auf den die Ausgabe erfolgen soll. Dann folgt der sogenannte *Formatstring*, den wir ab dem nächsten Absatz genauer behandeln, und darauf folgen eine variable Anzahl an Argumenten. (C beherrscht im Gegensatz zu Java variable Argumente in Funktionen.) Will man auf *stdout* ausgeben, kann man auch *printf* verwenden. Dabei ist `printf(...)` zu `fprintf(stdout, ...)` äquivalent.

Der Formatstring ist das Kernstück von *printf*. Dieser Formatstring beschreibt die zu tätigende Ausgabe und die Typen der nach ihm folgenden Argumente. Das hört sich kompliziert an, lässt sich aber an einem Beispiel leicht erklären:

```
unsigned int m, n, r;
// ...
printf("Remainder of %u divided by %u is %u\n", m, n, r);
```

Ein Formatstring besteht demnach aus normalen Zeichen, die einfach auf den Ausgabestrom kopiert werden, und Formatanweisungen, die mit einem Prozentzeichen '`%`' eingeleitet werden. In unserem Beispiel bedeutet z.B. die erste Formatanweisung, dass der erste Parameternach dem Formatstring vom Typ **unsigned** ist und zur Basis 10 mit der kleinstmöglichen Anzahl an Ziffern ausgegeben werden soll. (Die Basis 10 und die kleinstmögliche Anzahl Ziffern sind dabei default.)

Eine vollständige Beschreibung der möglichen Formatanweisungen führt hier zu weit; wir haben im folgenden Beispiel einige der häufigeren (und einige der selten

genutzten aber nützlichen) Formate zusammengestellt. Beachten Sie, dass der **double**-Wert korrekt gerundet wird.

```
// %20s = format string with minimum width of 20 places,
// right-justified.  This gives "        hello, world"
const char* s = "hello, world";
printf("hello = \"%20s\"\n", s);

// %02u = format with minimum width 2 and leading zeros
// This gives '2012-02-27'
unsigned int year = 2012, month = 2, day = 27;
printf("%04u-%02u-%02u\n", year, month, day);

// %.5f = format a double with 4 decimal places after the point
// This gives '3.1416'
double f = 3.14159265;
printf("pi = %.4f\n", f);

// %08x = format as hex with 8 places and leading zeros
// This gives '8080ffff'. For capital latters, use %08X
unsigned int mask = 0x8080ffff;
printf("mask = %08x\n", mask);

// %p = format pointer (for debugging)
// This gives machine-dependent results
printf("hello string = %p\n", s);

// %zu = format as size_t (unsigned)
// This gives machine-dependent results
size_t a = sizeof(int), b = sizeof(double), c = sizeof(void*);
printf("int = %zu, double = %zu, void* = %zu\n", a, b, c);
```

5.14.4 Dateien öffnen und schliessen

Streams haben in der Standard C Bibliothek den Datentyp **FILE**.

Dateien werden mit *fopen* geöffnet und mit *fclose* wieder geschlossen. Die Funktion *fopen* erhält als erstes Argument den Namen der zu öffnenden Datei und als zweites Argument den Modus, in dem die Datei geöffnet werden soll (zum lesen, zum schreiben, ...; siehe dazu Tabelle 8). Im Erfolgsfall liefert liefert *fopen* einen Wert vom Typ **FILE*** zurück, im Fehlerfall den Wert *NULL*.

Besondere Beachtung verdient dabei der Unterschied von Text- und Binärdateien. C wurde erfunden, speziell zu dem Zweck, Unix zu portieren. In Unix gibt es nur ein Zeilenendezeichen, '`\n`', was in ASCII dem Zeilenvorschub (Linefeed, dezimal 10, oktal 012) entspricht. In anderen Betriebssystemen, speziell Windows, wird ein Zeilenende durch zwei Zeichen angezeigt, nämlich dem Wagenrücklauf (Carriage Return, `\r`, in ASCII dezimal 13, oktal 015) *und* dem Zeilenvorschub. Das ist ein Überbleibsel aus der Steuerung von Telefaxgeräten, wo man, um eine neue Zeile zu beginnen, sowohl dafür sorgen musste, dass der "Wagen" (auf dem sich die Mechanik zur Buchstabenerzeuging befand) an den linken Papierrand zurückgeführt wurde, als auch, dass das Papier um eine Zeile vorgeschoben wurde.

Obwohl es in Unix keinen Unterschied zwischen Text- und Binärdateien gibt, sollte der C-Sprachstandard nun garantieren, dass es auf C-Ebene keine Rolle spielt, wie

Wert	Bedeutung
`r`	Textdatei zum Lesen öffnen
`w`	Textdatei leeren oder neue Textdatei zum Schreiben erzeugen
`a`	anfügen; Textdatei neu erzeugen oder zum Schreiben am Dateiende öffnen
`rb`	Binärdatei zum Lesen öffnen
`wb`	Binärdatei leeren oder neue Binärdatei zum Schreiben erzeugen
`ab`	anfügen; Binärdatei neu erzeugen oder zum Schreiben am Dateiende öffnen
`r+`	Textdatei zum Aktualisieren öffnen (Lesen und Schreiben)
`w+`	Textdatei leeren oder neue Textdatei zum Aktualisieren erzeugen
`a+`	anfügen; Textdatei neu erzeugen oder zum Aktualisieren am Dateiende öffnen
`r+b` oder `rb+`	Binärdatei zum Aktualisieren öffnen (Lesen und Schreiben)
`w+b` oder `wb+`	Binärdatei leeren oder neue Binärdatei zum Aktualisieren erzeugen
`a+b` oder `ab+`	anfügen; Binärdatei neu erzeugen oder zum Aktualisieren am Dateiende öffnen

Tabelle 2: Gültige Werte für das zweite *fopen*-Argument.

das Zeilenende im unterliegenden Betriebssystem repräsentiert ist. Wenn man das aber erreichen will, muss man der Bibliothek mitteilen, ob es sich bei der zu öffnenden Datei um eine Text- oder Binärdatei handelt, weil ja z.B. in einer JPEG-Datei durchaus die Byte-Werte 15 und 10 hintereinander vorkommen können, die dann natürlich nicht zu `\n` zusammengefasst werden dürfen.

Die Funktion *fclose* schliesst eine mit *fopen* geöffnete Datei wieder. Wollen SIe beispielsweise die Zeichen und die vollständigen Zeilen in der Textdatei *a* zählen (also die Zeilen, die mit einem Zeilenendezeichen abgeschlossen sind), können Sie das so machen:

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

int main() {
  unsigned int nlines = 0;
  unsigned int nchars = 0;
  FILE* in = fopen("a", "r");

  if (in == NULL) {
    fprintf(stderr, "Cannot open input file");
    exit(EXIT_FAILURE);
  }

  int c;
  while ((c = getc(in)) != EOF) {
    nchars++;
    if (c == '\n')
```

```
      nlines++;
  }
  (void) fclose(in); // ignore errors

  printf("%u characters, %u lines\n", nchars, nlines);
  return 0;
}
```

Wollen Sie die Länge einer Datei *a* in Bytes erfahren, können Sie das einfach so machen, indem Sie im obigen Programm den *fopen*-Parameter `"r"` durch `"rb"` ersetzen. Sie sollten dann für die Anzahl an Bytes unter Windows einen höheren Wert erzielen als unter Unix. Warnung: dies dient nur Verdeutlichung des Unterschieds zwischen Text- und Binärdateien; so sollten Sie *nie* versuchen, die Länge einer Datei zu ermitteln!

5.14.5 Blockweises Lesen und Schreiben

Statt einzelner Zeichen können Sie auch grössere Einheiten diret in den Speicher lesen oder schreiben. Das geht mit den Funktionen *fread* und *fwrite*. Hier eine vermutlich (aber nicht sicher!) etwas schnellere Version des einfachen Kopierprogramms von oben. Hier wird die Datei *a* auf die Datei *b* kopiert. Beachten Sie die komplizierte Fehlerbehandlung: Eine Leseoperation kann weniger als `BUFFER_SIZE` Bytes zurückliedern, weil die Dateigrösse kein ganzzahliges Vielfaches dieser Grösse ist, oder weil es zu einem Lesefehler kam, also müssen Sie mit *ferror* nachschauen, ob es einen Lesefehler gab. Bei Schreiboperationen ist es garantiert, dass ein Schreibfehler aufgetreten ist, wenn der Rückgabewert kleiner als die zum Schreiben angeforderte Anzahl an Objekten ist (hier sind das **unsigned char**).

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

#define BUFFER_SIZE 10240

int main() {
  unsigned char buf[BUFFER_SIZE];
  FILE* in = fopen("a", "rb");

  if (in == NULL) {
    fprintf(stderr, "Can't open input file\n");
    exit(EXIT_FAILURE);
  }

  FILE* out = fopen("b", "wb");
  if (out == NULL) {
    fprintf(stderr, "Can't open output file\n");
    exit(EXIT_FAILURE);
  }

  size_t bytes_read = fread(buf, BUFFER_SIZE, 1, in);
  while (bytes_read > 0) {
    size_t bytes_written = fwrite(buf, bytes_read, 1, out);
    if (bytes_writtten != bytes_read) {
      fprintf(stderr, "write error");
```

```
      exit(EXIT_FAILURE);
    }
  }

  if (ferror(in)) {
    fprintf(stderr, "write error");
    exit(EXIT_FAILURE);
  }

  (void) fclose(out); // ignore errors
  (void) fclose(in); // ignore errors
  return 0;
}
```

5.15 Programmargumente

C bietet eine Möglichkeit um ein Programm über die Kommandozeile mit Argumenten zu versehen. Dazu erhält die Funktion *main* zwei Argumente, *argc*, und *argv*. Hierbei ist *argc* ein Wert vom Typ **int**, der die Anzahl der übergebenen Parameter enthält. Der Parameter *argv* ist ein Array von Strings, das die einzelnen Parameter-Strings enthält. Dabei ist *argv* genau um Eins grösser als *argc*, und es gilt $argv[argc] = NULL$.

```
int main(int argc, char *argv[]) {
  for (unsigned int i = 0; i < argc; i++)
    printf("parameter no. %u is \"%s\"\n", i,  argv[i]);

  // equivalent version using argv[argc] == NULL
  for (unsigned int i = 0; argv[i] != NULL; i++)
    printf("parameter no. %u is \"%s\"\n", i,  argv[i]);
}
```

6 Referenz: Datentypen und Operatoren

6.1 Elementare Datentypen

In C (Standard C99) gibt es die folgenden elementaren Datentypen (die Folgende Liste kommt direkt aus dem Standard). Obwohl es diese Typen auch in C++ gibt, führen wir sie hier an, weil oft falsche Informationen über solche Typen kursieren, z.B. “ein **long** ist 32 bit lang”, oder “ein **int** ist kleiner als ein **long long**”.

- **_Bool**, reicht garantiert aus, um dort die Werte 0 und 1 zu speichern.
- **char**, reicht aus, um ein Zeichen des zugrundeliegenden Zeichensatzes zu speichern. In C unter Linux auf Intel-Plattformen ist das in der Regel 7-bit ASCII. Wird ein Zeichen aus diesem Zeichensatz in einem **char** gespeichert, ist der numerische Wert garantiert nicht negativ. Speichert man Werte ausserhalb des Zeichensatzes, kann man keine Aussagen über das Vorzeichen machen.
- Die fünf vorzeichenbehafteten Ganzzahltypen: **signed char**, **short int**, **int**, **long int**und **long long int**. Ein **signed char** benötigt denselben Speicherplatz wie ein normaler **char**.

Typ	Makro	Mindestwert	Formel
signed char	SCHAR_MIN	-127	$-(2^7-1)$
	SCHAR_MAX	$+127$	2^7-1
unsigned char	UCHAR_MAX	255	2^8-1
short	SHRT_MIN	-32767	$-(2^{15}-1)$
	SHRT_MAX	$+32767$	$2^{15}-1$
unsigned short	USHRT_MAX	65535	$2^{16}-1$
int	INT_MIN	-32767	$-(2^{15}-1)$
	INT_MAX	$+32767$	$2^{15}-1$
unsigned int	UINT_MAX	65535	$2^{16}-1$
long	LONG_MIN	-2147483647	$-(2^{32}-1)$
	LONG_MAX	$+2147483647$	$2^{32}-1$
unsigned long	ULONG_MAX	4294967295	$2^{32}-1$
long long	LLONG_MIN	-9223372036854775807	$-(2^{63}-1)$
	LLONG_MAX	$+9223372036854775807$	$2^{63}-1$
unsigned long long	ULLONG_MAX	18446744073709551615	$2^{64}-1$

Tabelle 3: Mindestwertebereiche der Ganzzahltypen

- Für jeden vorzeichenbehafteten Ganzzahltyp gibt es einen entsprechenden vorzeichenlosen Typ, der mit dem Schlüsselwort **unsigned** eingeleitet wird: **unsigned char**, **unsigned short int**, **unsigned int**, **unsigned long int** und **unsigned long long**. Diese Typen benötigen denselben Speicherplatz wie ihre vorzeichenbehafteten Gegenstücke.
- Die Wertebereiche der vorzeichenbehafteten Ganzzahlen ist eine Untermenge der entsprechenden vorzeichenlosen Ganzzahlen; speichert man einen nichtnegativen Wert in einer vorzeichenbehafteten Ganzzahl, ist ihre interne Repräsentation gleich der Repräsentation in einer vorzeichenlosen Ganzzahl.
- Berechnungen, die mit **unsigned** Datentypen durchgeführt werden, werden immer modulo (maximaler Wert des **unsigned**-Type plus 1) durchgeführt.
- Ein normaler **char** ist identisch entweder mit einem **signed char** oder einem **unsigned char**.
- Es gibt drei reelle Gleitkommatypen, **float**, **double** und **long double**; die Wertebereiche sind Untermengen voneinander: **float** von **double** und **double** von **long double**.
- Es gibt zu jedem reellen Gleitkommatyp auch ein komplexes Gegenstück: **float _Complex**, **double _Complex** und **long double _Complex**. Diese sind dann jeweils so gross wie ein Array aus zwei entsprechenden reellen Gleitkommatypen.

In C ist weder die Länge eines elementaren Datentyps in Bytes noch seine Repräsentation garantiert. Garantiert sind lediglich bestimmte Mindestwertebereiche und bestimmte Grössenhierarchien. Tabelle 3 entstammt aus dem C-Standard-Dokument und gibt Mindestwerte aus <*limits.h*> wieder. Makros die auf _MIN enden, geben dabei den kleinsten Wert wirder, Makros, die auf _MAX enden, den grössten.

Makro	Bedeutung	FLT	DBL	LDBL
DIG	Anzahl unterstützter Dezimalstellen	6	10	10
MIN_10_EXP	Kleinster unterstützter Zehnerexponent	-37	-37	-37
MAX_10_EXP	Grösster unterstützter Zehnerexponent	$+37$	$+37$	$+37$
MIN	Kleinste normalisierte Zahl	10^{-37}	10^{-37}	10^{-37}
MAX	Grösste Zahl	10^{37}	10^{37}	10^{37}
EPSILON	Wert von b^{1-p}	10^{-5}	10^{-9}	10^{-9}

Tabelle 4: Mindestwertebereiche der Gleitkommatypen

Makro	FLT	DBL
DIG	6	15
MIN_10_EXP	-37	-307
MAX_10_EXP	$+38$	$+308$
MIN	$1.17549435 \cdot 10^{-38}$	$2.2250738585072014 \cdot 10^{-308}$
MAX	$3.40282347 \cdot 10^{+38}$	$1.7976931348623157 \cdot 10^{+308}$
EPSILON	$1.19209290 \cdot 10^{-07}$	$2.2204460492503131 \cdot 10^{-16}$

Tabelle 5: Werte für eine IEEE 754-konforme Implementation

Die Charakteristika von Gleitkommazahlen werden entsprechend einem parametrisierten Modell beschrieben. Demnach besteht eine Gleitkommazahl aus einer ganzzahligen Basis $b > 1$, einem Vorzeichen s (entweder $+1$ oder -1), einem ganzzahligen Exponenten e, der zwischen zwei Extremwerten $e_{\min}$ und $e_{\max}$ liegt, der Anzahl der Ziffern p und den ganzzahligen Ziffern f_k, die zwischen 0 und $b-1$ liegen. Eine Gleitkommazahl x ist dann definiert durch das Modell $x = sb^e \sum_{k=1}^{p} f_k b^{-k}$. Wir betrachten hier nur Zahlen, bei denen $f_1 > 0$ ist (sogenannte *normalisierte* Zahlen). Die Gleitkomma-Implementation laut C-Standard hat die in Tabelle 4 angegebenen Mindestwerte. (Der Grund für diese merkwürdig anmutende Parametrisierung ist, dass es Gleitkommarepräsentationen zur Basis 10 gibt.)

Unter Linux auf Intel wird der Gleitkomma-Standard IEEE 754 unterstützt. Die entsprechenden Wertebereiche finden Sie dann in Tabelle 5. Suchen Sie einene bestimmten Wert, können Sie den zugehörigen Makro-Namen aus Zeilen- und Spaltennamen zusammensetzen: Das Epsilon für den Typ **double** wäre demnach DBL_EPSILON.

IEEE 754 besitzt besondere Werte für "Not a Number" (z.B. das Resultat der Division von 0.0 durch 0.0) $+\infty$ und $-\infty$, kann $+0$ von -0 unterscheiden, besitzt verschiedene Rundungsmodi (runde auf nächstliegende repräsentierbare Zahl, runde Richtung 0, runde weg von 0, usw.). Wenn Sie das wirklich brauchen, sollten Sie sich mit dem Standard genauer vertraut machen.

Im C99-Standard existieren Typen mit einer exakten Längengarantie. Diese sind in der Datei <*stdint.h*> definiert. Z.B. ist **int32_t** eine Ganzzahl mit Vorzeichen mit exakt 32 bit Länge.

6.2 Zahl-Literale

Ein *Zahl-Literal* ist eine Zeichenfolge[2], die vom Compiler als Repräsentation einer Zahl interpretiert wird. So wird die Zeichenfolge '`1`' als **int**-Literal interpretiert und '`1.0`' als **double**-Literal. Die Zeichenfolge '`q`' ist gar kein Zahl-Literal.

Bei Zahl-Literalen kann man oft bestimmen, in welcher *Basis* sie angegeben werden, also z.B. als dezimale Literale zur Basis 10 oder als hexadezimale Literale zur Basis 16. Bei Ganzzahl-Literalen kann man auch den *Mindest-Typ* bestimmen, also ob es sich um ein **long long**-Literal handeln soll, selbst wenn der Wert vielleicht noch in einen **int** hineinpasst.

6.2.1 Ganzzahl-Literale

Bei Ganzzahl-Literalen wird die Basis so angegeben: fängt das Literal mit einer von '`0`' verschiedenen Ziffer an und besteht es nur aus Ziffern, ist es zur Basis 10 (dezimal) notiert. Ist das erste Zeichen '`0`', und besteht das Literal ansonsten nur aus den Ziffern '`0`' bis '`7`', ist das Literal zur Basis 8 (oktal). Sind die ersten zwei Zeichen des Literals '`0x`' oder '`0X`', gefolgt von dezimalen Ziffern und den Buchstaben `a` bis `f`, bzw. `A` bis `F` (für die Ziffern 10–15), ist das Literal zur Basis 16 (hexadezimal) notiert.

```
int i = 213;
int j = 0315;
int k = 0xd5;
```

Hier haben alle Variablen denselben Wert, weil $(213)_{10} = (0315)_8 = (\texttt{d5})_{16}$.

Den Mindest-Typ kann man bei Ganzzahl-Literalen durch ein Suffix angeben. Die Suffixe '`l`' und '`L`' stehen dabei für den Datentyp **long**, die Suffixe '`ll`' und '`LL`' für **long long**. Davor oder danach darf (muss aber nicht) noch ein Suffix '`u`' oder '`U`' erscheinen, das den Typ auf den entsprechenden **unsigned**-Typ setzt, also z.B. '`UL`' für **unsigned long**. Man keinen **short**-Datentyp erzwingen; Literale vom Typ **short** existieren nicht.

Aus dem Literal, zusammen mit eventuell vorhandenen Suffixen, wird dessen Typ berechnet. Im wesentlichen ist das der kleinste Typ, der sowohl mit dem Literal als auch mit dem Suffix zusammen kompatibel ist. Details stehen in Tabelle 6, die dem C-Standard entnommen ist.

Es gibt in C auch noch erweiterte Ganzzahl-Typen, aber es ist unklug, sich auf deren Existenz oder gar deren Wertebereiche zu verlassen.

6.2.2 Gleitkomma-Literale

6.3 Typerweiterung

Es ist in C erlaubt, eine **int**-Variable einer **unsigned**-Variable zuzuweisen, oder Berechnungen zu erweitern. Das passiert häufiger als man denkt. Nehmen wir folgende C-Fragmente:

```
long double ld = 1.0;
unsigned i = 1 + 1;
int j = 10;
unsigned int k = 8;
k -= j;
```

[2]Eigentlich ein *token*, aber wir wollen nicht weiter darauf eingehen.

Suffix	Dezimal	Oktal oder Hex
keines	**int**	**int**
		unsigned
	long	**long**
		unsigned long
	long long	**long long**
		unsigned long long
`u` oder `U`	**unsigned**	**unsigned**
	unsigned long	**unsigned long**
	unsigned long long	**unsigned**
`u` oder `U`	**unsigned long**	**unsigned long**
und `l` oder `L`	**unsigned long long**	**unsigned long long**
`ll` oder `LL`	**long long**	**long long**
		unsigned long long
`u` oder `U`	**unsigned long long**	**unsigned long long**
und `l` oder `L`		

Tabelle 6: Typen von Ganzzahl-Konstanten

In diesem Fall ist '`1.0`' eine Konstante vom Typ **double**, *ld* hat aber den Typ **long double**. Also muss die Konstante '`1.0`' auf **long double** erweitert werden. Der Typ des Ausdrucks '`1 + 1`' ist **int**, der Typ von *i* aber **unsigned**.

Besonders schwierig wird es in den jetzten drei Zeilen. Schwierig ist zunächst die Erweiterung von *j* von **int** auf **unsigned**: was passiert, wenn *j* negativ ist? Aber selbst wo das nicht der Fall ist, lauern Fallstricke: der Minuend ist kleiner als der Subtrahend! Es gibt nun spezielle Regeln, die in diesem Fall Klarheit bringen sollen, was zu geschehen hat. Diese sind wie die Regeln von den Signal-sicheren Funktionen sehr, sehr kompliziert. weshalb wir Ihnen dringend raten wollen, sich nicht auf Ihre umfassende Kenntnis dieser Regeln zu verlassen! Wenn Sie diese information dennoch einmal benötigen sollten, konsultieren Sie den Standard.

Selbstverständlich kann es bei der Zuweisung von einem "langen" zu einem "kurzen" Typ dazu kommen, dass Werte nicht exakt konvertiert werden können. Da diese Probleme aber auch in Java und C++ gleichermassen existieren, gehen wir hier nicht näher darauf ein.

6.4 Bitweise Operatoren

Bitoperatoren sind abgewandelte logische Operatoren, die auf den Bits der Argumente arbeiten. Sie können auf alle ganzzahligen Typen angewendet werden; siehe Tabelle 7. Das Ergebnis ist vom Typ der Argumente und kann (je nach Operator) jeden Wert annehmen.

Schiebeoperationen können nur mit nichtnegativen Schiebewerten durchgeführt werden. Wird ein negativer Wert eingesetzt, ist das Ergebnis nicht definiert. Gleiches gilt, wenn man mehr Bitpositionen schieben will, als der linke Operand hat.

Beim Linksschieben werden von rechts Nullen nachgeschoben. Das ist von C Standard so garantiert. Beim Rechtsschieben werden von links Nullen nachgeschoben, wenn der linke Operand einen **unsigned**-Datentyp hat oder wenn er einen vorzeichen-

Operator	Bedeutung
`a & b, a | b`	Bitweises UND, bitweises ODER .
`a ^ b`	Bitweises XOR
`$~a`	Bitweise Negation
`a >> b, a << b`	Bitweise Rechts-/Linksverschiebung um `b` Stellen.

Tabelle 7: Bitoperatoren

behafteten Datentyp hat und der Wert nichtnegativ ist. Wenn aber der linke Operand vorzeichenbehaftet und negativ ist, muss entschieden werden, ob Nullen oder Einsen von links nachgeschoben werden. Laut C-Standard ist das Verhalten "implementation-defined", mit anderen Worten, wenn Sie ein bestimmtes Rechtsshift-Verhalten auf negativen Zahlen benötigen, müssen Sie sich das selbst bauen. Wir haben bislang jedoch noch nicht erlebt, dass man negative Zahlen nach rechts schieben muss.

Wenn man nach rechts schieben möchte, um eine Bitoperation vorzunehmen, sollte man als Datentyp für den linken Operanden ohnehin einen geeigneten **unsigned**-Typ wählen. Will man dann unbedingt Einsen nachschieben, kann man das so machen:

```
#include <limits.h>

static const n_unsigned_bits = sizeof(unsigned int)*CHAR_BIT;

unsigned int
rshift_ones(unsigned int value, unsigned int nbits) {
  if (nbits > n_unsigned_bits) {
    // Shifting by more bits than the left
    // operand has is undefined.
    ...; // Error handling goes here
  } else {
    return (value >> nbits) | (~0 << (n_unsigned_bits - nbits));
  }
}
```

Dieser Code führt bei standardkonformen C-Compilern zu keinerlei undefiniertem Verhalten und würde wohl auch bei einem Security-Audit durchgehen.

Wenn man die Zahl nach rechts schieben möchte, um durch eine Zweierpotenz zu dividieren, sollte man sich über die Wertebereiche des linken Operators Gedanken machen. Handelt es sich um eine nichtnegative Zahl, sollte man einen geeigneten **unsigned**-Typ wählen. Handelt es sich tatsächlich um eine möglicherweise negative Ganzzahl, sollte man einfach im Programmtext eine Division notieren und darauf vertrauen, dass der Compiler bei eingeschalteter Optimierung schon die richtigen Instruktionen wählen wird:

```
int rshift_divide(int value, unsigned int nbits) {
  if (nbits > sizeof(int)*CHAR_BIT) {
    // Shifting by more bits than the left
    // operand has is undefined.
    ...; // Error handling goes here
  } else {
    return value / (1 << nbits);
  }
}
```

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.—Donald E. Knuth, 1974

¡¡¡¡¡¡¡ .mine

6.5 Streams

Das Ein-Ausgabemodell von C fusst auf dem Konzept eines *streams*. Ein stream ist dabei eine Folge von Zeichen, die aus Sicht der C-Bibliothek ganz ohne innere Struktur ist. Es ist also z.B. nicht Aufgabe der C-Bibliothek, zu prüfen, dass eine Datei namens *mypic.jpg* tatsächlich ein korrekt formatiertes JPEG-Bild enthält, das muss die Anwendung selbst machen.

Streams werden üblicherweise entweder gelesen oder geschrieben, aber selten beides zusammen. Manche streams unterstützen das Konzept einer stream-Position, die sich beliebig verändern lässt. Streams, die auf Festplatten liegen gehören dazu, streams, die von der Tastatur gelesen werden, nicht.

Wenn ein C-Programm startet, sind zum Zeitpunkt des Aufrufs von *main* drei streams geöffnet: der stream *stdin* bezeichnet die "Standard-Eingabe" und ist oft mit der Tastatur verbunden; *stdout* bezeichnet die "Standard-Ausgabe" und ist oft mit einem Terminal-Emulator verbunden; *stderr* bezeichnet die "Standard-Fehlerausgabe". Dieser Stream ist oft ebenfalls mit dem Terminal-Emulator verbunden.

Diese Streams können Sie in der Shell nach Belieben umlenken. Nehmen wir an, Sie haben ein Programm *intoout*, dass Eingaben von *stdin* liest und nach *stdout* kopiert. Wenn Sie jetzt eine Datei *quelle* und diese in eine Datei *ziel* kopieren möchten, können Sie in der Shell folgendes schreiben:

```
./inttoout <quelle >ziel
```

Diese Syntax sorgt dafür, dass *stdin* nun aus der Datei *quelle* kommt und *stdout* nun in die Datei *quelle* geht. Fehlermeldungen erhalten Sie immer noch auf die Konsole. Wollen Sie Fehlermeldungen z.B. in die Datei *errs* speichern, geht das so:

```
./inttoout <quelle >ziel 2>errs
```

Das geht deshalb, weil *stderr* die Nummer 2 trägt. Der stream *stdin* hat die Nummer 0 und *stdout* die Nummer 1 (warum das so ist, würde an dieser Stelle zu weit führen).

6.6 Zeichenweise Ein/Ausgabe: *getc*, *putc*

Hier sollen nur die in <*stdio.h*> deklarierten Funktionen *getc* und *putc* kurz beschrieben werden, die jeweils ein einzelnes Zeichen von *stdin* lesen, bzw. nach *stdout* ausgeben. Diese Funktionen können dann benutzt werden, um komplexere Ein- und Ausgaben zu realisieren. Mit diesen beiden Funktionen können wir nun das Programm *inttoout* schreiben:

```
#include <stdio.h>

int main() {
  int c;
  while ((c = getc(stdin)) != EOF)
    putc(c, stdout);
  return 0;
}
```

Die Variable *c* ist als **int** definiert, weil *getc* neben allen möglichen **char**-Werten noch einen besonderen Wert zum Anzeigen des Dateiendes zurückgibt (*EOF*, end of file). Also muss *getc* mehr Werte zurückgeben können, als **char** aufnehmen kann.

Die verwendete Syntax für die Abbruchbedingung der **while**-Schleife ist ziemlich gewöhnungsbedürftig, aber idiomatisch und Standard. Hier zeigt sich, dass eine Zuweisung in C (und auch in C++) keine Anweisung ist, sondern ein Ausdruck, der einen Wert hat, nämlich dessen rechte Seite. Durch die Klammerug wird erreicht, dass zunächst *getc*(*stdin*) aufgerufen und das Resultat an *c* zugewiesen wird. Dann wird dieser Wert mit *EOF* verglichen.

Für *getc*(*stdin*) gibt es die um zwei Zeichen kürzere Schreibweise *getchar*() und für *putc*(*c*, *stdout*) existiert auch *putchar*(*c*).

Die Funktion *getc*, wie auch die meisten anderen Eingabefunktionen, bietet keine Möglichkeit vor Aufruf festzustellen, ob ein Zeichen vorliegt und blockiert den Programmablauf solange, bis ein Zeichen vorhanden ist. Eine Möglichkeit, das Vorhandensein von Eingabedaten festzustellen, ist die Benutzung der Funktion *select*; siehe dazu die einschlägige Dokumentation.

6.7 Formatierte Ausgabe, *fprintf*

Die Funktion *fprintf*, deklariert in <*stdio.h*> dient der formatierten Ausgabe auf einen stream. Dazu wird der Funktion als erstes der stream übergeben, auf dem die Ausgabe erfolgen soll. Dann folgt der sogenannte *Formatstring*, den wir ab dem nächsten Absatz genauer behandeln, und darauf folgen eine variable Anzahl an Argumenten. (C beherrscht im Gegensatz zu Java variable Argumente in Funktionen.) Will man auf *stdout* ausgeben, kann man auch *printf* verwenden. Dabei ist `printf(...)` zu `fprintf(stdout, ...)` äquivalent.

Der Formatstring ist das Kernstück von *printf*. Dieser Formatstring beschreibt die zu tätigende Ausgabe und die Typen der nach ihm folgenden Argumente. Das hört sich kompliziert an, lässt sich aber an einem Beispiel leicht erklären:

```
unsigned int m, n, r;
// ...
printf("Remainder of %u divided by %u is %u\n", m, n, r);
```

Ein Formatstring besteht demnach aus normalen Zeichen, die einfach auf den Ausgabestrom kopiert werden, und Formatanweisungen, die mit einem Prozentzeichen '`%`' eingeleitet werden. In unserem Beispiel bedeutet z.B. die erste Formatanweisung, dass der erste Parameter nach dem Formatstring vom Typ **unsigned** ist und zur Basis 10 mit der kleinstmöglichen Anzahl an Ziffern ausgegeben werden soll. (Die Basis 10 und die kleinstmögliche Anzahl Ziffern sind dabei default.)

Eine vollständige Beschreibung der möglichen Formatanweisungen führt hier zu weit; wir haben im folgenden Beispiel einige der häufigeren (und einige der selten genutzten aber nützlichen) Formate zusammengestellt. Beachten Sie, dass der **double**-Wert korrekt gerundet wird.

```
// %20s = format string with minimum width of 20 places,
// right-justified.  This gives "        hello, world"
const char* s = "hello, world";
printf("hello = \"%20s\"\n", s);

// %02u = format with minimum width 2 and leading zeros
```

```
// This gives '2012-02-27'
unsigned int year = 2012, month = 2, day = 27;
printf("%04u-%02u-%02u\n", year, month, day);

// %.5f = format a double with 4 decimal places after the point
// This gives '3.1416'
double f = 3.14159265;
printf("pi = %.4f\n", f);

// %08x = format as hex with 8 places and leading zeros
// This gives '8080ffff'. For capital latters, use %08X
unsigned int mask = 0x8080ffff;
printf("mask = %08x\n", mask);

// %p = format pointer (for debugging)
// This gives machine-dependent results
printf("hello string = %p\n", s);

// %zu = format as size_t (unsigned)
// This gives machine-dependent results
size_t a = sizeof(int), b = sizeof(double), c = sizeof(void*);
printf("int = %zu, double = %zu, void* = %zu\n", a, b, c);
```

6.8 Dateien öffnen und schliessen

Streams haben in der Standard C Bibliothek den Datentyp **FILE**. (Wenn die Dinger sowieso **FILE** heissen, wieso heissen sie dann stream? Gute Frage. Sagen Sie uns bescheid, wenn Sie die Antwort wissen.)

Dateien werden mit *fopen* geöffnet und mit *fclose* wieder geschlossen. Die Funktion *fopen* erhält als erstes Argument den Namen der zu öffnenden Datei und als zweites Argument den Modus, in dem die Datei geöffnet werden soll (zum lesen, zum schreiben, ...; siehe dazu Tabelle 8). Im Erfolgsfall liefert liefert *fopen* einen Wert vom Typ **FILE*** zurück, im Fehlerfall den Wert *NULL*.

Besondere Beachtung verdient dabei der Unterschied von Text- und Binärdateien. C wurde erfunden, speziell zu dem Zweck, Unix zu portieren. In Unix gibt es nur ein Zeilenendezeichen, '`\n`', was in ASCII dem Zeilenvorschub (Linefeed, dezimal 10, oktal 012) entspricht. In anderen Betriebssystemen, speziell Windows, wird ein Zeilenende durch zwei Zeichen angezeigt, nämlich dem Wagenrücklauf (Carriage Return, `\r`, in ASCII dezimal 13, oktal 015) *und* dem Zeilenvorschub. Das ist ein Überbleibsel aus der Steuerung von Telefaxgeräten, wo man, um eine neue Zeile zu beginnen, sowohl dafür sorgen musste, dass der "Wagen" (auf dem sich die Mechanik zur Buchstabenerzeuging befand) an den linken Papierrand zurückgeführt wurde, als auch, dass das Papier um eine Zeile vorgeschoben wurde.

Obwohl es in Unix keinen Unterschied zwischen Text- und Binärdateien gibt, sollte der C-Sprachstandard nun garantieren, dass es auf C-Ebene keine Rolle spielt, wie das Zeilenende im unterliegenden Betriebssystem repräsentiert ist. Wenn man das aber erreichen will, muss man der Bibliothek mitteilen, ob es sich bei der zu öffnenden Datei um eine Text- oder Binärdatei handelt, weil ja z.B. in einer JPEG-Datei durchaus die Byte-Werte 15 und 10 hintereinander vorkommen können, die dann natürlich nicht zu `\n` zusammengefasst werden dürfen.

Die Funktion *fclose* schliesst eine mit *fopen* geöffnete Datei wieder. Wollen Sie

Wert	Bedeutung
`r`	Textdatei zum Lesen öffnen
`w`	Textdatei leeren oder neue Textdatei zum Schreiben erzeugen
`a`	anfügen; Textdatei neu erzeugen oder zum Schreiben am Dateiende öffnen
`rb`	Binärdatei zum Lesen öffnen
`wb`	Binärdatei leeren oder neue Binärdatei zum Schreiben erzeugen
`ab`	anfügen; Binärdatei neu erzeugen oder zum Schreiben am Dateiende öffnen
`r+`	Textdatei zum Aktualisieren öffnen (Lesen und Schreiben)
`w+`	Textdatei leeren oder neue Textdatei zum Aktualisieren erzeugen
`a+`	anfügen; Textdatei neu erzeugen oder zum Aktualisieren am Dateiende öffnen
`r+b` oder `rb+`	Binärdatei zum Aktualisieren öffnen (Lesen und Schreiben)
`w+b` oder `wb+`	Binärdatei leeren oder neue Binärdatei zum Aktualisieren erzeugen
`a+b` oder `ab+`	anfügen; Binärdatei neu erzeugen oder zum Aktualisieren am Dateiende öffnen

Tabelle 8: Gültige Werte für das zweite *fopen*-Argument.

beispielsweise die Zeichen und die vollständigen Zeilen in der Textdatei *a* zählen (also die Zeilen, die mit einem Zeilenendezeichen abgeschlossen sind), können Sie das so machen:

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

int main() {
  unsigned int nlines = 0;
  unsigned int nchars = 0;
  FILE* in = fopen("a", "r");

  if (in == NULL) {
    fprintf(stderr, "Cannot open input file");
    exit(EXIT_FAILURE);
  }

  int c;
  while ((c = getc(in)) != EOF) {
    nchars++;
    if (c == '\n')
      nlines++;
  }
  (void) fclose(in); // ignore errors

  printf("%u characters, %u lines\n", nchars, nlines);
  return 0;
}
```

Wollen Sie die Länge einer Datei *a* in Bytes erfahren, können Sie das einfach so machen, indem Sie im obigen Programm den *fopen*-Parameter `"r"` durch `"rb"` ersetzen. Sie sollten dann für die Anzahl an Bytes unter Windows einen höheren Wert erzielen als unter Unix. Warnung: dies dient nur Verdeutlichung des Unterschieds zwischen Text- und Binärdateien; so sollten Sie *nie* versuchen, die Länge einer Datei zu ermitteln!

6.9 Blockweises Lesen und Schreiben

Statt einzelner Zeichen können Sie auch grössere Einheiten direkt in den Speicher lesen oder schreiben. Das geht mit den Funktionen *fread* und *fwrite*. Hier eine vermutlich (aber nicht sicher!) etwas schnellere Version des einfachen Kopierprogramms von oben. Hier wird die Datei *a* auf die Datei *b* kopiert. Beachten Sie die komplizierte Fehlerbehandlung: Eine Leseoperation kann weniger als `BUFFER_SIZE` Bytes zurückliedern, weil die Dateigrösse kein ganzzahliges Vielfaches dieser Grösse ist, oder weil es zu einem Lesefehler kam, also müssen Sie mit *ferror* nachschauen, ob es einen Lesefehler gab. Bei Schreiboperationen ist es garantiert, dass ein Schreibfehler aufgetreten ist, wenn der Rückgabewert kleiner als die zum Schreiben angeforderte Anzahl an Objekten ist (hier sind das **unsigned char**).

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

#define BUFFER_SIZE 10240

int main() {
  unsigned char buf[BUFFER_SIZE];
  FILE* in = fopen("a", "rb");

  if (in == NULL) {
    fprintf(stderr, "Can't open input file\n");
    exit(EXIT_FAILURE);
  }

  FILE* out = fopen("b", "wb");
  if (out == NULL) {
    fprintf(stderr, "Can't open output file\n");
    exit(EXIT_FAILURE);
  }

  size_t bytes_read = fread(buf, BUFFER_SIZE, 1, in);
  while (bytes_read > 0) {
    size_t bytes_written = fwrite(buf, bytes_read, 1, out);
    if (bytes_writtten != bytes_read) {
      fprintf(stderr, "write error");
      exit(EXIT_FAILURE);
    }
  }

  if (ferror(in)) {
    fprintf(stderr, "write error");
    exit(EXIT_FAILURE);
  }
```

```
  (void) fclose(out); // ignore errors
  (void) fclose(in); // ignore errors
  return 0;
}
```

7 Referenz: Programmargumente

C bietet eine Möglichkeit um ein Programm über die Kommandozeile mit Argumenten zu versehen. Dazu erhält die Funktion *main* zwei Argumente, *argc*, und *argv*. Hierbei ist *argc* ein Wert vom Typ **int**, der die Anzahl der übergebenen Parameter enthält. Der Parameter *argv* ist ein Array von Strings, das die einzelnen Parameter-Strings enthält. Dabei ist *argv* genau um Eins grösser als *argc*, und es gilt $argv[argc] = NULL$.

```
int main(int argc, char *argv[]) {
  for (unsigned int i = 0; i < argc; i++)
    printf("parameter no. %u is \"%s\"\n", i,  argv[i]);

  // equivalent version using argv[argc] == NULL
  for (unsigned int i = 0; argv[i] != NULL; i++)
    printf("parameter no. %u is \"%s\"\n", i,  argv[i]);
}
```

8 Referenz: Debugging

8.1 *printf*

Die am häufigsten verwendete und auch einfachste Variante zum Debuggen ist, sich Werte oder Pointer mittels `printf` ausgeben zu lassen.

```
struct foo {
  int data;
  struct foo *next;
};

int main(void) {
  struct foo *ptr = xmalloc(...)
  ...
  printf("%p=(%d,%p)\n", ptr, ptr->data, ptr->next);
  ...
  free(ptr);
}
```

8.2 GDB, dein Freund und Helfer

Der GNU Debugger (`gdb`) ist der älteste und auch am häufigsten genutzte Debugger für C. Er kann beispielsweise benutzt werden, um sich einen Stacktrace von einem Speicherfehler wie zum Beispiel eines *Segmentation Faults* anzusehen. Betrachten wir das folgende fehlerhafte Programm `foo.c`:

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

```
struct foo {
  int data;
  struct foo *next;
};

int main(void) {
  struct foo *ptr = NULL;
  if (ptr->data == 5) {
     printf("foo\n");
  }
  return 0;
}
```

Dieses Programm kompiliert ohne Probleme mittels `gcc -Wall -O2 foo.c`. Bei der Ausführung erscheint jedoch ein `Segmentation fault`, das heisst das Betriebssystem beendet das Programm, weil es versucht, auf einen nicht erlaubten Speicherbereich zuzugreifen - in unserem Falle einen Null-Pointer zu dereferenzieren.

Nun kann man mittels `gdb` sich anzeigen lassen, wo genau dieser Fehler verursacht wird. Dazu muss man das obige Programm mit der `gcc`-Option `-g` kompilieren, um auch Debug-Symbole in die Binärdatei zu inkludieren, also `gcc -Wall -g -O2 foo.c`. Ferner muss in der aktuellen Konsole eingestellt sein, dass der Linux Kernel einen sogenannten *Coredump* schreiben soll falls es zum Absturz kommt. Dieser beinhaltet einen Schnappschuss des Programm- und Speicherzustands bevor der Kernel das Programm forciert beendet hat. Eintippen von `ulimit -c unlimited` aktiviert dies. Wird das Programm nun nocheinmal ausgeführt, so erhalt man die Ausgabe: `Segmentation fault (core dumped)`. Das heisst, der Kernel hat nun in den aktuellen Ordner eine Datei names `core` erstellt, die mit `gdb ./a.out core` weiter analysiert werden kann.

```
$ gdb ./a.out core
GNU gdb (GDB) 7.0.1-debian
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...

Reading symbols from /home/bordanie/Desktop/a.out...done.
Reading symbols from /lib/libc.so.6...done.
Reading symbols from /usr/lib/debug/lib/libc-2.11.3.so...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib64/ld-linux-x86-64.so.2...done.
Reading symbols from /usr/lib/debug/lib/ld-2.11.3.so...done.
Loaded symbols for /lib64/ld-linux-x86-64.so.2
Core was generated by `./a.out'.
Program terminated with signal 11, Segmentation fault.
#0  main () at foo.c:10
10   if (ptr->data == 5) {
(gdb)
```

`gdb` zeigt sofort, dass das Programm abnormal durch das Signal Nummer 11 (`SIGSEGV`) beendet wurde. Die vorletzte Zeile gibt ferner Aufschluss darueber, an welcher Position sich zuletzt der Programmzaehler befand. Logischerweise an jener Stelle, an der versucht wird, den Null-Pointer zu dereferenzieren.

Einen Stacktrace, also der Inhalt des Stacks (Kette an Stackframes), kann man sich mittels `bt` anzeigen lassen:

```
(gdb) bt
#0  main () at foo.c:10
(gdb)
```

Wäre unsere Datei etwas komplexer, so sieht man hier gut innerhalb welcher Funktion der Fehler auftrat. Das heisst, kopiert man den Inhalt von *main* in eine neue Funktion *test* und ruft von *main test* auf, so ergibt sich folgendes:

```
(gdb) bt
#0  test () at foo.c:11
#1  0x0000000000400519 in main () at foo.c:17
(gdb)
```

An oberster Stelle (`#0`) ist zu sehen, dass der Fehler in der Funktion *test* auftrat, die bei Zeile 11 beginnt und von Funtion *main* (Zeile 17) gerufen wurde.

Um `gdb` wieder zu verlassen, muss man `q` oder `quit` eingeben.

Ein Programm kann auch mit `gdb` ohne core-Datei gestartet werden durch `gdb ./a.out`. In der `gdb`-Konsole muss dann `run` eingeben werden.

Laufende Prozesse können ebenso mittels `gdb` untersucht werden, indem man die ProzessID an `gdb` übergibt: `gdb --pid 27927`.

Man kann dann im laufenden Prozess sich den aktuellen Stacktrace wieder mit `bt` ausgeben lassen. Mit `c` kann man den Prozess fortfahren lassen (continue). Drückt man CTRL+C in der `gdb`-Konsole, so hält der untersuchte Prozess wieder an. Mit `q`, kann sich `gdb` von dem Prozess wieder abtrennen und den Prozess normal weiter laufen lassen.

Während ein Programm gerade unterbrochen ist, kann man sich den Inhalt einer Variable per `print <expression>` ausgeben lassen, wobei `<expression>` der Name der Variable ist.

```
#include <stdio.h>

struct foo {
  int data;
  struct foo *next;
};

void test(void)
{
  int bar = 4;
  struct foo *ptr = NULL;
  if (ptr->data == bar) {
     printf("foo\n");
  }
}

int main(void) {
```

```
  test();
  return 0;
}
```

Möchte man nun die Werte der Variablen in der Funktion auslesen, bei der das Programm abstürtzt, so geschieht dies wie im Beispiel mittels `print`-Befehl:

```
(gdb) run
Starting program: /home/bordanie/Desktop/a.out

Program received signal SIGSEGV, Segmentation fault.
0x00000000004004ff in test () at foo.c:12
12   if (ptr->data == bar) {
(gdb) print ptr
$1 = (struct foo *) 0x0
(gdb) print bar
$2 = 4
(gdb) print ptr->data
Cannot access memory at address 0x0
(gdb)
```

Manchmal werden Debuginformationen durch den Compiler wegoptimiert, sodass es zu Debugzwecken sinnvoll sein kann, ein Programm mit einer kleineren Optimierungsstufe zu kompilieren, zum Beispiel `gcc -Wall -O0 -g foo.c`.

Der `gdb` ist auch in der Lage, *Breakpoints* zu setzen, das heisst, an einer definierten Stelle den Programmablauf zu unterbrechen, sodass man Variablen oder anderes untersuchen kann. Einen Breakpoint kann man per `break <argument>` setzen, wobei `<argument>` der Name einer Funktion sein kann.

```
(gdb) break test
Breakpoint 1 at 0x4004ec: file foo.c, line 10.
(gdb) run
Starting program: /home/bordanie/Desktop/a.out

Breakpoint 1, test () at foo.c:10
10   int bar = 4;
(gdb) s
11   struct foo *ptr = NULL;
(gdb) s
12   if (ptr->data == bar) {
(gdb) s

Program received signal SIGSEGV, Segmentation fault.
0x00000000004004ff in test () at foo.c:12
12   if (ptr->data == bar) {
(gdb)
```

Sobald `gdb` nun in den Breakpoint hineinspringt, bekommt man wieder die `gdb`-Konsole, in der man Variablen inspizieren kann. Mit `s` kann man zur nächsten Zeile single-steppen, mit `c` das Programm fortfahren lassen.

Neben Breakpoints gibt es auch *Watchpoints*, das heisst, der Programmablauf stoppt, sobald sich beispielsweise eine Variable geändert hat. Watchpoints können mit `watch <expression>` gesetzt werden, wobei `<expression>` der Name einer Variable sein kann. Zur Demonstration wird das Beispielprogramm leicht verändert:

```
#include <stdio.h>

struct foo {
  int data;
  struct foo *next;
};

void test(void)
{
  int ret;
  int bar = 4;
  struct foo *ptr = NULL;
  bar = 7;
  if (ptr->data == bar) {
     printf("foo\n");
  }
}

int main(void) {
  test();
  return 0;
}
```

Nun soll innerhalb der Funktion *test* die Variable `bar` auf Veränderungen hin beobachtet werden:

```
(gdb) break test
Breakpoint 1 at 0x4004ec: file foo.c, line 11.
(gdb) run
Starting program: /home/bordanie/Desktop/a.out

Breakpoint 1, test () at foo.c:11
11   int bar = 4;
(gdb) watch bar
Hardware watchpoint 2: bar
(gdb) c
Continuing.
Hardware watchpoint 2: bar

Old value = 0
New value = 7
test () at foo.c:14
14   if (ptr->data == bar) {
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400506 in test () at foo.c:14
14   if (ptr->data == bar) {
(gdb)
```

Hier ist zu sehen, dass die Programmausführung nach Zeile 13 stoppt, da sich die Variable `bar` auf 7 geändert hat. Beim alten Wert wird 0 angezeigt, a wir den Watchpoint gesetzt haben, nachdem der Breakpoint den Ablauf unterbrochen hat. Dieser wurde nach der ersten Instruktion in der Funktion *test* unterbrochen, bei welcher `bar` auf

4 gesetzt wird. Weiter zu sehen ist, dass diesmal kein Single-Stepping durchgeführt, sondern per Continue der Watchpoint getriggert wurde.

Weitere nützliche Befehle sind `up` und `down`, bei der man den Stack hoch beziehungsweise herunterlaufen kann, um Variablen zu inspizieren. Mit `next` lässt sich zur nächsten Schleifeniteration oder Subroutinenaufruf navigieren.

Möchte man sich den Assemblerinhalt der Funktion *test* ausgeben lassen, so kann dies mittels `disassemble <function-name>` durchgeführt werden:

```
(gdb) disassemble test
Dump of assembler code for function test:
0x00000000004004f0 <test+0>: cmpl   $0x5,0x0
0x00000000004004f8 <test+8>: je     0x400500 <test+16>
0x00000000004004fa <test+10>: repz retq
0x00000000004004fc <test+12>: nopl   0x0(%rax)
0x0000000000400500 <test+16>: mov    $0x40060c,%edi
0x0000000000400505 <test+21>: jmpq   0x4003e0 <puts@plt>
End of assembler dump.
(gdb)
```

Die Mnemonics sind dann entsprechend eines Prozessor-Handbuchs nachzuschlagen. Weiterführende Befehle sind durch den Befehl `help` nachzuschlagen.

8.3 Valgrind

8.4 strace

Literatur

[1] `http://gcc.gnu.org/`.

[2] `http://www.gnu.org/software/libc/`.