Systemnahe Software I
WS 2019/2020

Andreas F. Borchert
Universität Ulm

13. Februar 2020

Widmung

Ausschnitt eines Fotos von Denise Panyik-Dale, CC-BY-2.0

Diese Vorlesung ist Dennis M. Ritchie gewidmet, der am 12. Oktober 2011 verstorben ist und ohne dessen Beiträge diese Vorlesung in dieser Form nicht denkbar wäre. Dennis M. Ritchie hat nicht nur die Programmiersprache C entworfen und implementiert, sondern auch zusammen mit Ken Thompson Unix entwickelt. Mit C wurde erstmals eine höhere Programmiersprache statt Assembler für die Systemprogrammierung verwendet. Dennis Ritchie und seinen Kollegen gelang es bei Unix, die Betriebsystemsschnittstellen und die Systemwerkzeuge in einem bis dahin unbekanntem Maß zu vereinfachen. Auf sie geht die Umsetzung des wichtigen Grundsatzes zurück, dass jedes Werkzeug genau eine Aufgabe zu erfüllen habe und das möglichst gut.

Syllabus

Inhalte:

- Einführung in die Programmiersprache C
- Dynamische Speicherverwaltung
- Entwicklungswerkzeuge im Umfeld von C
- Dateisysteme
- Systemnahe Programmierung

Syllabus: Ziele

- Erwerb von Grundkenntnissen der Programmiersprache C, wobei ein besonderer Wert gelegt wird auf den Umgang mit der dynamischen Speicherverwaltung und mit den Zeigern in C. Ziel ist es auch, den versehentlichen Einbau von Sicherheitslücken zu vermeiden.
- Erlernen des Umgangs mit den klassischen Entwicklungswerkzeugen unter UNIX wie beispielsweise make.
- Verständnis der Abstraktion eines Dateisystems, einiger Implementierungen und praktische Erfahrungen mit der zugehörigen System-Schnittstelle.

Syllabus: Prüfung

- Schriftliche Prüfungen gibt es am 17. Februar und 31. März 2020. Die genaue Uhrzeit und die Räume stehen noch nicht fest. Die Prüfungen sind offen.
- Zur Teilnahme an der Prüfung ist keine Vorleistung mehr erforderlich.
- Ohne eine intensive Teilnahme an den Übungen ist jedoch eine erfolgreiche schriftliche Prüfung eher unrealistisch.
- Der Umfang beträgt 6 Leistungspunkte.
- Studiengänge:
  - **Bachelor:** CSE, Mathematik, Wirtschaftsmathematik, Wirtschaftswissenschaften, Informatik, Medieninformatik, Software Engineering, Informationssystemtechnik, Elektrotechnik.
  - **Master:** Informatik, Medieninformatik, Software Engineering, Cognitive Systems.

Syllabus: Voraussetzungen

- Grundkenntnisse in Informatik. Insbesondere sollte keine Scheu davor bestehen, etwas zu programmieren.
- Freude daran, etwas auch an einem Rechner auszuprobieren und genügend Ausdauer, dass nicht beim ersten Fehlversuch aufgegeben wird.

Syllabus: Struktur

- Jede Woche gibt es zwei Vorlesungsstunden an jedem Donnerstag von 16-18 Uhr im H12.
- Die Übungen unter Leitung von Moritz Carmesin finden am Freitag von 14-16 Uhr im H12 statt.
- Organisatorische Feinheiten werden in der ersten Übungsstunde erläutert.
- Webseite:
  `https://www.uni-ulm.de/mawi/mawi-numerik/lehrenumerik/wintersemester-20192020/vorlesung-systemnahe-software-i/`
  Oder kürzer: `https://tinyurl.com/y5gtdpej`

Syllabus: Übungen

- Wer an den Übungen teilnehmen möchte, der sollte sich über SLC für die Vorlesung registrieren.
- Eine Registrierung bei SLC ist völlig unverbindlich.
- Die Lösungen zu den Übungen werden elektronisch eingereicht. Das eröffnet uns die Möglichkeit, die Lösungen teilweise automatisiert auszuwerten und/oder als Grundlage für Feedback in den Übungen zu nehmen.
- Ebenso sollten alle über einen Shell-Zugang zu unseren Servern (wie z.B. die Theon) verfügen.
- Einzelheiten werden in der ersten Übungsstunde am Freitag, den 18. Oktober, um 14 Uhr im H12 vorgestellt.

Syllabus: Skript

- Es gibt ein Skript (entwickelt von mehreren Mitgliedern meines ehemaligen Instituts), das auf der Vorlesungswebseite zur Verfügung steht.
- Das Skript wird schrittweise entsprechend dem Vorlesungsverlauf zur Verfügung gestellt werden.
- Parallel gibt es Präsentationen (wie diese), die ebenfalls als PDF zur Verfügung gestellt werden.

Syllabus: Sprechstunden

- Sie sind eingeladen, mich jederzeit per E-Mail zu kontaktieren:
  E-Mail: andreas.borchert@uni-ulm.de
- Meine reguläre Sprechzeit ist am Mittwoch 10:00-11:30 Uhr. Zu finden bin ich in der Helmholtzstraße 20, Zimmer 1.23.
- Zu anderen Zeiten können Sie auch gerne vorbeischauen, aber es ist dann nicht immer garantiert, daß ich Zeit habe. Gegebenenfalls lohnt sich vorher ein Telefonanruf: 23572.

Syllabus: Nicht verzweifeln!

- Immer wieder kann es mal vorkommen, dass es zu scheinbar unlösbaren Problemen bei einer Übungsaufgabe kommt.
- Geben Sie dann bitte nicht auf. Nutzen Sie unsere Hilfsangebote.
- Sie können dazu gerne den Übungsleiter Moritz Carmesin oder auch mich kontaktieren.
- Schicken Sie bitte in so einem Fall alle Quellen zu und vergessen Sie nicht, eine präzise Beschreibung des Problems mitzuliefern.
- Das kann auch am Wochenende funktionieren.

Syllabus: Feedback

- Feedback ist ausdrücklich erwünscht.
- Es besteht insbesondere auch immer die Möglichkeit, auf Punkte noch einmal einzugehen, die zunächst noch nicht klar geworden sind.
- Vertiefende Fragen und Anregungen sind auch willkommen.
- Wir spulen hier nicht immer das gleiche Programm ab. Jede Vorlesung und jedes Semester verläuft anders und das hängt auch von Ihnen ab!

Syllabus: Wie geht es weiter?

- Im SS 2020 wird Objektorientierte Programmierung mit C++ angeboten werden.
- Im SS 2021 folgte der zweite Teil der Vorlesung (Systemnahe Software II).
- Ebenfalls im SS 2021 werde ich auch die Vorlesung Parallele Programmierung mit C++ anbieten.

Was ist Systemnahe Software?

- Der Begriff „System“ bezieht sich hier auf den Kern eines Betriebssystems.
- Betriebssysteme (bzw. deren Kerne) erfüllen drei Funktionen:
  - Sie greifen direkt auf die Hardware zu,
  - sie verwalten all die Hardware-Ressourcen wie beispielsweise Speicherplatz, Plattenplatz und CPU-Zeit und
  - sie bieten eine Schnittstelle für Anwendungsprogramme.
- Systemnahe Software ist Software, die direkt mit der Betriebssystems-Schnittstelle zu tun hat.

Wie sehen die Schnittstellen aus?

- Teilweise bieten die Betriebssystems-Schnittstellen (auch Systemaufrufe genannt) ein sehr hohes Abstraktions-Niveau.
- So kann beispielsweise aus der Sicht einer Anwendung eine Netzwerk-Kommunikation abgewickelt werden, ohne darüber nachzudenken, was für Netzwerk-Hardware konkret genutzt wird, wie die Pakete geroutet werden oder wann Pakete erneut zu senden sind, wenn der erste Versuch nicht geklappt hat.
- Zwar gibt es teilweise große Unterschiede bei den Schnittstellen, jedoch steht erfreulicherweise ein Standard zur Verfügung, kurz POSIX genannt oder ausführlicher IEEE Standard 1003.1-2017.
- Dieser Standard entspricht weitgehend einer gemeinsamen Schnittmenge von Unix, Linux und den BSD-Varianten. Dank Cygwin gibt es auch weitgehend eine POSIX-Schnittstelle unter Windows. Seit Windows 10 gibt es auch ein *Windows Subsystem for Linux*, das über POSIX hinaus auch die gesamte Schnittstelle des Linux-Kernels anbietet.

Warum ist C relevant?

- Durch den Erfolg von Unix erreichte C eine gewisse Monopolstellung für die systemnahe Software.
- Da POSIX (und auch die einzelnen Betriebssysteme) die Schnittstelle nur auf der Ebene von C definieren, führt an C kaum ein Weg vorbei.
- Praktisch alle anderen Sprach-Implementierungen (wie beispielsweise C++, Java, Fortran oder Ada) basieren letztenendes auf C bzw. benötigen die C-Bibliothek, die die Schnittstelle zum Betriebssystem liefert.

Warum überhaupt etwas anderes als C?

- Ursprünglich wurde systemnahe Software in Assembler geschrieben. Das ist sehr umständlich, fehleranfällig und überhaupt nicht portabel, da jede Prozessor-Architektur vollkommen anders zu programmieren ist.
- C entstand Mitte der 70er Jahre als Alternative zu Assembler. Manche bezeichnen C deswegen bis heute als „portablen Assembler“.
- C liefert Portabilität, ist aber immer noch sehr maschinennah.
- Wir verlassen mit C die gewohnte „heile Welt“ von Java (oder anderer ähnlicher moderner Programmiersprachen).
- C setzt maschinennahes Denken voraus und bietet viele Fallstricke, die wir in der „heilen Welt“ nicht kennen.
- Insofern wird heute C bevorzugt nur im systemnahen Bereich eingesetzt, wo diese Freiheiten benötigt werden.

Wie funktionieren Systemaufrufe?

- Programme laufen auf modernen Betriebssystemen in ihrer eigenen virtuellen Welt ab, d.h. sie sehen in ihrem Adressraum weder das Betriebssystem noch die anderen parallel laufenden Programme.
- Die virtuelle Welt wird nur durch besondere Ereignisse verlassen, wenn z.B. durch 0 geteilt wird, ein Nullzeiger dereferenziert wird, die Uhr sagt, dass ein anderer Prozess mal an der Reihe ist, sich die Platte meldet, weil ein gewünschter Datenblock endlich da ist, irgendeine Taste auf der Tastatur gedrückt wurde oder ...
- ... ein Programm mit dem Betriebssytem kommunizieren möchte.
- All diese Ereignisse unterbrechen die Ausführung eines Benutzerprogramms und übergeben die Kontrolle dem Betriebssystem-Kern, der dann zu untersuchen hat, was die Unterbrechung auslöste.
- Im Falle eines Systemaufrufs werden die Parameter aus der Welt des Benutzerprogramms mühsam herausgeholt, der Aufruf bearbeitet und die Resultate in die Benutzer-Welt überführt.
- In Wirklichkeit ist das noch viel komplizierter ...

Wie wird eine Unterbrechung initiiert?

- Für absichtliche Unterbrechungen gibt es spezielle Maschinen-Instruktionen.
- Diese gehören nicht zum Vokabular eines C-Compilers, so dass in jeder C-Bibliothek die Systemaufrufe in Assembler (bzw. mit in C eingebetten Assembler-Instruktionen) geschrieben sind.
- Die Aufrufsyntax in C ist portabel, die jeweilige Implementierung ist es nicht, da Assembler-Programme von der jeweiligen Plattform (Maschinenarchitektur und Betriebssystem) abhängig sind.

Wie sehen Systemaufrufe konkret aus?

hello.s

```
/*
   Hello world demo in Assembler
   for the SPARCv8/Solaris platform
*/
        .section ".text"
        .globl  _start
_start:
/* write(1, msg, 13); */
        or      %g0,4,%g1
        or      %g0,1,%o0
        sethi   %hi(msg),%o1
        add     %o1,%lo(msg),%o1
        or      %g0,13,%o2
        ta      8
/* exit(0) */
        or      %g0,1,%g1
        or      %g0,0,%o0
        ta      8
msg:    .ascii  "Hello world!\012"
```

Wie sehen Systemaufrufe konkret aus?

- Das Beispiel wurde für die SPARC-Architektur geschrieben.
- Das Assembler-Programm besteht aus 9 Instruktionen, die jeweils 4 Bytes benötigen und 13 Bytes Text.
- `%g1`, `%o0`, `%o1` und `%o2` sind alles sogenannte Register, die im 32-Bit-Modus jeweils 32 Bit aufnehmen können.
- *%g0* ist ein spezielles Register, das immer den Wert 0 hat.
- Instruktionen haben bei der SPARC-Architektur normalerweise drei Operanden, wobei der dritte Operand das Ziel ist. Beispiel: `or %g0,4,%g1`
  Das ist eine binäre Oder-Operation mit *%g0* (also dem Wert 0) und der Zahl 4, dessen Resultat in *%g1* abgelegt wird. Kurz gefasst wird damit dem Register *%g1* der Wert 4 zugewiesen.

Wie sehen Systemaufrufe konkret aus?

- Die spezielle Instruktion *ta* (*trap always*) unterbricht die Programmausführung, bis der Prozess vom Betriebssystem wieder zum Leben erweckt wird.
- Die Parameter des Systemaufrufs werden bei der SPARC/Solaris-Plattform in den Registern *%o0* bis *%o5* abgelegt (bis zu 6 Parameter, die allerdings auf irgendwelche Speicherflächen mit mehr Parametern verweisen können).
- Im Register *%g1* wird eine Nummer abgelegt, die den Systemaufruf selektiert. So steht beispielsweise die 1 für *exit()* und 4 für *write()*. (Die Nummern finden sich unter Solaris in */usr/include/sys/syscall.h*.)
- Die Nummer 8, die bei *ta* angegeben wird, dient als Index in die Trap-Tabelle ...

Was passiert nach einem Systemaufruf?

usr/src/uts/sun4u/ml/trap_table.s

```
trap_table:
    /* hardware traps */
    NOT;                        /* 000  reserved */
    RED;                        /* 001  power on reset */
    RED;                        /* 002  watchdog reset */
    RED;                        /* 003  externally initiated reset */
/* ... */
    /* user traps */
    GOTO(syscall_trap_4x);      /* 100  old system call */
    TRAP(T_BREAKPOINT);         /* 101  user breakpoint */
    TRAP(T_DIV0);               /* 102  user divide by zero */
    FLUSHW();                   /* 103  flush windows */
    GOTO(.clean_windows);       /* 104  clean windows */
    BAD;                        /* 105  range check ?? */
    GOTO(.fix_alignment);       /* 106  do unaligned references */
    BAD;                        /* 107  unused */
    SYSCALL(syscall_trap32)     /* 108  ILP32 system call on LP64 */
/* ... */
```

- (Der Programmtext stammt aus den OpenSolaris-Quellen, ursprünglich von `http://www.opensolaris.org/`, jetzt verfügbar in revidierter Form über `http://src.illumos.org`).

Was passiert nach einem Systemaufruf?

usr/src/uts/sun4u/ml/trap_table.s

```
#define SYSCALL(which)                  \
        TT_TRACE(trace_gen)             ;\
        set     (which), %g1            ;\
        ba,pt   %xcc, sys_trap          ;\
        sub     %g0, 1, %g4             ;\
        .align  32
```

- Zunächst werden alle Register gesichert und es findet ein Wechsel in den privilegierten Prozessor-Modus statt.
- Zu jeder Unterbrechungsart gibt es eine Nummer, wobei Unterbrechungen durch Benutzerprogramme von Hardware-Unterbrechungen unterschieden werden. Aus *ta 8* wird die Nummer $256 + 8 = 0\text{x}108$.
- Zu jedem der 512 verschiedenen Unterbrechungsmöglichkeiten sind in der Trap-Tabelle 32 Bytes Code vorgesehen, die den Trap behandeln, indem sie typischerweise eine entsprechende Routine aufrufen.

Was passiert nach einem Systemaufruf?

usr/src/uts/sparc/v9/ml/syscall_trap.s

```
    ENTRY_NP(syscall_trap32)
    ldx      [THREAD_REG + T_CPU], %g1     ! get cpu pointer
    mov      %o7, %l0                      ! save return addr
/* ... */
    lduw     [%l1 + G1_OFF + 4], %g1       ! get 32-bit code
    set      sysent32, %g3                 ! load address of vector table
    cmp      %g1, NSYSCALL                 ! check range
    sth      %g1, [THREAD_REG + T_SYSNUM] ! save syscall code
    bgeu,pn %ncc, _syscall_ill32
      sll    %g1, SYSENT_SHIFT, %g4        ! delay - get index
    add      %g3, %g4, %g5                 ! g5 = addr of sysentry
    ldx      [%g5 + SY_CALLC], %g3         ! load system call handler
/* ... */
    call     %g3                           ! call system call handler
    nop
/* ... */
    jmp      %l0 + 8
     nop
```

- Danach werden die Parameter so kopiert, dass sie als Parameter einer C-Funktion übergeben werden können (nicht dargestellt) und dann wird passend zur Systemaufrufsnummer die entsprechende Funktion aus einer Tabelle ausgewählt.

Was passiert nach einem Systemaufruf?

usr/src/uts/common/os/sysent.c

```
struct sysent sysent[NSYSCALL] =
{
/* ONC_PLUS EXTRACT END */
    /*  0 */ IF_LP64(
                    SYSENT_NOSYS(),
                    SYSENT_C("indir",       indir,          1)),
    /*  1 */ SYSENT_CI("exit",              rexit,          1),
    /*  2 */ SYSENT_2CI("forkall",          forkall,        0),
    /*  3 */ SYSENT_CL("read",              read,           3),
    /*  4 */ SYSENT_CL("write",             write,          3),
    /*  5 */ SYSENT_CI("open",              open,           3),
```

- In der *sysent*-Tabelle finden sich alle Systemaufrufe zusammen mit einigen Infos zur Anzahl der Parameter und die Art der Rückgabewerte. Unter Solaris hat diese Tabelle inzwischen 256 Einträge.

Was passiert nach einem Systemaufruf?

usr/src/uts/common/syscall/rw.c

```
ssize_t write(int fdes, void *cbuf, size_t count) {
        struct uio auio;
        struct iovec aiov;
/* ... */
        if ((cnt = (ssize_t)count) < 0)
                return (set_errno(EINVAL));
        if ((fp = getf(fdes)) == NULL)
                return (set_errno(EBADF));
        if (((fflag = fp->f_flag) & FWRITE) == 0) {
                error = EBADF;
                goto out;
        }
/* ... */
        aiov.iov_base = cbuf;
        aiov.iov_len = cnt;
/* ... */
        auio.uio_loffset = fileoff;
        auio.uio_iov = &aiov;
/* ... */
        error = VOP_WRITE(vp, &auio, ioflag, fp->f_cred, NULL);
        cnt -= auio.uio_resid;
/* ... */
out:
/* ... */
        if (error)
                return (set_errno(error));
        return (cnt);
}
```

Weiterleitung der Schreib-Operation

usr/src/uts/common/sys/vnode.h

```
#define VOP_WRITE(vp, uiop, iof, cr, ct) \
        fop_write(vp, uiop, iof, cr, ct)
```

usr/src/uts/common/fs/vnode.c

```
int
fop_write(vnode_t *vp, uio_t *uiop, int ioflag, cred_t *cr,
        caller_context_t *ct)
{
        int     err;
        ssize_t resid_start = uiop->uio_resid;

        VOPXID_MAP_CR(vp, cr);

        err = (*(vp)->v_op->vop_write)(vp, uiop, ioflag, cr, ct);
        VOPSTATS_UPDATE_IO(vp, write,
            write_bytes, (resid_start - uiop->uio_resid));
        return (err);
}
```

- Hier wird ein Funktionszeiger aufgerufen – das entspricht einem dynamischen Methodenaufruf.

Wie sehen Systemaufrufe unter Linux aus?

hello-x86.s

```
/*
 * Hello world demo in Assembler
 * for the IA-64/Linux platform
 */
        .text
        .globl  _start
_start:
/* write(1, msg, 13) */
        movl    $13, %edx
        movl    $msg, %esi
        movl    $1, %edi
        movl    $1, %eax
        syscall

/* exit(0) */
        movl    $0, %edi
        movl    $60, %eax
        syscall

        .data
msg:
        .string "Hello world!\n"
```

Wie sehen Systemaufrufe unter Linux aus?

- Auf der IA-64/Linux-Plattform werden die Parameter ebenfalls über Register übermittelt. Hier die ersten drei Parameter über *%edi*, *%esi* und *%edx*.
- Die Systemaufrufnummer wird in *%eax* abgelegt. Linux verwendet jedoch seine eigene Numerierung mit 1 für *write()* und 60 für *edit()*.
- Der Trap für einen Systemaufruf wird auf der IA-64-Plattform mit der Instruktion *syscall* initiiert.

Übersetzung eines Assembler-Programms

```
heim$ as -o hello-x86.o hello-x86.s
heim$ ld -o hello-x86 hello-x86.o
heim$ ./hello-x86
Hello world!
heim$ strace ./hello-x86 >/dev/null
execve("./hello-x86", ["./hello-x86"], [/* 46 vars */]) = 0
write(1, "Hello world!\n", 13)          = 13
_exit(0)                                = ?
+++ exited with 0 +++
heim$
```

- Mit *as* wird der Assemblertext in Maschinencode überführt. Das Resultat ist eine sogenannte Objektdatei mit der Endung „.o“.
- Objekt-Dateien können mit dem Werkzeug *ld* (*loader* oder *linkage editor*) zu einem ausführbaren Programm zusammengefügt werden, ggf. unter Verwendung von Bibliotheken.
- Mit dem Werkzeug *strace* können die Systemaufrufe eines Prozesses verfolgt werden (unter Solaris ist hier *truss* zu verwenden).

Zusammenfassung

- Die Schnittstelle zwischen Anwendungen und dem Betriebssystems-Kern besteht aus über 200 einzelnen Funktionen, die zu einem großen Teil standardisiert sind.
- Systemaufrufe sind sehr viel teurer als reguläre Funktionsaufrufe. Das liegt an dem Mechanismus der Unterbrechungsbehandlung, dem notwendigen Kontextwechsel, der Parameterschaufelei und auch der asynchronen Natur vieler Funktionen (wie beispielsweise beim I/O).
- Deswegen ist es wichtig, auf der Anwendungsseite Bibliotheken zu entwickeln, die die Zahl der Systemaufrufe bzw. deren Aufwand minimieren.
- Verbunden mit den einzelnen Systemaufrufen sind viele Abstraktionen und Objekte, die wir uns im Laufe der Vorlesung genauer ansehen werden wie etwa das Dateisystem, die Prozesse, die Signalbehandlung (Unterbrechungen auf der Benutzerseite) die Interprozess-Kommunikation und die allgemeine Netzwerk-Kommunikation.

- Um einen raschen Start in den praktischen Teil zu ermöglichen, wird C zunächst etwas oberflächlich mit einigen Beispielen vorgestellt.
- Später werden dann die Feinheiten vertieft vorgestellt.
- Im Vergleich zu Java gibt es in C keine Klassen. Stattdessen sind alle Konstrukte recht nah an den gängigen Prozessorarchitekturen, die das ebenfalls nicht kennen.
- In C gibt es statt Klassen und Methoden nur globale Funktionen, die Parameter erhalten und einen Wert zurückliefern. Da die Funktionen sich nicht implizit auf ein Objekt beziehen, sind sie am ehesten vergleichbar mit den statischen Methoden in Java.
- Jedes C-Programm benötigt ähnlich wie in Java eine *main*-Funktion. (Die Parameter für die Kommandozeilenargumente können aber weggelassen werden, wenn diese nicht benötigt werden.)

Ein erstes C-Programm

hallo.c

```
main() {
   /* puts: Ausgabe einer Zeichenkette nach stdout */
   puts("Hallo zusammen!");
}
```

- Dieses Programm gibt den gezeigten Text mit einem abschließenden Zeilentrenner aus, – analog zu *System.out.println*.
- Im Unterschied zu Java muss wirklich eine Zeichenkette angegeben werden. Andere Datentypen werden hier nicht implizit über eine *toString*-Methode in Zeichenketten verwandelt.
- Zeichenketten werden (wie in den Assembler-Programmen *hello.s* und *hello-x86.s*) durch Zeiger auf eine nullbyte-terminierte Sequenz von Zeichen im Speicher repräsentiert.

Übersetzung des ersten C-Programms

```
theon$ gcc -fmessage-length=70 -Wall hallo.c
hallo.c:1:1: warning: return type
   defaults to 'int' [-Wimplicit-int]
 main() {
 ^~~~
hallo.c: In function 'main':
hallo.c:3:4: warning: implicit
   declaration of function 'puts'
   [-Wimplicit-function-declaration]
    puts("Hallo zusammen!");
    ^~~~
theon$ a.out
Hallo zusammen!
theon$
```

- Der *gcc* ist der *GNU-C-Compiler*, mit dem wir unsere Programme übersetzen.
- Ist kein Name für das zu generierende ausführbare Programm angegeben, so wird dieses *a.out* genannt.
- Die Option *-Wall* bedeutet, dass alle Warnungen ausgegeben werden sollen.

Wahl des zugrundeliegenden Standards

- Die Programmiersprache C ist standardisiert (ISO 9899). Den Standard gibt es in den Versionen von 1990, 1999, 2011 und 2018.
- Voreinstellungsgemäß geht *gcc* ab Version 5.1 von *C11* aus, bei älteren Versionen (etwa 4.9) ist noch *C90* die Voreinstellung. Es ist auch möglich, mit der Option „*-std=c11*“ den aktuellen Standard von 2011 explizit auszuwählen.
- Statt „*-std=c11*“ ist auch „*-std=gnu11*“ möglich – dann stehen auch verschiedene Erweiterungen (u.a. der POSIX-Standard) zur Verfügung, die nicht über C99 oder C11 vorgegeben sind.
- Für die Übungen empfiehlt sich grundsätzlich die Wahl von *gnu11*, wobei letzteres erst ab GCC 4.7.x unterstützt wird. Auf unseren Maschinen haben wir folgende Versionen: 7.3.0 (Solaris/Intel, Theon), 5.2.0 (Solaris/Intel, Thales), 6.3.0 (Debian, Pool in E.44) oder 4.8.0 (Solaris/SPARC, Theseus).
- Mit dem Aufruf von *gcc --version* lässt sich die Version des *gcc*-Übersetzers feststellen.

hallo1.c

```
#include <stdio.h> /* Standard-I/O-Bibliothek einbinden */

int main() {
   /* puts: Ausgabe eines Strings nach stdout */
   puts("Hallo zusammen!");
}
```

- Da die Ausgabefunktion *puts()* nicht bekannt war, hat der Übersetzer „geraten“. Nun ist diese Funktion durch das Einbinden der Deklarationen der Standard-I/O-Bibliothek (siehe **#include** <stdio.h>) bekannt.
- Der *Typ des Rückgabewertes* der *main()*-Funktion ist nun als **int** (Integer) angegeben (der Übersetzer hat dies zuvor implizit angenommen.)

```
theon$ gcc -Wall -o hallo1 hallo1.c
theon$ ./hallo1
Hallo zusammen!
theon$
```

- Mit der Option „*-o*“ kann der Name des Endprodukts beim Aufruf des *gcc* spezifiziert werden.
- Anders als bei Java ist das Endprodukt selbständig ausführbar, da es in Maschinensprache übersetzt wurde.
- Das bedeutet jedoch auch, dass das Endprodukt nicht portabel ist, d.h. bei anderen Prozessorarchitekturen oder Betriebssystemen muss das Programm erneut übersetzt werden.
- Das gilt auch auf unseren Maschinen, die in drei Gruppen fallen: Solaris/Intel, Solaris/SPARC und Debian/Intel.

Erzeugter Assemblertext: SPARC


```
        .file   "hallo1.c"
        .section        ".rodata"
        .align 8
.LLC0:
        .asciz  "Hallo zusammen!"
        .section        ".text"
        .align 4
        .global main
        .type   main, #function
        .proc   04
main:
        save    %sp, -96, %sp
        sethi   %hi(.LLC0), %g1
        or      %g1, %lo(.LLC0), %o0
        call    puts, 0
         nop
        mov     0, %g1
        mov     %g1, %i0
        return  %i7+8
         nop
        .size   main, .-main
        .ident  "GCC: (GNU) 4.8.0"
```

- Resultat von „*gcc -S hallo.c*“ auf einer SPARC-Plattform.

Erzeugter Assemblertext: Intel/x86


```
        .file   "hallo1.c"
        .section        .rodata
.LC0:
        .string "Hallo zusammen!"
        .text
        .globl  main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        andl    $-16, %esp
        subl    $16, %esp
        movl    $.LC0, (%esp)
        call    puts
        movl    $0, %eax
        leave
        ret
        .size   main, .-main
        .ident  "GCC: (GNU) 4.7.1"
```

- Resultat von „*gcc -S hallo.c*“ auf einer Intel/x86-Plattform.

Berechnung von Quadratzahlen

quadrate.c

```
#include <stdio.h>

const int MAX = 20;    /* globale Integer-Konstante */

int main() {
   puts("Zahl | Quadratzahl");
   puts("-----+------------");
   for (int n = 1; n <= MAX; n++) {
      printf("%4d | %7d\n", n, n * n); /* formatierte Ausgabe */
   }
}
```

- Dieses Programm gibt die ersten 20 natürlichen Zahlen und ihre zugehörigen Quadratzahlen aus.
- Variablendeklarationen können außerhalb von Funktionen stattfinden. Dann gibt es die Variablen genau einmal und ihre Lebensdauer erstreckt sich über die gesamte Programmlaufzeit.

quadrate.c

```
printf("%4d | %7d\n", n, n * n); /* formatierte Ausgabe */
```

- Formatierte Ausgaben erfolgen in C mit Hilfe von *printf*.
- Die erste Zeichenkette kann mehrere Platzhalter enthalten, die jeweils mit „%“ beginnen und die Formatierung eines auszugebenden Werts und den Typ des entsprechenden Arguments spezifizieren.
- „%4d“ bedeutet hier, dass ein Wert des Typs **int** auf einer Breite von (mindestens) vier Zeichen dezimal auszugeben ist.

quadrate.c

```
for (int n = 1; n <= MAX; n++) {
   printf("%4d | %7d\n", n, n * n); /* formatierte Ausgabe */
}
```

- Wie in Java kann eine Schleifenvariable im Initialisierungsteil einer **for**-Schleife deklariert und initialisiert werden.
- Dies ist im Normalfall vorzuziehen.
- Gelegentlich finden sich noch Deklarationen von Schleifenvariablen außerhalb der **for**-Schleife, weil dies frühere C-Versionen nicht anders vorsahen.

Euklidischer Algorithmus

euklid.c

```
#include <stdio.h>

int main() {
   printf("Geben Sie zwei positive ganze Zahlen ein: ");
   /* das Resultat von scanf ist die
      Anzahl der eingelesenen Zahlen
   */
   int x, y;
   if (scanf("%d %d", &x, &y) != 2) {  /* &-Operator konstruiert Zeiger */
      return 1;      /* Exit-Status ungleich 0 => Fehler */
   }

   int x0 = x;
   int y0 = y;

   while (x != y) {
      if (x > y) {
         x = x - y;
      } else {
         y = y - x;
      }
   }

   printf("ggT(%d, %d) = %d\n", x0, y0, x);
}
```

euklid.c

```
if (scanf("%d %d", &x, &y) != 2) {
   /* Fehlerbehandlung */
}
```

- Die Programmiersprache C kennt nur die *Werteparameter-Übergabe* (*call by value*).
- Daher stehen auch bei *scanf()* nicht direkt die Variablen x und y als Argumente, weil dann *scanf()* nur die Kopien der beiden Variablen zur Verfügung stehen würden.
- Mit dem Operator „&“ wird hier jeweils ein *Zeiger* auf die folgende Variable erzeugt. Der Wert eines Zeigers ist die *virtuelle Adresse* der Variablen, auf die er zeigt.
- Daher wird in diesem Zusammenhang der unäre Operator „&“ auch als *Adressoperator* bezeichnet.

euklid.c

```
if (scanf("%d %d", &x, &y) != 2) {
   /* Fehlerbehandlung */
}
```

- Die Programmiersprache C kennt weder eine Überladung von Operatoren oder Funktionen.
- Entsprechend gibt es nur eine einzige Instanz von *scanf()*, die in geeigneter Weise „erraten“ muss, welche Datentypen sich hinter den Zeigern verbergen.
- Das erfolgt (analog zu *printf*) über Platzhalter. Dabei steht „%d“ für das Einlesen einer ganzen Zahl in Dezimaldarstellung in eine Variable des Typs **int**.
- Variablen des Typs **float** (einfache Genauigkeit) können mit „%f“ eingelesen werden, **double** (doppelte Genauigkeit) mit „%lf“.

euklid.c

```
if (scanf("%d %d", &x, &y) != 2) {
   /* Fehlerbehandlung */
}
```

- Der Rückgabewert von *scanf* ist die Zahl der erfolgreich eingelesenen Werte.
- Deswegen wird hier das Resultat mit der 2 verglichen.
- Es sollte immer überprüft werden, ob Einlesefehler vorliegen. Normalerweise empfiehlt sich dann eine Fehlermeldung und ein Ausstieg mit *exit*(1) bzw. innerhalb von *main* mit **return** 1.
- Ausnahmenbehandlungen (*exception handling*) gibt es in C nicht. Stattdessen geben alle Ein- und Ausgabefunktionen (in sehr unterschiedlicher Form) den Erfolgsstatus zurück.

$$
\begin{array}{rcl}
\langle\text{translation-unit}\rangle & \longrightarrow & \langle\text{external-declaration}\rangle \\
 & \longrightarrow & \langle\text{translation-unit}\rangle\ \langle\text{external-declaration}\rangle \\
\langle\text{external-declaration}\rangle & \longrightarrow & \langle\text{declaration}\rangle \\
 & \longrightarrow & \langle\text{function-definition}\rangle
\end{array}
$$

- Eine Übersetzungseinheit (*translation unit*) in C ist eine Folge von globalen (d.h. normalerweise extern sichtbaren) *Vereinbarungen*, zu denen Funktionsdefinitionen, Funktionsdeklarationen, Typ-Vereinbarungen und Variablenvereinbarungen gehören.
- Deklarationen und Definitionen sind Vereinbarungen. Eine Definition führt zum Anlegen einer Variablen oder Funktion. Eine Deklaration verbindet einen Variablen- oder Funktionsnamen nur mit einem Typ bzw. Signatur, ohne diese anzulegen.

Deklarationen vs. Definitionen


```
int i = 7; // Variablendefinition
extern int j; // Variablendeklaration
double sin(double x); // Funktionsdeklaration
double f(double x) { // Funktionsdefinition
   return 2 * x;
}
```

- Deklarationen und Definitionen sind Vereinbarungen. Eine Definition führt zum Anlegen einer Variablen oder Funktion. Eine Deklaration verbindet einen Variablen- oder Funktionsnamen nur mit einem Typ bzw. Signatur, ohne diese anzulegen.
- Header-Dateien bestehen typischerweise bei C nur aus Deklarationen von Objekten, die in den entsprechenden Bibliotheken definiert sind.
- Global deklarierte Objekte mit uneingeschränkter Sichtbarkeit füllen mit ihrem Namen den globalen Namensraum.
- Die Mehrfachdefinition eines Namens führt normalerweise zu Fehlern beim Zusammenbau eines Programms. Genauso eine Deklaration ohne zugehörige Definition.

$$
\begin{array}{rcl}
\langle\text{declaration}\rangle & \longrightarrow & \langle\text{declaration-specifiers}\rangle \\
 & & \langle\text{init-declarator-list}\rangle \text{ „;“} \\
 & \longrightarrow & \langle\text{static-assert-declaration}\rangle \\
\langle\text{declaration-specifiers}\rangle & \longrightarrow & \langle\text{declaration-specifier}\rangle \\
 & & [\ \langle\text{declaration-specifiers}\rangle\ ] \\
\langle\text{declaration-specifier}\rangle & \longrightarrow & \langle\text{storage-class-specifier}\rangle \\
 & \longrightarrow & \langle\text{type-specifier}\rangle \\
 & \longrightarrow & \langle\text{type-qualifier}\rangle \\
 & \longrightarrow & \langle\text{function-specifier}\rangle \\
 & \longrightarrow & \langle\text{alignment-specifier}\rangle
\end{array}
$$

- Bei einer einfachen Variablendeklaration **int** $i = 7$; ist „**int**“ der ⟨type-specifier⟩, während „$i = 7$“ zur ⟨init-declarator-list⟩ gehört.

$$
\begin{array}{rcl}
\langle\text{function-definition}\rangle & \longrightarrow & \langle\text{declaration-specifiers}\rangle\ \langle\text{declarator}\rangle \\
 & & [\ \textit{declaration} - \textit{list}] \\
 & & \langle\text{compound-statement}\rangle \\
\langle\text{compound-statement}\rangle & \longrightarrow & \text{„}\{\text{“}\ \ [\ \langle\text{block-item-list}\rangle\ ]\ \ \text{„}\}\text{“} \\
\langle\text{block-item-list}\rangle & \longrightarrow & \langle\text{block-item}\rangle \\
 & \longrightarrow & \langle\text{block-item-list}\rangle\ \langle\text{block-item}\rangle \\
\langle\text{block-item}\rangle & \longrightarrow & \langle\text{declaration}\rangle \\
 & \longrightarrow & \langle\text{statement}\rangle
\end{array}
$$

- Bei „**double** *square*(**double** *x*){ **return** *x*∗*x*; }“ wird das erste „**double**“ unter die ⟨declaration-specifiers⟩ subsummiert, „*square*(**double** *x*)“ bildet den ⟨declarator⟩ und „{ **return** *x*∗*x*; }“ das ⟨compound-statement⟩.

Beispiel für eine Funktion

euklid2.c

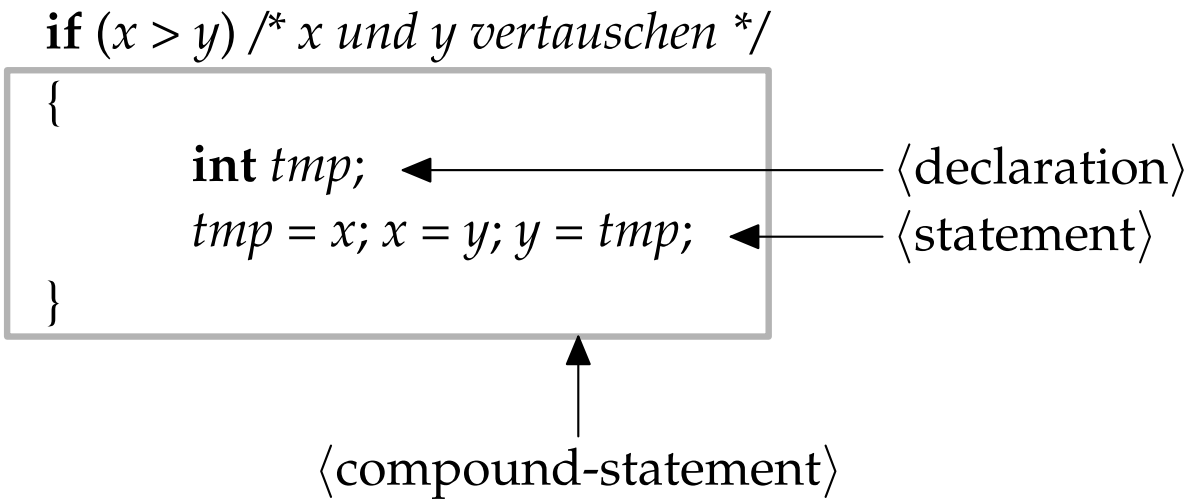
```
#include <stdio.h>

int ggt(int a, int b) {
   while (a != b) {
      if (a > b) {
         a -= b;
      } else {
         b -= a;
      }
   }
   return a;
}
int main() {
   printf("Geben Sie zwei positive ganze Zahlen ein: ");
   int x, y;
   if (scanf("%d %d", &x, &y) != 2 || x <= 0 || y <= 0) {
      return 1;
   }
   printf("ggT(%d, %d) = %d\n", x, y, ggt(x, y));
}
```

Einige Regeln zu Funktionen:

- Es gilt „*declare before use*“, d.h. Funktionen sind zu deklarieren, bevor sie benutzt werden.
- Wenn eine unbekannte Funktion benutzt wird, erfolgt eine implizite Deklaration mit **int** für den Rückgabewert und eine Parameterübergabe nach festgelegten Konventionen. Dies ist grundsätzlich zu vermeiden.
- Deswegen ist darauf zu achten, dass bei jeder Bibliotheksfunktion, die zugehörige Header-Datei mit **#include** zuvor eingebunden worden ist.
- Die Parameterübergabe erfolgt immer per *call by value*.
- Bei Arrays liegen implizit Zeiger vor, so dass bei einer Parameterübergabe nur ein Zeiger per *call by value* übergeben wird und nicht etwa der Array-Inhalt kopiert wird.

Anweisungen in C

$$
\begin{aligned}
\langle \text{statement} \rangle \quad &\longrightarrow \quad \langle \text{expression-statement} \rangle \\
&\longrightarrow \quad \langle \text{labeled-statement} \rangle \\
&\longrightarrow \quad \langle \text{compound-statement} \rangle \\
&\longrightarrow \quad \langle \text{conditional-statement} \rangle \\
&\longrightarrow \quad \langle \text{iterative-statement} \rangle \\
&\longrightarrow \quad \langle \text{switch-statement} \rangle \\
&\longrightarrow \quad \langle \text{break-statement} \rangle \\
&\longrightarrow \quad \langle \text{continue-statement} \rangle \\
&\longrightarrow \quad \langle \text{return-statement} \rangle \\
&\longrightarrow \quad \langle \text{goto-statement} \rangle \\
&\longrightarrow \quad \langle \text{null-statement} \rangle
\end{aligned}
$$



- Mit **int** *tmp*; wird eine lokale Variable mit dem Datentyp **int** deklariert.
- Die Sichtbarkeit von *tmp* erstreckt sich auf den umrandeten Anweisungsblock (⟨compound-statement⟩).
- Lokale Variablen werden dann erzeugt, wenn der sie umgebende Block ausgeführt wird. Ihre Existenz endet mit dem Erreichen des Blockendes. Bei Rekursion kann eine lokale Variable mehrfach instantiiert werden.

Initialisierung lokaler Variablen

varinit.c

```
#include <stdio.h>

int main() {
   int i; /* left uninitialized */
   int j = i; /* effect is undefined, yet compilers accept it */
   printf("%d\n", j);
}
```

- In Java dürfen lokale Variablen solange nicht verwendet werden, bis zweifelsfrei sichergestellt ist, dass sie ordentlich initialisiert sind. Dies wird bei Java vom Übersetzer zur Übersetzzeit überprüft.
- In C geschieht dies nicht. Der Wert einer uninitialisierten lokalen Variable ist undefiniert.
- Um das Problem zu vermeiden, sollten lokale Variablen entweder bei der Deklaration oder der darauffolgenden Anweisung initialisiert werden.
- Der *gcc* warnt bei eingeschalteter Optimierung und bei neueren Versionen auch ohne Optimierung. Viele Übersetzer tun dies jedoch nicht.

Bei etwas älteren gcc-Übersetzern:

```
clonard$ gcc --version | sed 1q
gcc (GCC) 4.1.1
clonard$ gcc -std=gnu99 -Wall -o varinit varinit.c && ./varinit
4
clonard$ gcc -O2 -std=gnu99 -Wall -o varinit varinit.c && ./varinit
varinit.c: In function 'main':
varinit.c:5: warning: 'i' is used uninitialized in this function
7168
clonard$
```

Bei einer etwas neueren gcc-Version:

```
theon$ gcc -std=gnu11 -fmessage-length=82 -Wall -o varinit varinit.c
varinit.c: In function 'main':
varinit.c:5:8: warning: 'i' is used
   uninitialized in this function [-Wuninitialized]
    int j = i; /* effect is undefined, yet compilers accept it */
        ^
theon$ varinit
-32513
theon$
```

Kommentare

- Kommentare beginnen mit „/*“, enden mit „*/“, und dürfen nicht geschachtelt werden.
- Alternativ kann seit C99 in Anlehnung an C++ ein Kommentar auch mit „//“ begonnen werden, der sich bis zum Zeilenende erstreckt.
- Kommentarzeichen werden innerhalb von Literalen nicht als solche erkannt.

auto	**else**	**long**	**switch**	**_Atomic**
break	**enum**	**register**	**typedef**	**_Bool**
case	**extern**	**restrict**	**union**	**_Complex**
char	**float**	**return**	**unsigned**	**_Generic**
const	**for**	**short**	**void**	**_Imaginary**
continue	**goto**	**signed**	**volatile**	**_Noreturn**
default	**if**	**sizeof**	**while**	**_Static_assert**
do	**inline**	**static**	**_Alignas**	**_Thread_local**
double	**int**	**struct**	**_Alignof**	

- Im übrigen sind alle Namen, die mit einem Unterstrich beginnen und von einem weiteren Unterstrich oder einem Großbuchstaben gefolgt werden, reserviert. Beispiel: __*func*__.
- Im übrigen sind diverse Namen aus der Standardbibliothek reserviert.
- Da C mit einem globalen Namensraum arbeitet, sind Namenskonflikte ein Problem.

Schlüsselworte mit Unterstrich

- C ist eine über Jahrzehnte gewachsene Programmiersprache.
- So gab es ursprünglich keinen Datentyp *bool*. Stattdessen wurde einfach **int** verwendet. Viele Programmierer definierten sich gerne selbst aus Gründen der Lesbarkeit einen Typ *bool* mitsamt Definitionen für *true* und *false*.
- Als in C90 ein entsprechender Datentyp eingeführt wurde, sollten aus Gründen der Aufwärtskompatibilität Programme mit selbstdefinierten *bool*, *true* und *false* nicht beeinträchtigt werden.
- Deswegen wurde für den Boolean-Datentyp das Schlüsselwort **_Bool** vergeben. Dieses wird aber typischerweise nicht direkt verwendet. Stattdessen können mit Hilfe von **#include** <stdbool.h> darauf basierende Definitionen für *bool*, *true* und *false* bezogen werden.
- Analog gibt es <*stdalign.h*>, <*complex.h*>, <*assert.h*> und <*threads.h*>.

sum.c

```
#include <stdio.h>
#include <stdbool.h>
int main() { bool EOF = false; int sum = 0;
   do {
      int value; EOF = (scanf("%d", &value) != 1);
      if (!EOF) sum += value;
   } while (!EOF); printf("sum = %d\n", sum);
}
```

- Dieses nicht sehr elegant geschriebene Programm lässt sich nicht übersetzen und die Fehlermeldungen geben Rätsel auf:

```
clonmel$ gcc -Wall -o sum sum.c
In file included from /usr/include/stdio.h:66:0,
                 from sum.c:1:
sum.c: In function 'main':
sum.c:3:19: error: expected identifier or '(' before '-' token
 int main() { bool EOF = false; int sum = 0;
                   ^
sum.c:5:22: error: lvalue required as left operand of assignment
       int value; EOF = (scanf("%d", &value) != 1);
                      ^
clonmel$
```

```
clonmel$ gcc -E sum.c | sed -n '/^#/d; /main/,$p'
int main() {
            _Bool (-1)
                      =
                        0
                              ; int sum = 0;
   do {
      int value;
                 (-1)
                     = (scanf("%d", &value) != 1);
      if (!
          (-1)
             ) sum += value;
   } while (!
            (-1)
               ); printf("sum = %d\n", sum);
}
clonmel$
```

- Dieses Rätsel löst sich erst, wenn das Ergebnis des Präprozessors betrachtet wird. Im gesamten Programmtext wurde offenbar *EOF* durch (-1) ersetzt.

```
clonmel$ find /usr/include -type f |
> xargs perl -ne 'print "$ARGV:$_" if /^#\s*define\s*EOF\b/'
/usr/include/iso/stdio_iso.h:#define    EOF     (-1)
clonmel$
```

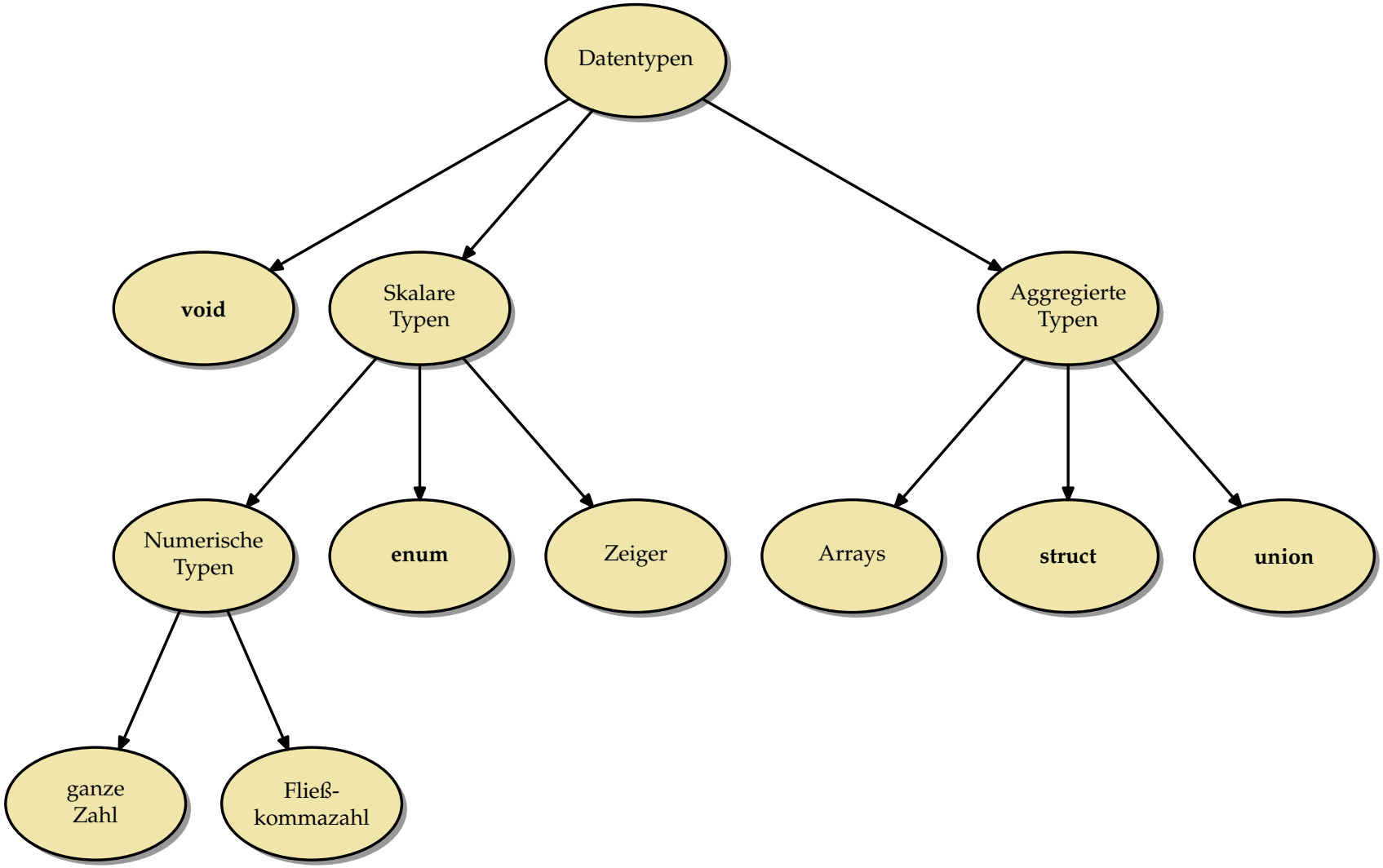
- Durch **#include** <stdio.h> werden direkt oder indirekt (durch weitere **#include**-Direktiven) Makros definiert, die zum Textersatz führen.
- Konkret definiert <*stdio.h*> das Makro *EOF* als (-1).
- Im Zweifelsfall hilft es, den Programmtext nach dem Präprozessor anzusehen und per *brute force* nach einem entsprechenden Makro zu suchen.

Datentypen

Datentypen legen

- den *Speicherbedarf*,
- die *Interpretation* des Speicherplatzes sowie
- die *erlaubten Operationen* fest.

Übersicht


Datentypen
void
Skalare Typen
Aggregierte Typen
Numerische Typen
enum
Zeiger
Arrays
struct
union
ganze Zahl
Fließ-kommazahl

- Im einfachsten Falle lässt sich eine Variablenvereinbarung sehr einfach zerlegen in die Angabe eines Typs und die des Variablennamens:
  int *i*;
  Hier ist *i* der Variablenname und **int** der Typ.
- Diese Zweiteilung entspricht soweit der Grammatik, wobei ⟨declaration-specifiers⟩ zu „**int**“ expandiert und „*i*“ zu ⟨init-declarator-list⟩:

$$
\begin{array}{rcl}
\langle\text{declaration}\rangle & \longrightarrow & \langle\text{declaration-specifiers}\rangle \\
 & & [\ \langle\text{init-declarator-list}\rangle\ ] \\
\langle\text{declaration-specifiers}\rangle & \longrightarrow & \langle\text{declaration-specifier}\rangle \\
 & & [\ \langle\text{declaration-specifiers}\rangle\ ]
\end{array}
$$

- Leider trennt die Syntax nicht in jedem Falle sauber den Namen von dem Typ.
- Beispiel:
  int∗ *ip*;
- Hier besteht die linke Seite, d.h. der ⟨declaration-specifier⟩ nur aus **int**. Der Dereferenzierungs-Operator wird stattdessen syntaktisch der rechten Seite, der ⟨init-declarator-list⟩ zugeordnet.
- Dies hat zur Konsequenz, dass bei
  int∗ *ip1*,*ip2*;
  ip1 und *ip2* unterschiedliche Typen erhalten. So ist *ip1* ein Zeiger auf **int**, während *ip2* schlicht nur den Typ **int** hat.

Skalare Datentypen

- Zu den skalaren Datentypen gehören alle Typen, die entweder numerisch sind oder sich zu einem numerischen Typ konvertieren lassen.
- Das schließt Datentypen wie **int**, **double** und auch beliebige Zeigertypen ein.
- Ein Wert eines skalaren Datentyps kann ohne weitere Konvertierung in einer Bedingung verwendet werden.
- Entsprechend wird die 0 im entsprechenden Kontext auch als Null-Zeiger interpretiert. (Ein Schlüsselwort wie **null** oder **nullptr** wird von C nicht unterstützt.)
- Und umgekehrt ist ein Null-Zeiger äquivalent zu *false* und ein Nicht-Null-Zeiger entspricht innerhalb einer Bedingung *true*.
- Ferner liegt die Nähe zwischen Zeigern und ganzen Zahlen auch in der von C unterstützten Adressarithmetik begründet.

Ganzzahlige Datentypen

⟨integer-type-specifier⟩	$\longrightarrow$	⟨signed-type-specifier⟩
	$\longrightarrow$	⟨unsigned-type-specifier⟩
	$\longrightarrow$	⟨character-type-specifier⟩
	$\longrightarrow$	⟨bool-type-specifier⟩
⟨signed-type-specifier⟩	$\longrightarrow$	[**signed**] **short** [**int**]
	$\longrightarrow$	[**signed**] **int**
	$\longrightarrow$	[**signed**] **long** [**int**]
	$\longrightarrow$	[**signed**] **long long** [**int**]
⟨unsigned-type-specifier⟩	$\longrightarrow$	**unsigned short** [**int**]
	$\longrightarrow$	**unsigned** [**int**]
	$\longrightarrow$	**unsigned long** [**int**]
	$\longrightarrow$	**unsigned long long** [**int**]
⟨character-type-specifier⟩	$\longrightarrow$	**char**
	$\longrightarrow$	**signed char**
	$\longrightarrow$	**unsigned char**
⟨bool-type-specifier⟩	$\longrightarrow$	**_Bool**

Ganzzahlige Datentypen

- Die Spezifikation eines ganzzahligen Datentyps besteht aus einem oder mehreren Schlüsselworten, die die Größe festlegen, und dem optionalen Hinweis, ob der Datentyp vorzeichenbehaftet ist oder nicht.
- Fehlt die Angabe von **signed** oder **unsigned**, so wird grundsätzlich **signed** angenommen.
- Die einzigen Ausnahmen hiervon sind **char** und **_Bool**.
- Bei **char** darf der Übersetzer selbst eine Voreinstellung treffen, die sich am effizientesten auf der Zielarchitektur umsetzen lässt.

Auch wenn Angaben wie **short** oder **long** auf eine gewisse Größe hindeuten, so legt keiner der C-Standards die damit verbundenen tatsächlichen Größen fest. Stattdessen gelten nur folgende Regeln:

- Der jeweilige „größere“ Datentyp in der Reihe **char**, **short**, **int**, **long**, **long long** umfasst den Wertebereich der kleineren Datentypen, d.h. **char** ist nicht größer als **short**, **short** nicht größer als **int** usw.
- Für jeden der ganzzahligen Datentypen gibt es Mindestintervalle, die abgedeckt sein müssen. (Die zugehörige Übersichtstabelle folgt.)
- Die korrespondierenden Datentypen mit und ohne Vorzeichen (etwa **signed int** und **unsigned int**) belegen exakt den gleichen Speicherplatz und verwenden die gleiche Zahl von Bits. (Entsprechende Konvertierungen erfolgen entsprechend der Semantik des Zweier-Komplements.)

In C werden alle ganzzahligen Datentypen durch Bitfolgen fester Länge repräsentiert: $\{a_i\}_{i=1}^{n}$ mit $a_i \in \{0, 1\}$. Bei ganzzahligen Datentypen ohne Vorzeichen ergibt sich der Wert direkt aus der binären Darstellung:

$$a = \sum_{i=1}^{n} a_i 2^{i-1}$$

Daraus folgt, dass der Wertebereich bei n Bits im Bereich von 0 bis $2^n - 1$ liegt.

Bei ganzzahligen Datentypen mit Vorzeichen übernimmt a_n die Rolle des Vorzeichenbits. Für die Repräsentierung gibt es bei C11 (und den früheren Standards) nur drei zugelassene Varianten:

- **Zweier-Komplement**:

  $$a = \sum_{i=1}^{n-1} a_i 2^{i-1} - a_n 2^{n-1}$$

  Wertebereich: $\left[-2^{n-1}, 2^{n-1} - 1\right]$
  Diese Darstellung hat sich durchgesetzt und wird von fast allen Prozessor-Architekturen unterstützt.

- Vorsicht: Im Wertebereich ist -2^{n-1} enthalten, jedoch nicht 2^{n-1}, somit liefert das unäre Minus angewendet auf die niedrigste Zahl nicht das erwartete Resultat.

- **Einer-Komplement**:

$$a = \sum_{i=1}^{n-1} a_i 2^{i-1} - a_n \left(2^{n-1} - 1\right)$$

Wertebereich: $\left[-2^{n-1} + 1, 2^{n-1} - 1\right]$
Vorsicht: Es gibt zwei Repräsentierungen für die Null. Es gilt:
$-a == \sim a$
Diese Darstellung gibt es auf einigen historischen Architekturen wie etwa der PDP-1, der UNIVAC 1100/2200 oder der 6502-Architektur.

- **Trennung zwischen Vorzeichen und Betrag**:

$$a = (-1)^{a_n} \sum_{i=1}^{n-1} a_i 2^{i-1}$$

Wertebereich: $\left[-2^{n-1} + 1, 2^{n-1} - 1\right]$
Vorsicht: Es gibt zwei Repräsentierungen für die Null.
Diese Darstellung wird ebenfalls nur von historischen Architekturen verwendet wie etwa der IBM 7090.

Wie wird ausgehend von a die Darstellung von $-a$ bestimmt?

- Zuerst wird das 1-Komplement gebildet, d.h. alle Bits in der Darstellung von a werden invertiert: Aus Einern werden Nullen und aus Nullen werden Einer.
- Danach wird 1 addiert.

Beispiel für $a = 5$ und $n = 4$:

	Darstellung	Wert
Ausgangswert	0101	$2^0 + 2^2 = 5$
1-Komplement	1010	$2^1 - 2^3 = -6$
1 addiert	1011	$2^0 + 2^1 - 2^3 = -5$

Zweierkomplement

Das Zweierkomplement ist anzusehen als Komplement in Bezug auf 2^n, d.h. es gilt bei der (vorzeichenlosen) Addition von a und dem im Zweierkomplement dargestellten $-a$

$$a + (-a) \;=\; 2^n,$$

falls wir $n + 1$ Bits zur Darstellung des Resultats hätten.

Beispiel: $0101 + 1011 = 10000$

C unterstützt einige Bit-Operationen für ganzzahlige Datentypen:

$\sim x$	bitweises 1-Komplement; bei ganzen Zahlen ohne Vorzeichen entspricht dies dem maximalen Wert des Typs abzüglich von x.
$x \mid y$	bitweise Oder-Operation
$x \,\&\, y$	bitweise Und-Operation
$x \,\hat{}\, y$	bitweise Exklusiv-Oder-Operation
$x << i$	Verschiebeoperation mit dem Wert $x \times 2^i$, wobei $i \geq 0$ gelten und das Ergebnis repräsentierbar sein muss, damit es wohldefiniert ist
$x >> i$	Verschiebeoperation mit dem Wert $\lfloor \frac{x}{2^i} \rfloor$. Bei vorzeichenbehafteten ganzen Zahlen lässt der Standard den Effekt undefiniert, falls $x < 0$.

Mit Hilfe der Bit-Operationen lassen sich für Mengen mit sehr kleinen Kardinalitäten ganze Zahlen ohne Vorzeichen verwenden. Seien A und B Variablen des Typs **unsigned int**, die zwei Mengen repräsentieren. Die Elemente der Menge werden durch ganze Zahlen repräsentiert von 0 bis maximal *WORD_BIT*−1 (aus **#include** <limits.h>).

Mengenoperation	Bit-Operation in C
$A \cup B$	$A \mid B$
$A \cap B$	A & B
$A \setminus B$	A & ~B
$\{i\}$	1<<i

Analog gibt es auch *LONG_BIT* für **long**. Sowohl *WORD_BIT* als auch *LONG_BIT* gibt es nur im Rahmen des POSIX-Standards.

Was passiert bei Überläufen?

Was passiert bei einer Addition, Subtraktion oder Multiplikation, die den Wertebereich des jeweiligen Datentyps verlässt?

- Bei vorzeichenbehafteten ganzen Zahlen ist das Resultat undefiniert. In der Praxis bedeutet dies, dass wir die repräsentierbaren niederwertigen Bits im Zweierkomplement erhalten.
- Bei ganzen Zahlen ohne Vorzeichen stellt C sicher, dass wir das korrekte Resultat modulo 2^n erhalten.

Alle gängigen Prozessorarchitekturen erkennen einen Überlauf, dies wird aber von C nicht unterstützt. Das wird in Java genauso gehandhabt.

Was passiert, wenn durch 0 geteilt wird?

div0.c

```
int main() {
   int i = 1; int j = 0;
   int k = i / j;
   return k;
}
```

- Dies ist generell offen.
- Es kann zu einem undefinierten Resultat führen oder zu einem Abbruch der Programmausführung.
- Letzteres ist die Regel.

```
clonard$ gcc -std=gnu11 -Wall -o div0 div0.c && ./div0
Arithmetic Exception (core dumped)
clonard$
```

Mindestbereiche bei ganzen Zahlen

Datentyp	Bits	Intervall	Konstanten
signed char	8	$[-127, 127]$	*SCHAR_MIN*, *SCHAR_MAX*
unsigned char	8	$[0, 255]$	*UCHAR_MAX*
char	8		*CHAR_MIN*, *CHAR_MAX*
short	16	$[-32767, 32767]$	*SHRT_MIN*, *SHRT_MAX*
unsigned short	16	$[0, 65535]$	*USHRT_MAX*
int	16	$[-32767, 32767]$	*INT_MIN*, *INT_MAX*
unsigned int	16	$[0, 65535]$	*UINT_MAX*
long	32	$\left[-2^{31}+1, 2^{31}-1\right]$	*LONG_MIN*, *LONG_MAX*
unsigned long	32	$[0, 4294967295]$	*ULONG_MAX*
long long	64	$\left[-2^{63}+1, 2^{63}-1\right]$	*LLONG_MIN*, *LLONG_MAX*
unsigned long long	64	$\left[0, 2^{64}-1\right]$	*ULLONG_MAX*

Einige typische Konstellationen haben sich durchgesetzt:

	32-Bit	64-Bit		
		LLP	LP	ILP
sizeof(short)	2	2	2	2
sizeof(int)	4	4	4	8
sizeof(long int)	4	4	8	8
sizeof(long long int)	8	8	8	8
sizeof(void∗)	4	8	8	8

- Das 32-Bit-Modell lässt sich durch die Übersetzungsoption „-m32“ auswählen.
- Mit „-m64“ gibt es einen 64-Bit-Adressraum. Dies ist inzwischen überwiegend die Voreinstellung. Linux, Solaris und OS X unterstützen das LP-Modell, Microsoft Windows das LLP-Modell.

Um portabel programmieren zu können ohne Kenntnis des verwendeten Modells, empfiehlt es sich, folgende vordefinierten Datentypen zu verwenden, die über <*stddef.h*> zugänglich sind:

- *size_t* ist der ganzzahlige Datentyp ohne Vorzeichen, den **sizeof** zurückliefert. Dieser Typ kann auch grundsätzlich als Index für Arrays verwendet werden. (Bei LLP oder LP kann **int** dafür zu klein sein.)
- *uintptr_t* (ohne Vorzeichen) bzw. *intptr_t* (mit Vorzeichen) können einen Zeigerwert aufnehmen.
- *ptrdiff_t* (mit Vorzeichen) wird für die Differenz zweier Zeiger zurückgeliefert.

Normalerweise haben diese ganzzahligen Typen die gleiche Größe (nämlich die von **sizeof**(**void***)), aber auf einigen historischen Architekturen (z.B. mit segmentierten Adressräumen) können diese voneinander abweichen.

Datentypen für Zeichen

- Der Datentyp **char** orientiert sich in seiner Größe typischerweise an dem Byte, der kleinsten adressierbaren Einheit.
- In <*limits.h*> findet sich die Konstante *CHAR_BIT*, die die Anzahl der Bits bei **char** angibt. Dieser Wert muss mindestens 8 betragen und weicht davon (von historischen Architekturen abgesehen) auch nicht ab.
- Der Datentyp **char** gehört mit zu den ganzzahligen Datentypen und entsprechend können Zeichen wie ganze Zahlen und umgekehrt behandelt werden.
- Der C-Standard überlässt den Implementierungen die Entscheidung, ob **char** vorzeichenbehaftet ist oder nicht. Wer sicher gehen möchte, spezifiziert dies explizit mit **signed char** oder **unsigned char**.
- Für größere Zeichensätze gibt es den Datentyp *wchar_t* aus <*wchar.h*> bzw. ab C11 über <*uchar.h*> auch *char16_t* und *char32_t*.

char mit oder ohne Vorzeichen?

char.c

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

int main() {
   printf("on this platform, char equals ");
   if (CHAR_MIN == SCHAR_MIN) {
      printf("signed ");
   } else {
      printf("unsigned ");
   }
   printf("char with values within [%d,%d]\n", CHAR_MIN, CHAR_MAX);
}
```

```
theon$ gcc -Wall -o char char.c
theon$ ./char
on this platform, char equals signed char with values within [-
128,127]
theon$ gcc -Wall -o char -funsigned-char char.c
theon$ ./char
on this platform, char equals unsigned char with values within [0,255]
theon$
```

Zeichenkonstanten

Zeichenkonstanten werden in einfache Hochkommata eingeschlossen, etwa 'a' (vom Datentyp **char**) oder *L*'a' (vom Datentyp *wchar_t*).
Für eine Reihe von nicht druckbaren Zeichen gibt es Ersatzdarstellungen:

\a	BEL	*alert*, Signalton
\b	BS	*backspace*
\f	FF	*formfeed*
\n	LF	*newline*, Zeilentrenner
\r	CR	*carriage return*, „Wagenrücklauf“
\t	HT	Horizontaler Tabulator
\v	VT	Vertikaler Tabulator
\\	\	„Fluchtsymbol“
\'	'	einfaches Hochkomma
\0	NUL	Null-Byte
\ddd		ASCII-Code (oktal)

rot13.c

```
#include <stdio.h>

const int letters = 'z' - 'a' + 1;
const int rotate = 13;
int main() {
   int ch;
   while ((ch = getchar()) != EOF) {
      if (ch >= 'a' && ch <= 'z') {
         ch = 'a' + (ch - 'a' + rotate) % letters;
      } else if (ch >= 'A' && ch <= 'Z') {
         ch = 'A' + (ch - 'A' + rotate) % letters;
      }
      putchar(ch);
   }
}
```

$$
\begin{aligned}
\langle\text{floating-point-type-specifier}\rangle &\longrightarrow \textbf{float} \\
&\longrightarrow \textbf{double} \\
&\longrightarrow \textbf{long double} \\
&\longrightarrow \langle\text{complex-type-specifier}\rangle \\
\langle\text{complex-type-specifier}\rangle &\longrightarrow \textbf{float _Complex} \\
&\longrightarrow \textbf{double _Complex} \\
&\longrightarrow \textbf{long double _Complex}
\end{aligned}
$$

- In der Vergangenheit gab es eine Vielzahl stark abweichender Darstellungen für Gleitkommazahlen, bis 1985 mit dem Standard IEEE-754 (auch IEC 60559 genannt) eine Vereinheitlichung gelang, die sich rasch durchsetzte und von allen heute üblichen Prozessor-Architekturen unterstützt wird.
- Der C-Standard bezieht sich ausdrücklich auf IEEE-754, auch wenn die Einhaltung davon nicht für Implementierungen garantiert werden kann, bei denen die Hardware-Voraussetzungen dafür fehlen.

Bei IEEE-754 besteht die binäre Darstellung einer Gleitkommazahl aus drei Komponenten,

- dem Vorzeichen s (ein Bit),
- dem aus q Bits bestehenden Exponenten $\{e_i\}_{i=1}^{q}$,
- und der aus p Bits bestehenden Mantisse $\{m_i\}_{i=1}^{p}$.

- Für die Darstellung des Exponenten e hat sich folgende verschobene Darstellung als praktisch erwiesen:

$$e = -2^{q-1} + 1 + \sum_{i=1}^{q} e_i 2^{i-1}$$

- Entsprechend liegt e im Wertebereich $\left[-2^{q-1} + 1, 2^{q-1}\right]$.
- Da die beiden Extremwerte für besondere Kodierungen verwendet werden, beschränkt sich der reguläre Bereich von e auf $[e_{min}, e_{max}]$ mit $e_{min} = -2^{q-1} + 2$ und $e_{max} = 2^{q-1} - 1$.
- Bei dem aus insgesamt 32 Bits bestehenden Format für den Datentyp **float** mit $q = 8$ ergibt das den Bereich $[-126, 127]$.
- Bei **double** mit insgesamt 64 Bit, davon $q = 11$ für den Exponenten, erhalten wir den Bereich $[-1022, 1023]$.

Normalisierte Darstellung

- Wenn e im Intervall $[e_{min}, e_{max}]$ liegt, dann wird die Mantisse m so interpretiert:

$$m = 1 + \sum_{i=1}^{p} m_i 2^{i-p-1}$$

- Wie sich dieser sogenannten normalisierten Darstellung entnehmen lässt, gibt es ein implizit auf 1 gesetztes Bit, d.h. m entspricht der im Zweier-System notierten Zahl $1, m_p m_{p-1} \ldots m_2 m_1$.
- Der gesamte Wert ergibt sich dann aus $x = (-1)^s \times 2^e \times m$.
- Um die 0 darzustellen, gilt der Sonderfall, dass $m = 0$, wenn alle Bits des Exponenten gleich 0 sind, d.h. $e = -2^{q-1} + 1 = e_{min} - 1$, und zusätzlich auch alle Bits der Mantisse gleich 0 sind. Da das Vorzeichenbit unabhängig davon gesetzt sein kann oder nicht, gibt es zwei Darstellungen für die Null: -0 und $+0$.

- IEEE-754 unterstützt auch die sogenannte denormalisierte Darstellung, bei der alle Bits des Exponenten gleich 0 sind (also $e = e_{min} - 1$), es aber in der Mantisse mindestens ein Bit mit $m_i = 1$ gibt. In diesem Falle ergibt sich folgende Interpretation:

$$\begin{aligned} m &= \sum_{i=1}^{p} m_i 2^{i-p-1} \\ x &= (-1)^s \times 2^{e_{min}} \times m \end{aligned}$$

- Der Fall $e = e_{max} + 1$ erlaubt es, ∞, $-\infty$ und *NaN* (*not a number*) mit in den Wertebereich der Gleitkommazahlen aufzunehmen. ∞ und $-\infty$ werden bei Überläufen verwendet und NaN bei undefinierten Resultaten (Beispiel: Wurzel aus einer negativen Zahl).

Unter- und Überläufe bei float

overflow.c

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

int main() {
   float ov = 1.0, uv = 1.0;
   for (int i = 0; isfinite(ov) || uv > 0; ov *= 2, uv /= 2, ++i) {
      printf("%3d %20g %20g\n", i, uv, ov);
   }
}
```

- Mit *isfinite* aus <*math.h*> kann getestet werden, ob eine Gleitkommazahl einen endlichen Wert besitzt, d.h. bei *NaN*, ∞ und $-\infty$ liefert *isfinite false*.

```
clonmel$ overflow | sed -n '1p; 128,129p; 148,$p'
  0                    1                    1
127          5.87747e-39          1.70141e+38
128          2.93874e-39                  Inf
147          5.60519e-45                  Inf
148           2.8026e-45                  Inf
149           1.4013e-45                  Inf
clonmel$
```

fclass.c

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

const char* get_class(float f) {
   switch (fpclassify(f)) {
      case FP_INFINITE:  return "infinite";
      case FP_NAN:       return "NaN";
      case FP_NORMAL:    return "normal";
      case FP_SUBNORMAL: return "subnormal";
      case FP_ZERO:      return "zero";
      default:           return "?";
   }
}

int main() {
   for (int exp = 126; exp <= 128; ++exp) {
      float f1 = exp2f(exp);
      printf("2^%d: %s\n", exp, get_class(f1));
      float f2 = exp2f(-exp);
      printf("2^-%d: %s\n", exp, get_class(f2));
   }
   printf("sqrt(-1): %s\n", get_class(sqrt(-1)));
}
```

```
clonmel$ fclass
2^126: normal
2^-126: normal
2^127: normal
2^-127: subnormal
2^128: infinite
2^-128: subnormal
sqrt(-1): NaN
clonmel$
```

- Der Bereich der normalisierten Exponenten erstreckt sich bei **float** auf $[-126, 127]$. Entsprechend lässt sich 2^{127} noch darstellen, während bei 2^{-127} bereits auf die denormalisierte Darstellung zurückgegriffen werden muss. 2^{128} ist nicht mehr darstellbar, während bei der denormalisierten Darstellung das Ende erst bei 2^{-149} erreicht wird, d.h. 2^{-150} würde zur Null werden.

- IEEE-754 gibt Konfigurationen für einfache, doppelte und erweiterte Genauigkeiten vor, die auch so von C übernommen wurden.
- Allerdings steht nicht auf jeder Architektur **long double** zur Verfügung, so dass in solchen Fällen ersatzweise nur eine doppelte Genauigkeit verwendet wird.
- Umgekehrt rechnen einige Architekturen grundsätzlich mit einer höheren Genauigkeit und runden dann, wenn eine Zuweisung an eine Variable des Typs **float** oder **double** erfolgt. Dies alles ist entsprechend IEEE-754 zulässig – auch wenn dies zur Konsequenz hat, dass Ergebnisse selbst bei elementaren Operationen auf verschiedenen konformen Architekturen voneinander abweichen können.
- Hier ist die Übersicht:

Datentyp	Bits	q	p
float	32	8	23
double	64	11	52
long double		≥ 15	≥ 63

Problematik von Rundungsfehlern

- Rundungsfehler beim Umgang mit Gleitkomma-Zahlen sind unvermeidlich.
- Sie entstehen in erster Linie, wenn Werte nicht exakt darstellbar sind. So gibt es beispielsweise keine Repräsentierung für 0,1. Stattdessen kann nur eine der „Nachbarn“ verwendet werden.
- Bedauerlicherweise können selbst kleine Rundungsfehler katastrophale Ausmaße nehmen.
- Dies passiert beispielsweise, wenn Werte völlig unterschiedlicher Größenordnungen zueinander addiert oder voneinander subtrahiert werden. Dies kann dann zur Auslöschung wesentlicher Bits der kleineren Größenordnung führen.

Flächenberechnung eines Dreiecks

- Gegeben seien die Längen a, b, c eines Dreiecks. Zu berechnen ist die Fläche A des Dreiecks.
- Dazu bietet sich folgende Berechnungsformel an:

$$
\begin{aligned}
s &= \frac{a+b+c}{2} \\
A &= \sqrt{s(s-a)(s-b)(s-c)}
\end{aligned}
$$

triangle.c

```
double triangle_area1(double a, double b, double c) {
   double s = (a + b + c) / 2;
   return sqrt(s*(s-a)*(s-b)*(s-c));
}
```

- Bei der Addition von $a + b + c$ kann bei einem schmalen Dreieck die kleine Seitelänge verschwinden, wenn die Größenordnungen weit genug auseinander liegen.
- Wenn dann später die Differenz zwischen s und der kleinen Seitelänge gebildet wird, kann der Fehler katastrophal werden.
- William Kahan hat folgende alternative Formel vorgeschlagen, die diese Problematik vermeidet:

$$A = \frac{\sqrt{(a+(b+c))(c-(a-b))(c+(a-b))(a+(b-c))}}{4}$$

Wobei hier die Werte a, b und c so zu vertauschen sind, dass gilt: $a \geq b \geq c$.

triangle.c

```
#define SWAP(a,b) {int tmp; tmp = a; a = b; b = tmp;}
double triangle_area2(double a, double b, double c) {
   /* sort a, b, and c in descending order,
      applying a bubble-sort */
   if (a < b) SWAP(a, b);
   if (b < c) SWAP(b, c);
   if (a < b) SWAP(a, b);
   /* formula by W. Kahan */
   return sqrt((a + (b + c)) * (c - (a - b)) *
               (c + (a - b)) * (a + (b - c)))/4;
}
```

Flächenberechnung eines Dreiecks

triangle.c

```
int main() {
   double a, b, c;
   printf("triangle side lenghts a b c: ");
   if (scanf("%lf %lf %lf", &a, &b, &c) != 3) {
      printf("Unable to read three floats!\n");
      return 1;
   }
   double a1 = triangle_area1(a, b, c);
   double a2 = triangle_area2(a, b, c);
   printf("Formula #1 delivers %.16lf\n", a1);
   printf("Formula #2 delivers %.16lf\n", a2);
   printf("Difference: %lg\n", fabs(a1-a2));
}
```

```
dublin$ gcc -Wall -std=c99 triangle.c -lm
dublin$ a.out
triangle side lenghts a b c: 1e10 1e10 1e-10
Formula #1 delivers 0.0000000000000000
Formula #2 delivers 0.5000000000000000
Difference: 0.5
dublin$
```

Vergleich von Gleitkommazahlen

- Wann können zwei Gleitkommazahlen als gleich betrachtet werden?
- Oder wann kann das gleiche Resultat erwartet werden?
- Gilt beispielsweise $(x/y)*y == x$?
- Interessanterweise garantiert hier IEEE-754 die Gleichheit, falls n und m beide ganze Zahlen sind, die sich in doppelter Genauigkeit repräsentieren lassen (also **double**), $|m| < 2^{52}$ und $n = 2^i + 2^j$ für natürliche Zahlen i, j. (siehe Theorem 7 aus dem Aufsatz von Goldberg).
- Aber beliebig verallgemeinern lässt sich dies nicht.

equality.c

```
#include <stdio.h>
int main() {
   double x, y;
   printf("x y = ");
   if (scanf("%lf %lf", &x, &y) != 2) {
      printf("Unable to read two floats!\n");
      return 1;
   }
   if ((x/y)*y == x) {
      printf("equal\n");
   } else {
      printf("not equal\n");
   }
   return 0;
}
```

```
dublin$ gcc -Wall -std=c99 equality.c
dublin$ a.out
x y = 3 10
equal
dublin$ a.out
x y = 2 0.77777777777777
not equal
dublin$
```

- Gelegentlich wird nahegelegt, wegen den Rundungsfehlern auf den ==-Operator zu verzichten und stattdessen auf die Nähe zu testen, d.h. $x \sim y \Leftrightarrow |x - y| < \epsilon$, wobei ϵ für eine angenommene Genauigkeit steht.
- Dies lässt jedoch folgende Fragen offen:
  - Wie sollte ϵ gewählt werden?
  - Ist der Wegfall der (bei == selbstverständlichen) Äquivalenzrelation zu verschmerzen? (Schließlich lässt sich aus $x \sim y$ und $y \sim z$ nicht mehr $x \sim z$ folgern.)
  - Soll auch dann $x \sim y$ gelten, wenn beide genügend nahe an der 0 sind, aber die Vorzeichen sich voneinander unterscheiden.
- Die Frage nach einem Äquivalenztest lässt sich nicht allgemein beantworten, sondern hängt von dem konkreten Fall ab.

Aufzählungstypen

⟨enumeration-type-specifier⟩	⟶	⟨enumeration-type-definition⟩
	⟶	⟨enumeration-type-reference⟩
⟨enumeration-type-definition⟩	⟶	**enum** [⟨enumeration-tag⟩] „{“
		⟨enumeration-definition-list⟩ [„,“] „}“
⟨enumeration-tag⟩	⟶	⟨identifier⟩
⟨enumeration-definition-list⟩	⟶	⟨enumeration-constant-definition⟩
	⟶	⟨enumeration-definition-list⟩ „,“
		⟨enumeration-constant-definition⟩
⟨enumeration-constant-definition⟩	⟶	⟨enumeration-constant⟩
	⟶	⟨enumeration-constant⟩ „=“ ⟨expression⟩
⟨enumeration-constant⟩	⟶	⟨identifier⟩
⟨enumeration-type-reference⟩	⟶	**enum** ⟨enumeration-tag⟩

- Aufzählungsdatentypen sind grundsätzlich ganzzahlig und entsprechend auch kompatibel mit anderen ganzzahligen Datentypen.
- Welcher vorzeichenbehaftete ganzzahlige Datentyp als Grundtyp für Aufzählungen dient (etwa **int** oder **short**) ist nicht festgelegt.
- Steht zwischen **enum** und der Aufzählung ein Bezeichner (⟨identifier⟩), so kann dieser Name bei späteren Deklarationen (bei einer ⟨enumeration-type-reference⟩) wieder verwendet werden.
- Sofern nichts anderes angegeben ist, erhält das erste Aufzählungselement den Wert 0.
- Bei den übrigen Aufzählungselementen wird jeweils der Wert des Vorgängers genommen und 1 dazuaddiert.
- Diese standardmäßig vergebenen Werte können durch die Angabe einer Konstante verändert werden. Damit wird dann auch implizit der Wert der nächsten Konstante verändert, sofern die nicht ebenfalls explizit gesetzt wird.

- Gegeben sei folgendes (nicht nachahmenswertes) Beispiel:

```
enum msglevel {
   notice, warning, error = 10,
   alert = error + 10, crit, emerg = crit * 2,
   debug = -1, debug0
};
```

- Dann ergeben sich daraus folgende Werte: *notice* $= 0$, *warning* $= 1$, *error* $= 10$, *alert* $= 20$, *crit* $= 21$, *emerg* $= 42$, *debug* $= -1$ und *debug0* $= 0$. C stört es dabei nicht, dass zwei Konstanten (*notice* und *debug0*) den gleichen Wert haben.

days.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <time.h>

enum days { Monday, Tuesday, Wednesday, Thursday,
   Friday, Saturday, Sunday };
char* dayname[] = { "Monday", "Tuesday", "Wednesday",
   "Thursday", "Friday", "Saturday", "Sunday"
};

int main() {
   enum days day;
   for (day = Monday; day <= Sunday; ++day) {
      printf("Day %d = %s\n", day, dayname[day]);
   }
   /* seed the pseudo-random generator */
   unsigned int seed = time(0); srand(seed);
   /* select and print a pseudo-random day */
   enum days favorite_day = rand() % 7;
   printf("My favorite day: %s\n", dayname[favorite_day]);
}
```

$$
\begin{array}{rcl}
\langle\text{declaration}\rangle & \longrightarrow & \langle\text{declaration-specifiers}\rangle\ [\ \langle\text{init-declarator-list}\rangle\ ] \\
\langle\text{declaration-specifiers}\rangle & \longrightarrow & \langle\text{storage-class-specifier}\rangle\ [\ \langle\text{declaration-specifiers}\rangle\ ] \\
 & \longrightarrow & \langle\text{type-specifier}\rangle\ [\ \langle\text{declaration-specifiers}\rangle\ ] \\
 & \longrightarrow & \langle\text{type-qualifier}\rangle\ [\ \langle\text{declaration-specifiers}\rangle\ ] \\
 & \longrightarrow & \langle\text{function-specifier}\rangle\ [\ \langle\text{declaration-specifiers}\rangle\ ] \\
\langle\text{init-declarator-list}\rangle & \longrightarrow & \langle\text{init-declarator}\rangle \\
 & \longrightarrow & \langle\text{init-declarator-list}\rangle\ „,“\ \langle\text{init-declarator}\rangle \\
\langle\text{init-declarator}\rangle & \longrightarrow & \langle\text{declarator}\rangle \\
 & \longrightarrow & \langle\text{declarator}\rangle\ „=“\ \langle\text{initializer}\rangle \\
\langle\text{declarator}\rangle & \longrightarrow & [\ \langle\text{pointer}\rangle\ ]\ \langle\text{direct-declarator}\rangle \\
\langle\text{pointer}\rangle & \longrightarrow & „*“\ [\ \langle\text{type-qualifier-list}\rangle\ ] \\
 & \longrightarrow & „*“\ [\ \langle\text{type-qualifier-list}\rangle\ ]\ \langle\text{pointer}\rangle
\end{array}
$$

zeiger.c

```
#include <stdio.h>

int main() {
   int i = 13;
   int* p = &i; /* Zeiger p zeigt auf i; &i = Adresse von i */

   printf("i=%d, p=%p (Adresse), *p=%d (Wert)\n", i, p, *p);

   ++i;
   printf("i=%d, *p=%d\n", i, *p);

   ++*p; /* *p ist ein Links-Wert */
   printf("i=%d, *p=%d\n", i, *p);
}
```

Zeigerarithmetik

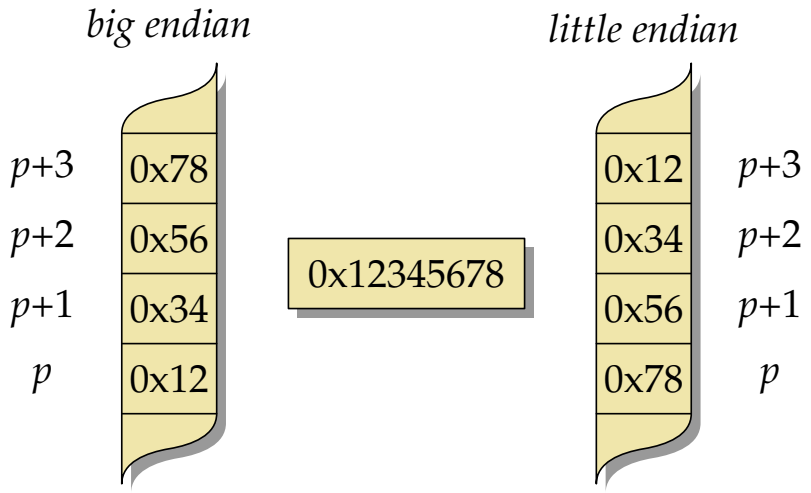
- Es ist zulässig, ganze Zahlen zu einem Zeiger zu addieren oder davon zu subtrahieren.
- Dabei wird jedoch der zu addierende oder zu subtrahierende Wert implizit mit der Größe des Typs multipliziert, auf den der Zeiger zeigt.
- Weiter ist es zulässig, Zeiger des gleichen Typs voneinander zu subtrahieren. Das Resultat wird dann implizit durch die Größe des referenzierten Typs geteilt.

zeiger1.c

```
#include <stdio.h>

int main() {
   unsigned int value = 0x12345678;
   unsigned char* p = (unsigned char*) &value;

   for (int i = 0; i < sizeof(unsigned int); ++i) {
      printf("p+%d --> 0x%02hhx\n", i, *(p+i));
   }
}
```

- Hier wird der Speicher byteweise „durchleuchtet“.
- Hierbei fällt auf, dass die interne Speicherung einer ganzen Zahl bei unterschiedlichen Architekturen (SPARC vs. Intel x86) verschieden ist: *big endian* vs. *little endian*.

big vs. little endian



- Bei *little endian* wird das niedrigstwertige Byte an der niedrigsten Adresse abgelegt, während bei
- *big endian* das niedrigstwertige Byte sich bei der höchsten Adresse befindet.

Typkonvertierungen

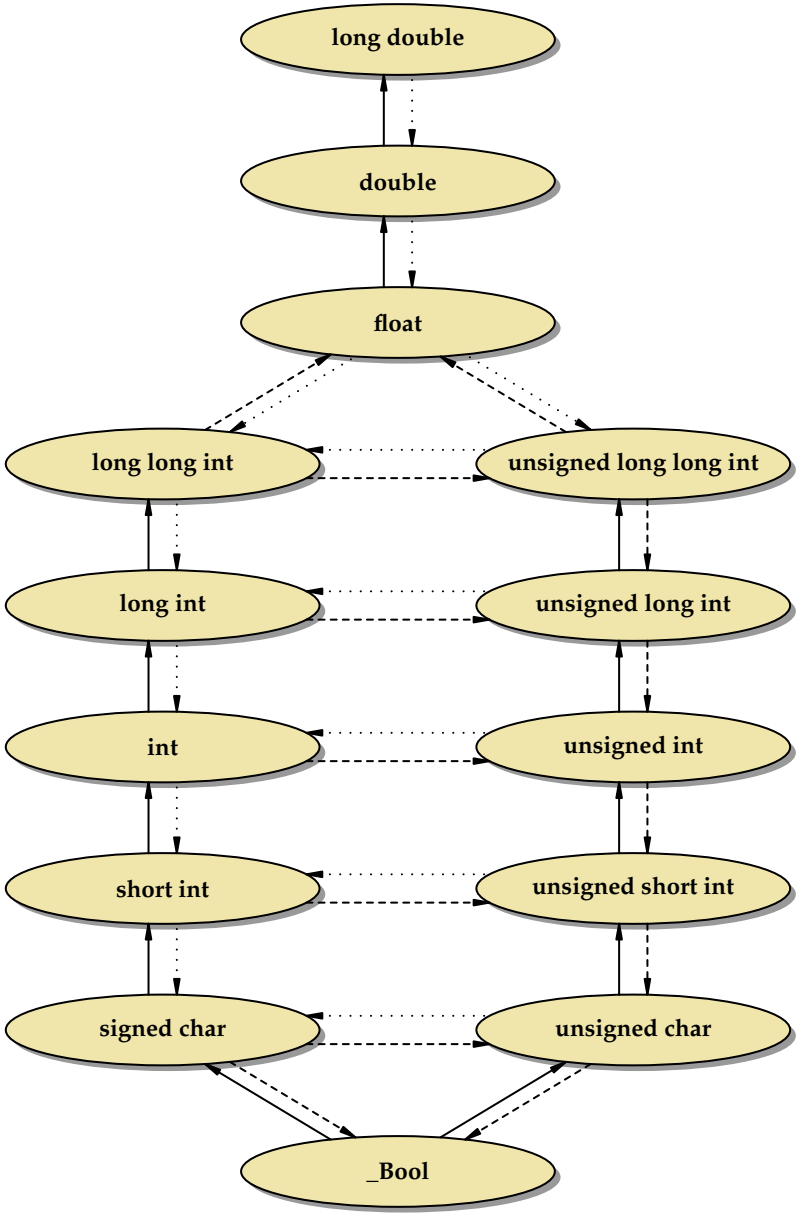
- Typ-Konvertierungen können in C sowohl implizit als auch explizit erfolgen.
- Implizite Konvertierungen werden angewendet bei Zuweisungs-Operatoren, Parameterübergaben und Operatoren. Letzteres schliesst auch die monadischen Operatoren mit ein.
- Explizite Konvertierungen erfolgen durch den sogenannten Cast-Operator.

Bei einer Konvertierung zwischen numerischen Typen gilt der Grundsatz, dass – wenn irgendwie möglich – der Wert zu erhalten ist. Falls das jedoch nicht möglich ist, gelten folgende Regeln:

- Bei einer Konvertierung eines vorzeichenbehafteten ganzzahligen Datentyps zum Datentyp ohne Vorzeichen *gleichen Ranges* (also etwa von **int** zu **unsigned int**) wird eine ganze Zahl $a < 0$ zu b konvertiert, wobei gilt, dass $a \bmod 2^n = b \bmod 2^n$ mit n der Anzahl der verwendeten Bits, wobei hier der mod-Operator entsprechend der F-Definition bzw. Euklid gemeint ist. Dies entspricht der Repräsentierung des Zweier-Komplements.
- Der umgekehrte Weg, d.h. vom ganzzahligen Datentyp ohne Vorzeichen zum vorzeichenbehafteten Datentyp gleichen Ranges (also etwa von **unsigned int** zu **int**) hinterlässt ein *undefiniertes* Resultat, falls der Wert nicht darstellbar ist.

- Bei einer Konvertierung von größeren ganzzahligeren Datentypen zu entsprechenden kleineren Datentypen werden die nicht mehr darstellbaren höherwertigen Bits weggeblendet, d.h. es gilt wiederum $a \bmod 2^n = b \bmod 2^n$, wobei n die Anzahl der Bits im kleineren Datentyp ist. (Das Resultat ist aber nur bei ganzzahligen Datentypen ohne Vorzeichen wohldefiniert.)
- Bei Konvertierungen zu *_Bool* ist das Resultat 0 (*false*), falls der Ausgangswert 0 ist, ansonsten immer 1 (*true*).
- Bei Konvertierungen von Gleitkommazahlen zu ganzzahligen Datentypen wird der ganzzahlige Anteil verwendet. Ist dieser im Zieltyp nicht darstellbar, so ist das Resultat undefiniert.
- Umgekehrt (beispielsweise auf dem Wege von **long long int** zu **float**) ist einer der beiden unmittelbar benachbarten darstellbaren Werte zu nehmen, d.h. es gilt entweder $a = b$ oder $a < b \wedge \nexists x : a < x < b$ oder $a > b \wedge \nexists x : a > x > b$ mit x aus der Menge des Zieltyps.

Konvertierungen bei numerischen Datentypen



- Jeder Aufzählungsdatentyp ist einem der ganzzahligen Datentypen implizit und implementierungsabhängig zugeordnet. Eine Konvertierung hängt von dieser (normalerweise nicht bekannten) Zuordnung ab.
- Zeiger lassen sich in C grundsätzlich als ganzzahlige Werte betrachten. Allerdings garantiert C nicht, dass es einen ganzzahligen Datentyp gibt, der den Wert eines Zeigers ohne Verlust aufnehmen kann.
- C99 hat hier die Datentypen *intptr_t* und *uintptr_t* in <*stdint.h*> eingeführt, die für die Repräsentierung von Zeigern als ganze Zahlen den geeignetsten Typ liefern.
- Selbst wenn diese groß genug sind, um Zeiger ohne Verlust aufnehmen zu können, so lässt der Standard dennoch offen, wie sich die beiden Typen *intptr_t* und *uintptr_t* innerhalb der Hierarchie der ganzzahligen Datentypen einordnen. Aber die weiteren Konvertierungsschritte und die damit verbundenen Konsequenzen ergeben sich aus dieser Einordnung.
- Die Zahl 0 wird bei einer Konvertierung in einen Zeigertyp immer in den Null-Zeiger konvertiert.

Implizite Konvertierungen

- Bei Zuweisungen wird der Rechts-Wert in den Datentyp des Links-Wertes konvertiert.
- Dies geschieht analog bei Funktionsaufrufen, wenn eine vollständige Deklaration der Funktion mit allen Parametern vorliegt.
- Wenn diese fehlt oder (wie beispielsweise bei *printf*) nicht vollständig ist, dann wird **float** implizit zu **double** konvertiert.

Die monadischen Operatoren !, $-$, $+$, $\sim$ und $*$ konvertieren implizit ihren Operanden:

- Ein vorzeichenbehafteter ganzzahliger Datentyp mit einem Rang niedriger als **int** wird zu **int** konvertiert,
- Ganzzahlige Datentypen ohne Vorzeichen werden ebenfalls zu **int** konvertiert, falls sie einen Rang niedriger als **int** haben und ihre Werte in jedem Falle von **int** darstellbar sind. Ist nur letzteres nicht der Fall, so erfolgt eine implizite Konvertierung zu **unsigned int**.
- Ranghöhere ganzzahlige Datentypen werden nicht konvertiert.

Die gleichen Regeln werden auch getrennt für die beiden Operanden der Schiebe-Operatoren $<<$ und $>>$ angewendet.

Implizite Konvertierung

Bei dyadischen Operatoren mit numerischen Operanden werden folgende implizite Konvertierungen angewendet:

- Sind die Typen beider Operanden vorzeichenbehaftet oder beide ohne Vorzeichen, so findet eine implizite Konvertierung zu dem Datentyp mit dem höheren Rang statt. So wird beispielsweise bei einer Addition eines Werts des Typs **short int** zu einem Wert des Typs **long int** der erstere in den Datentyp des zweiten Operanden konvertiert, bevor die Addition durchgeführt wird.
- Ist bei einem gemischten Fall (**signed** vs. **unsigned**) in jedem Falle eine Repräsentierung eines Werts des vorzeichenlosen Typs in dem vorzeichenbehafteten Typ möglich (wie etwa typischerweise bei **unsigned short** und **long int**), so wird der Operand des vorzeichenlosen Typs in den vorzeichenbehafteten Typ des anderen Operanden konvertiert.
- Bei den anderen gemischten Fällen werden beide Operanden in die vorzeichenlose Variante des höherrangigen Operandentyps konvertiert. So wird beispielsweise eine Addition bei **unsigned int** und **int** in **unsigned int** durchgeführt.

C sieht einige spezielle Attribute bei Typ-Deklarationen vor. Darunter ist auch **const**:

⟨declaration-specifiers⟩	$\longrightarrow$	⟨storage-class-specifier⟩ [⟨declaration-specifiers⟩]
	$\longrightarrow$	⟨type-specifier⟩ [⟨declaration-specifiers⟩]
	$\longrightarrow$	⟨type-qualifier⟩ [⟨declaration-specifiers⟩]
	$\longrightarrow$	⟨function-specifier⟩ [⟨declaration-specifiers⟩]
⟨type-qualifier⟩	$\longrightarrow$	**const**
	$\longrightarrow$	**volatile**
	$\longrightarrow$	**restrict**
	$\longrightarrow$	**_Atomic**

Die Verwendung des **const**-Attributs hat zwei Vorteile:

- Der Programmierer wird davor bewahrt, ein mit **const** deklarierte Variable versehentlich zu verändern. (Dies funktioniert aber nur beschränkt.)
- Besondere Optimierungen sind für den Übersetzer möglich, wenn bekannt ist, dass sich bestimmte Variablen nicht verändern dürfen.

const.c

```
#include <stdio.h>
int main() {
    const int i = 1;
    i++;        /* das geht doch nicht, oder?! */
    printf("i=%d\n", i);
}
```

- Ältere Versionen des gcc beschränken sich selbst dann nur auf Warnungen, wenn **const**-Variablen offensichtlich verändert werden. Ab gcc 4 wurde das geändert.

```
xylopia$ gcc --version | head -1
gcc (GCC) 3.2
xylopia$ gcc -o const const.c
const.c: In function `main':
const.c:6: warning: increment of read-only variable `i'
xylopia$ ./const
i=2
xylopia$
```

update-const.c

```
#include <stdio.h>

int data;

void set_data(int val) {
   data = val;
}

void work_with_dataptr(const int* data_ptr) {
   printf("*data_ptr = %d\n", *data_ptr);
   set_data(*data_ptr + 1);
   printf("*data_ptr = %d\n", *data_ptr);
}

int main() {
   work_with_dataptr(&data);
}
```

- Teilweise schränkt **const** auch den Zugriff nur auf das Lesen ein, während nicht ausgeschlossen ist, dass andere ihn verändern.

⟨direct-declarator⟩	⟶	⟨simple-declarator⟩
	⟶	„(“ ⟨simple-declarator⟩ „)“
	⟶	⟨function-declarator⟩
	⟶	⟨array-declarator⟩
⟨array-declarator⟩	⟶	⟨direct-declarator⟩ „[“ [⟨array-qualifier-list⟩] [⟨array-size-expression⟩] „]“
⟨array-qualifier-list⟩	⟶	⟨array-qualifier⟩
	⟶	⟨array-qualifier-list⟩ ⟨array-qualifier⟩
⟨array-qualifier⟩	⟶	**static**
	⟶	**restrict**
	⟶	**const**
	⟶	**volatile**
⟨array-size-expression⟩	⟶	⟨assignment-expression⟩
	⟶	„*“
⟨simple-declarator⟩	⟶	⟨identifier⟩

Deklaration von Arrays

- Wie bei den Zeigertypen erfolgt die Typspezifikation eines Arrays nicht im Rahmen eines ⟨type-specifier⟩.
- Stattdessen gehört eine Array-Deklaration zu dem ⟨init-declarator⟩. Das bedeutet, dass die Präzisierung des Typs zur genannten Variablen unmittelbar gehört.
- Entsprechend deklariert

```
int a[10], i;
```

  eine Array-Variable *a* und eine ganzzahlige Variable *i*.

- Arrays und Zeiger sind eng miteinander verwandt.
- Der Variablenname eines Arrays ist ein konstanter Zeiger auf den zugehörigen Element-Typ, der auf das erste Element verweist.
- Allerdings liefert **sizeof** mit dem Namen des Arrays als Operand die Größe des gesamten Arrays und nicht etwa nur die des Zeigers.
- Entsprechend liefert **sizeof**(a)/ **sizeof**(a[0]) die Anzahl der Elemente eines Arrays a. (Grundsätzlich gilt, dass **sizeof**(a[0]) ein Teiler von **sizeof**(a) ist, d.h. Alignment-Anforderungen eines Element-Typs erzwingen bei Bedarf eine Aufrundung des Speicherbedarfs des Element-Typs und nicht erst des Arrays.)

array.c

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

int main() {
   int a[] = {1, 2, 3, 4, 5};
   /* Groesse des Arrays bestimmen */
   const size_t SIZE = sizeof(a) / sizeof(a[0]);
   int* p = a;  /* kann statt a verwendet werden */
   /* aber: a weiss noch die Gesamtgroesse, p nicht */
   printf("SIZE=%zu, sizeof(a)=%zu, sizeof(p)=%zu\n",
      SIZE, sizeof(a), sizeof(p));
   for (size_t i = 0; i < SIZE; ++i) {
      *(a + i) = i+1; /* gleichbedeutend mit a[i] = i+1 */
   }
   /* Elemente von a aufsummieren */
   int sum = 0;
   for (size_t i = 0; i < SIZE; i++) {
      sum += p[i];   /* gleichbedeutend mit ... = a[i]; */
   }
   printf("Summe: %d\n", sum);
}
```

- Die Indizierung beginnt immer bei 0.
- Ein Array mit 5 Elementen hat entsprechend zulässige Indizes im Bereich von 0 bis 4.
- Wird mit einem Index außerhalb des zulässigen Bereiches zugegriffen, so ist der Effekt undefiniert.
- Es ist dann damit zu rechnen, dass irgendeine andersweitig belegte Speicherfläche adressiert wird oder es zu einer harten Unterbrechung kommt, weil eine unzulässige Adresse dereferenziert wurde. Was tatsächlich passiert, hängt von der jeweiligen Adressraumbelegung ab.
- Viele bekannte Sicherheitslücken beruhen darauf, dass in C-Programmen die zulässigen Indexbereiche verlassen werden und auf diese Weise eingeschleuster Programmtext zur Ausführung gebracht werden kann.
- Anders als in Java gibt es aber keine automatisierte Überprüfung. Diese wäre auch wegen der Verwandtschaft von Arrays und Zeigern nicht mit einem vertretbaren Aufwand in C umzusetzen.

In C ist das Belegen bzw. Deklarieren von Speicher für Arrays getrennt von dem Zugriff:

- Mit **int** $a[10]$; lässt sich ein Array mit 10 Elementen auf globaler oder lokaler Ebene deklarieren.
- Bei globalen Arrays erfolgt die Speicherbelegung beim Programmstart. Bei lokalen Arrays geschieht dies bei jedem Aufruf des umgebenden Blocks.
- Der Zugriff erfolgt über Zeiger und Zeigerarithmetik. Bei Zeigern ist jedoch nicht mehr der ursprüngliche Umfang des Arrays bekannt.
- Im folgenden Beispiel erfolgt der Zugriff über den Zeiger b:
  int$*$ $b = \&a[2]$; $b[1] = 42$;
  Nun hat $a[3]$ den Wert 42.
- Es ist darauf zu achten, dass der Speicher belegt bleibt, solange mit Zeigern noch darauf zugegriffen wird.

Parameterübergabe bei Arrays

- Da der Name eines Arrays nur ein Zeiger auf das erste Element ist, werden bei der Parameterübergabe entsprechend nur Zeigerwerte übergeben.
- Entsprechend arbeitet die aufgerufene Funktion nicht mit einer Kopie des Arrays, sondern hat dank dem Zeiger den direkten Zugriff auf das Array des Aufrufers.
- Die Dimensionierung eines Arrays muss explizit mit Hilfe weiterer Parameter übergeben werden, wenn diese variabel sein soll.

array2.c

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

const size_t SIZE = 10;

/* Array wird veraendert, naemlich mit
   0, 1, 2, 3, ... initialisiert! */
void init(int a[], size_t length) {
   for (size_t i = 0; i < length; i++) {
      a[i] = i;
   }
}

int summe1(int a[], size_t length) {
   int sum = 0;
   for (size_t i = 0; i < length; i++) {
      sum += a[i];
   }
   return sum;
}
```

array2.c

```
int summe2(int* a, size_t length) {
   int sum = 0;
   for (size_t i = 0; i < length; i++) {
      sum += *(a+i); /* aequivalent zu ... += a[i]; */
   }
   return sum;
}

int main() {
   int array[SIZE];

   init(array, SIZE);

   printf("Summe: %d\n", summe1(array, SIZE));
   printf("Summe: %d\n", summe2(array, SIZE));
}
```

array3.c

```
int summe3(size_t length, int a[length]) {
   int sum = 0;
   for (size_t i = 0; i < length; i++) {
      sum += a[i];
   }
   return sum;
}

int main() {
   int array[17];

   init(sizeof(array)/sizeof(array[0]), array);
   printf("Summe: %d\n", summe3(sizeof(array)/sizeof(array[0]), array));
}
```

- Seit C99 sind auch variabel lange Arrays möglich bei lokalen Deklarationen und Parameterübergaben.
- Dies wird jedoch von C++ nicht unterstützt und sollte daher vermieden werden.

- So könnte ein zweidimensionales Array angelegt werden:

```
int matrix[2][3];
```

- Eine Initialisierung ist sofort möglich. Die geschweiften Klammern werden dann entsprechend verschachtelt:

```
int matrix[2][3] = {{0, 1, 2}, {3, 4, 5}};
```

Angenommen, die Anfangsadresse des Arrays liege bei 0*x1000* und eine ganze Zahl vom Typ **int** würde vier Bytes belegen, dann würde die Repräsentierung des Arrays *matrix* im Speicher folgendermaßen aussehen:

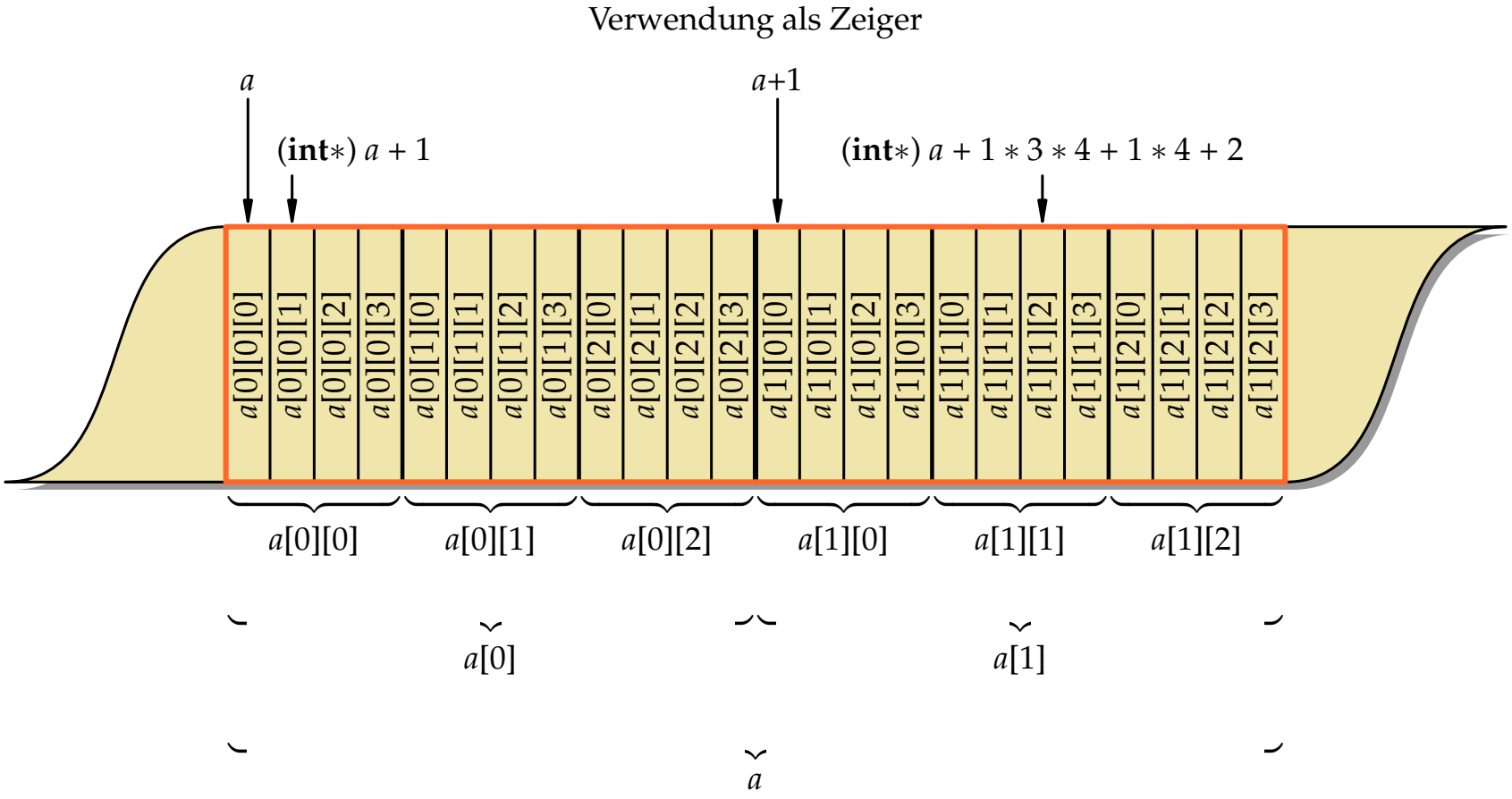
Element	**Adresse**	**Inhalt**
matrix[0][0]	0x1000	0
matrix[0][1]	0x1004	1
matrix[0][2]	0x1008	2
matrix[1][0]	0x100C	3
matrix[1][1]	0x1010	4
matrix[1][2]	0x1014	5

Diese zeilenweise Anordnung nennt sich *row-major* und hat sich weitgehend durchgesetzt mit der wesentlichen Ausnahme von Fortran, das Matrizen spaltenweise anordnet (*column-major*).

Repräsentierung eines Arrays im Speicher

- Gegeben sei:

```
int a[2][3][4];
```



Folgende Möglichkeiten stehen zur Verfügung:

- Alle Dimensionen mit Ausnahme der ersten werden explizit bei der Parameterdeklaration festgelegt. Nur die erste Dimension ist dann noch variabel.
- Das gesamte Array wird zu einem eindimensionalen Array verflacht. Eine mehrdimensionale Indizierung erfolgt dann „per Hand“.
- Beginnend mit C99 wird auch mehrdimensionale dynamische Parameterübergaben für Arrays unterstützt. Dies wird von C++ nicht unterstützt und entsprechend ist davon abzuraten.

dynarray.c

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

void print_matrix(size_t rows, size_t cols,
      const double A[rows][cols]) {
   for (size_t i = 0; i < rows; ++i) {
      for (size_t j = 0; j < cols; ++j) printf(" %10lg", A[i][j]);
      printf("\n");
   }
}

int main() {
   double A[][3] = {{1.0, 2.3, 4.7}, {2.3, 4.4, 9.9}};
   print_matrix(/* rows = */ sizeof(A)/sizeof(A[0]),
      /* cols = */ sizeof(A[0])/sizeof(A[0][0]), A);
}
```

- Die Dimensionierungsparameter müssen dem entsprechenden Array in der Parameterdeklaration vorangehen.

Matrizen (oder generell mehrdimensionale Arrays) können auch über Zeigerlisten realisiert werden:

- Jede Zeile ist ein eigenes Array.
- Die Zeigerliste ist ein Array von Zeigern auf die einzelnen Zeilen.
- Das gesamte Array wird dann durch einen einzigen Zeiger repräsentiert zuzüglich der Information über die Dimensionierungen.

Der Vorteil liegt darin, dass notationell der Zugriff auf die gleiche Weise erfolgt. Nachteil ist, dass solche Konstruktionen cache-unfreundlich sind, wenn die Zeilenarrays nicht kontinuierlich hintereinander im Speicher liegen. Java realisiert mehrdimensionale Arrays mit solchen Zeigerlisten.

ptrlist.c

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

void print_matrix(size_t rows, size_t cols, const double* const* A) {
   for (size_t i = 0; i < rows; ++i) {
      for (size_t j = 0; j < cols; ++j) printf(" %10lg", A[i][j]);
      printf("\n");
   }
}

int main() {
   const double A_rows[] = {
      1.0, 2.3, 4.7, 2.0,
      2.3, 4.4, 9.9, 3.4,
      2.1, 7.2, 3.1, 4.2};
   const double* A[] = {A_rows, A_rows + 4, A_rows + 8};
   print_matrix(3, 4, A);
}
```

Es bietet sich auch an, Matrizen durch ein eindimensionales Array zu repräsentieren. Wir benötigen dann zusätzliche Parameter:

- *incRow* spezifiziert, wieviel Elemente wir überspringen müssen, um zur nächsten Zeile zu gelangen.
- *incCol* spezifiziert, wieviel Elemente wir überspringen müssen, um zur nächsten Spalte zu gelangen.
- Bei einer *row-major*-Anordnung wird *incRow* auf die Zahl der Spalten gesetzt und *incCol* auf 1.
- Bei einer *col-major*-Anordnung wird *incCol* auf die Zahl der Zeilen gesetzt und *incRow* auf 1.

Diese Vorgehensweise ist flexibler und lässt sich auch bei C++ übernehmen.

dynarray2.c

```
void print_matrix(size_t rows, size_t cols,
      ptrdiff_t incRow, ptrdiff_t incCol,
      double* A) {
   for (size_t i = 0; i < rows; ++i) {
      for (size_t j = 0; j < cols; ++j) {
         printf(" %10lg", A[i * incRow + j * incCol]);
      }
      printf("\n");
   }
}
```

- Die Notaton $A[i][j]$ ist nun nicht mehr länger möglich. Stattdessen ist die Speicheradresse des gewünschten Elements explizit zu berechnen: $A[i * incRow + j * incCol]$.

dynarray2.c

```
void print_matrix(size_t rows, size_t cols,
      ptrdiff_t incRow, ptrdiff_t incCol,
      double* A) {
   for (size_t i = 0; i < rows; ++i) {
      for (size_t j = 0; j < cols; ++j) {
         printf(" %10lg", A[i * incRow + j * incCol]);
      }
      printf("\n");
   }
}

int main() {
   double A[4*3];
   init_matrix(4, 3, 3, 1, A); /* 4x3 matrix in row-major */
   printf("4x3 matrix A:\n");
   print_matrix(4, 3, 3, 1, A);
   printf("A transposed:\n");
   print_matrix(3, 4, 1, 3, A);
}
```

```
theon$ gcc -Wall -o dynarray2 dynarray2.c
theon$ dynarray2
4x3 matrix A:
          1          2          3
          4          5          6
          7          8          9
         10         11         12
A transposed:
          1          4          7         10
          2          5          8         11
          3          6          9         12
theon$
```

- Zum Transponieren einer Matrix sind nur *rows* und *cols* sowie *incRow* und *incCol* jeweils miteinander zu vertauschen.

mkl.h

```
/* from https://software.intel.com/en-us/node/520775 */
void cblas_dgemm (const CBLAS_LAYOUT Layout,
   const CBLAS_TRANSPOSE transa, const CBLAS_TRANSPOSE transb,
   const MKL_INT m, const MKL_INT n, const MKL_INT k,
   const double alpha,
   const double *a, const MKL_INT lda,
   const double *b, const MKL_INT ldb,
   const double beta,
   double *c, const MKL_INT ldc);
```

- Diese Funktionsdeklaration für eine Matrix-Matrix-Multiplikation entstammt der BLAS-Implementierung von Intel (Intel Math Kernel Library).
- BLAS steht für *Basic Linear Algebra Subprograms*, einer Standardschnittstelle, die sich seit 1979 entwickelt hat (ursprünglich nur für Fortran).

mkl.h

```
/* from https://software.intel.com/en-us/node/520775 */
void cblas_dgemm (const CBLAS_LAYOUT Layout,
   const CBLAS_TRANSPOSE transa, const CBLAS_TRANSPOSE transb,
   const MKL_INT m, const MKL_INT n, const MKL_INT k,
   const double alpha,
   const double *a, const MKL_INT lda,
   const double *b, const MKL_INT ldb,
   const double beta,
   double *c, const MKL_INT ldc);
```

- Die Funktion berechnet $C \leftarrow \alpha op_A(A) \times op_B(B) + \beta C$, wobei op_A und op_B entweder die identische Abbildung sind oder die Matrix jeweils transponieren (Parameter *transa* und *transb*).
- Der *layout*-Parameter legt fest, ob die Matrizen normalerweise in *col-major* (typisch für Fortran) oder in *row-major* angeordnet werden.

mkl.h

```
/* from https://software.intel.com/en-us/node/520775 */
void cblas_dgemm (const CBLAS_LAYOUT Layout,
   const CBLAS_TRANSPOSE transa, const CBLAS_TRANSPOSE transb,
   const MKL_INT m, const MKL_INT n, const MKL_INT k,
   const double alpha,
   const double *a, const MKL_INT lda,
   const double *b, const MKL_INT ldb,
   const double beta,
   double *c, const MKL_INT ldc);
```

- *lda*, *ldb* und *ldc* stehen für die sogenannten *leading dimensions*, d.h. statt *incRow* und *incCol* wird nur einer der beiden Parameter angegeben, der andere wird auf 1 gesetzt. Bei *col-major* wäre *incCol* bei der *leading dimension* anzugeben.
- Transponierungen führen dann zu entsprechenden Vertauschungen der Dimensionierungen und der Offsets *incRow* und *incCol*.

- Zeichenketten werden in C als Arrays von Zeichen repräsentiert: **char**[]
- Das Ende der Zeichenkette wird durch ein sogenanntes Null-Byte ('\0') gekennzeichnet.
- Da es sich bei Zeichenketten um Arrays handelt, werden bei der Parameterübergabe nur die Zeiger als Werteparameter übergeben.
- Die Zeichenkette (also der Inhalt des Arrays) kann entsprechend von der aufgerufenen Funktion verändert werden.

- Zeichenketten-Konstanten können durch von Doppelapostrophen eingeschlossene Zeichenfolgen spezifiziert werden. Hier im Rahmen einer Initialisierung:

```
char greeting[] = "Hallo";
```

- Dies ist eine Kurzform für

```
char greeting[] = {'H', 'a', 'l', 'l', 'o', '\0'};
```

- Eine Zeichenketten-Konstante steht für einen Zeiger auf den Anfang der Zeichenkette:

```
char* greeting = "Hallo";
```

- Wenn die Zeichenketten-Konstante nicht eigens mit einer Initialisierung in ein deklariertes Array kopiert wird und somit der Zugriff nur über einen Zeiger erfolgt, sind nur Lesezugriffe zulässig.

Zeichenketten: Was ist zulässig?

strings.c

```
#include <stdio.h>

int main() {
   char array[10];
   char string[] = "Hallo!"; /* Groesse wird vom Compiler bestimmt */
   char* s1 = "Welt";
   char* s2;

   /* array = "not OK"; */   /* nicht zulaessig */
   array[0] = 'A';           /* zulaessig */
   array[1] = '\0';
   printf("array: %s\n", array);
   /* s1[5] = 'B'; */        /* nicht zulaessig */
   s1 = "ok";                /* zulaessig */
   printf("s1: %s\n", s1);
   s2 = s1;                  /* zulaessig */
   printf("s2: %s\n", s2);
   string[0] = 'X';          /* zulaessig */
   printf("string: %s\n", string);
   printf("sizeof(string): %zu\n", sizeof(string));
}
```

strings1.c

```
/* Laenge einer Zeichenkette bestimmen */
size_t my_strlen1(char s[]) {
   size_t len = 0;
   while (s[len]) { /* mit Null-byte vergleichen */
      ++len;
   }
   return len;
}
```

- Die Bibliotheksfunktion *strlen()* liefert die Länge einer Zeichenkette zurück.
- Als Länge einer Zeichenkette wird die Zahl der Zeichen *vor* dem Null-Byte betrachtet.
- *my_strlen1()* bildet hier die Funktion nach unter Verwendung der vektoriellen Notation.

strings1.c

```
/* Laenge einer Zeichenkette bestimmen */
size_t my_strlen2(char* s) {
   char* t = s;
   while (*t++);
   return t-s-1;
}
```

- Alternativ wäre es auch möglich, mit der Zeigernotation zu arbeiten.
- Zu beachten ist hier, dass der Post-Inkrement-Operator $++$ einen höheren Rang hat als der Dereferenzierungs-Operator $*$.
- Entsprechend bezieht sich das Inkrement auf t. Das Inkrement wird aber erst *nach* der Dereferenzierung als verspäteter Seiteneffekt ausgeführt.

strings1.c

```
/* Kopieren einer Zeichenkette von s nach t
   Vorauss.: genuegend Platz in t */
void my_strcpy1(char t[], char s[]) {
   for (size_t i = 0; (t[i] = s[i]) != '\0'; i++);
}
```

- Das ist ein Nachbau der Bibliotheksfunktion *strcpy()* die (analog zur Anordnung bei einer Zuweisung) den linken Parameter als Ziel und den rechten Parameter als Quelle der Kopier-Aktion betrachtet.
- Hier zeigt sich auch eines der großen Probleme von C im Umgang mit Arrays: Da die tatsächlich zur Verfügung stehende Länge des Arrays *t* unbekannt bleibt, können weder *my_strcpy1()* noch die Laufzeitumgebung dies überprüfen.

strings1.c

```
/* Kopieren einer Zeichenkette von s nach t
   Vorauss.: genuegend Platz in t */
void my_strcpy2(char* t, char* s) {
   for (; (*t = *s) != '\0'; t++, s++);
}
```

- In der Zeigernotation wird es einfacher.

strings1.c

```
/* Kopieren einer Zeichenkette von s nach t
   Vorauss.: genuegend Platz in t */
void my_strcpy3(char* t, char* s) {
   while ((*t++ = *s++) != '\0');
}
```

- Die Inkrementierung lässt sich natürlich (wie schon bei der Längenbestimmung) mit integrieren.

strings1.c

```
/* Kopieren einer Zeichenkette von s nach t
   Vorauss.: genuegend Platz in t */
void my_strcpy4(char* t, char* s) {
   while ((*t++ = *s++));
}
```

- Der Vergleichstest mit dem Nullbyte lässt sich natürlich streichen.
- Allerdings gibt es dann eine Warnung des gcc, dass möglicherweise der Vergleichs-Operator == mit dem Zuweisungs-Operator = verwechselt worden sein könnte.
- Diese Warnung lässt sich (per Konvention) durch die doppelte Klammerung unterdrücken. Damit wird klar lesbar zum Ausdruck gegeben, dass es sich dabei nicht um ein Versehen handelt.

strings1.c

```
/* Vergleich zweier Zeichenketten
   Ergebnis: 0 fuer s = t, > 0 fuer s > t und < 0 fuer s < t */
int my_strcmp1(char s[], char t[]) {
   size_t i;
   for (i = 0; s[i] == t[i] && s[i] != '\0'; i++);
   return s[i] - t[i];
}
```

- Um alle sechs Vergleichsrelationen mit einer Funktion unterstützen zu können, arbeitet die Bibliotheksfunktion *strcmp()* mit einem ganzzahligen Rückgabewert, der < 0 ist, falls $s < t$, $= 0$ ist, falls s mit t übereinstimmt und > 0, falls $s > t$.

strings1.c

```
/* Vergleich zweier Zeichenketten
   Ergebnis: 0 fuer s = t, > 0 fuer s > t und < 0 fuer s < t */
int my_strcmp2(char* s, char* t) {
   for (; *s == *t && *s != '\0'; s++, t++);
   return *s - *t;
}
```

- Auch dies lässt sich in die Zeigernotation umsetzen.
- Auf ein integriertes Post-Inkrement wurde hier verzichtet, da dann die beiden Zeiger eins zu weit stehen, wenn es darum geht, die Differenz der unterschiedlichen Zeichen zu berechnen.

⟨structure-type-specifier⟩	⟶	**struct** [⟨identifier⟩] „{“
		⟨struct-declaration-list⟩ „}“
	⟶	**struct** ⟨identifier⟩
⟨struct-declaration-list⟩	⟶	⟨struct-declaration⟩
	⟶	⟨struct-declaration-list⟩ ⟨struct-declaration⟩
⟨struct-declaration⟩	⟶	⟨specifier-qualifier-list⟩
		⟨struct-declarator-list⟩ „;“
⟨specifier-qualifier-list⟩	⟶	⟨type-specifier⟩ [⟨specifier-qualifier-list⟩]
	⟶	⟨type-qualifier⟩ [⟨specifier-qualifier-list⟩]
⟨struct-declarator-list⟩	⟶	⟨struct-declarator⟩
	⟶	⟨struct-declarator-list⟩ „,“ ⟨struct-declarator⟩
⟨struct-declarator⟩	⟶	⟨declarator⟩
	⟶	[⟨declarator⟩] „:“ ⟨constant-expression⟩

- Ein *Verbundtyp* (in C auch Struktur genannt) fasst mehrere *Elemente* zu einem Datentyp zusammen. Im Gegensatz zu Arrays können die Elemente *unterschiedlichen* Typs sein.
- Mit dem Schlüsselwort **struct** kann ein Verbundtyp wie folgt deklariert werden:

```
struct datum {
   unsigned short tag, monat, jahr;
};
```

- Hier ist *datum* der *Name des Verbundtyps*, der allerdings nur in Verbindung mit dem Schlüsselwort **struct** erkannt wird. Der hier deklarierte Verbundtyp repräsentiert – wie der Name schon andeutet – ein Datum. Jede Variable dieses Verbundtyps besteht aus drei ganzzahligen Komponenten, dem Tag, dem Monat und dem Jahr.

- Eine Variable *geburtsdatum* des Verbundtyps **struct** *datum* kann danach wie folgt angelegt werden:

```
struct datum geburtsdatum;
```

- Analog zu Aufzählungen lassen sich auch Variablen für namenlose Verbundtypen anlegen:

```
struct {
   unsigned short tag, monat, jahr;
} my_geburtsdatum;
```

- Ohne den Namen fehlt jedoch die Möglichkeit, weitere Variablen dieses Typs zu deklarieren oder den Typnamen in einer Typkonvertierung oder einem Aggregat zu spezifizieren.

- Variablen eines Verbund-Typs können bereits bei ihrer Definition initialisiert werden:

```
struct datum geburtsdatum = {3, 5, 1978};
```

- Es ist auch zulässig, die Elementnamen für die Initialisierung zu verwenden:

```
struct datum geburtsdatum = {.tag = 3, .monat = 5, .jahr = 1978};
```

- Alternativ kann auch der Wert eines Verbundtyps innerhalb eines Ausdrucks mit Hilfe eines Aggregats konstruiert werden:

```
struct datum geburtsdatum;
geburtsdatum = (struct datum) {3, 5, 1978};
/* oder mit Namen: */
geburtsdatum = (struct datum) {.tag = 3, .monat = 5, .jahr = 1978};
```

- Wenn durch eine Initialisierung nicht alle Variablen eines Verbundtyps nicht explizit initialisiert werden, dann werden die verbleibenden Teile auf 0 gesetzt.

Komponentenzugriff bei Verbundtypen

- Auf die *Komponenten* eines Verbundtyps kann wie folgt zugegriffen werden:

```
struct datum gebdat = ...;

printf("%hu.%hu.%hu", gebdat.tag, gebdat.monat, gebdat.jahr);

struct datum *p = ...;

/* Zeiger zuerst dereferenzieren ... */
printf("%hu.%hu.%hu", (*p).tag, (*p).monat, (*p).jahr);
/* ... oder einfacher (und aequivalent) mit -> ... */
printf("%hu.%hu.%hu", p->tag, p->monat, p->jahr);
```

- Aufgrund der *Vorrang-Regeln* bei Operatoren ist $*p.tag$ äquivalent zu $*(p.tag)$ und nicht zu $(*p).tag$.
- Das Ausgabeformat *%hu* passt genau zu dem verwendeten Datentyp **unsigned short**.

- Die Elemente eines Verbundtyps können (beinahe) beliebigen Typs sein. Insbesondere ist es auch möglich, Verbundtypen ineinander zu verschachteln:

```
struct person {
   char* name;
   char* vorname;
   struct datum geburtsdatum;
};
```

- Wenn dann eine Variable *p* als **struct** *person p* vereinbart ist, dann kann wie folgt auf die Elemente zugegriffen werden:

```
p.name = ...;
p.vorname = ...;
p.geburtsdatum.tag = ...;
p.geburtsdatum.monat = ...;
p.geburtsdatum.jahr = ....;
```

struct.c

```
struct s {
   /* ... */
   struct s* p; /* Zeiger auf die eigene Struktur ist ok */
   /* struct s elem; */ /* nicht erlaubt! */
};

struct s1 {
   /* ... */
   struct s2* p;        /* Zeiger als Vorwaertsverweis ist ok */
   /* struct s2 elem;*/ /* nicht erlaubt! */
};

struct s2 {
   /* ... */
   struct s1* p;        /* Zeiger als Rueckwaertsverweis ok */
   struct s1 elem;      /* ok */
};
```

- Zeiger auf Verbundtypen können bereits verwendet werden, auch wenn die zugehörigen Strukturen noch nicht (bzw. nicht vollständig) deklariert sind.

struct1.c

```
#include <stdio.h>

struct datum {
   unsigned short tag, monat, jahr;
};

int main() {
   struct datum vorl_beginn = {15, 10, 2015};
   struct datum ueb_beginn = {16, 10, 2015};

   printf("vorher: %hu.%hu.%hu\n",
      vorl_beginn.tag, vorl_beginn.monat, vorl_beginn.jahr);

   vorl_beginn = ueb_beginn;

   printf("nachher: %hu.%hu.%hu\n",
      vorl_beginn.tag, vorl_beginn.monat, vorl_beginn.jahr);
}
```

- Variablen des gleichen Verbundtyps können einander auch zugewiesen werden.
- Dabei werden die einzelnen *Elemente* der Struktur jeweils *kopiert*.

struct2.c

```
/* Werteparameter-Semantik */
void ausgabe1(struct datum d) {
   printf("%hu.%hu.%hu\n", d.tag, d.monat, d.jahr);
}

/* Referenzparameter-Semantik (wirkt sich hier nicht aus) */
void ausgabe2(struct datum* d) {
   printf("%hu.%hu.%hu\n", d->tag, d->monat, d->jahr);
}
```

- Verbunde können als Werteparameter übergeben werden oder – durch die Verwendung von Zeigern – auch als Referenz-Parameter verwendet werden.

struct2.c

```
/* Werteparameter-Semantik: Verbund des Aufrufers aendert sich nicht */
void setJahr1(struct datum d, int jahr) {
   d.jahr = jahr;
}

/* Referenzparameter-Semantik erlaubt die Aenderung */
void setJahr2(struct datum* d, int jahr) {
   d->jahr = jahr;
}

int main() {
   struct datum start = {15, 10, 2015};

   ausgabe1(start);
   setJahr1(start, 2016);     /* keine Aenderung! */
   ausgabe2(&start);          /* aequivalent zu ausgabe1(...) */
   setJahr2(&start, 2016);    /* setzt das Jahr auf 2016 */
   ausgabe1(start);
}
```

Verbunde als Ergebnis von Funktionen

- Funktionen können als Ergebnistyp auch einen Verbundtyp verwenden.
- Hingegen ist die Rückgabe eines Zeiger auf eine lokale Variable (wie etwa einen lokalen Verbund) nicht zulässig.

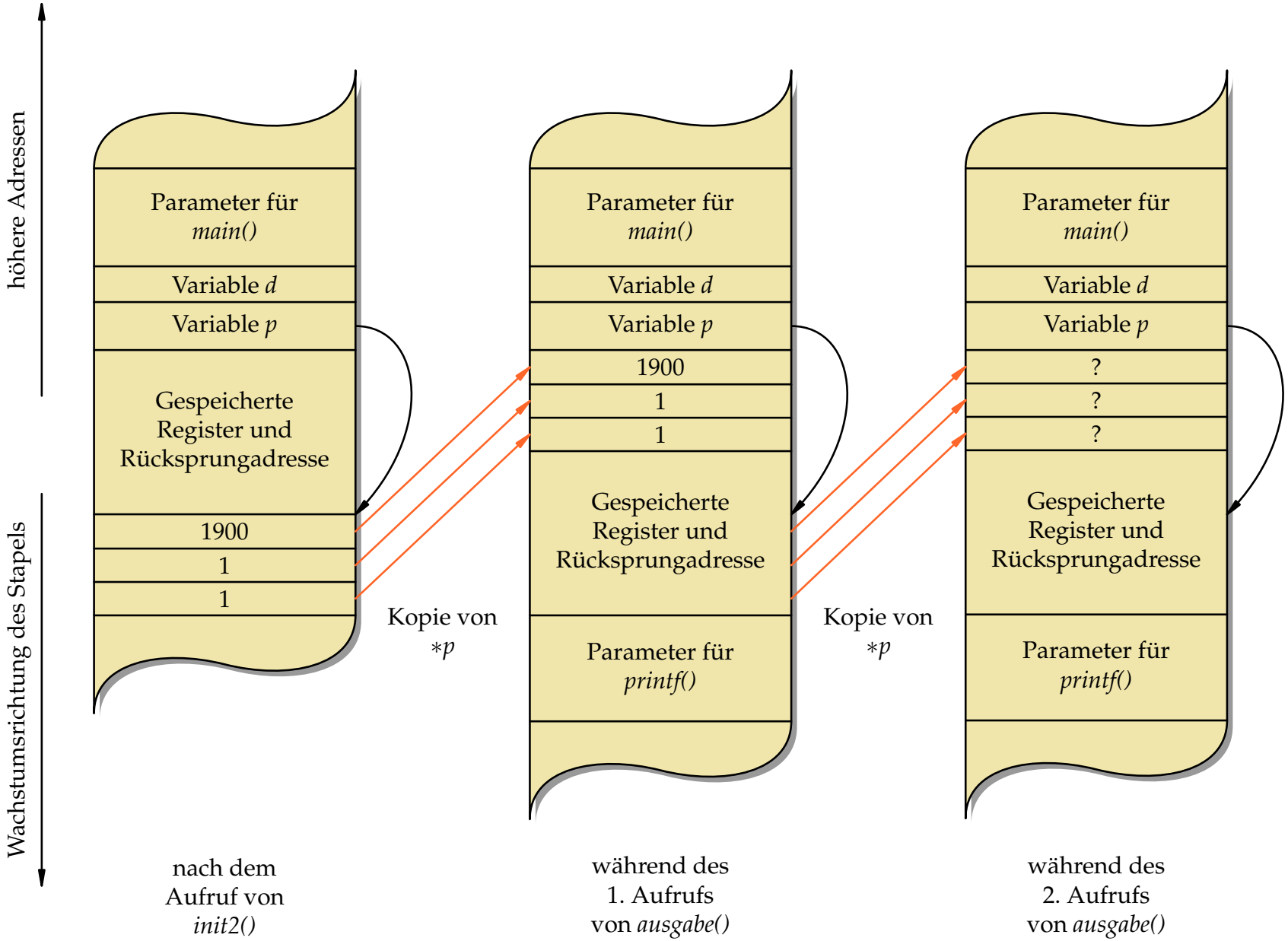
struct3.c

```
struct datum init1() {
   struct datum d = {1, 1, 1900};
   return d;   /* ok, denn es wird eine Kopie erzeugt */
}

struct datum* init2() {
   struct datum d = {1, 1, 1900};
   return &d;  /* nicht zulaessig, da Zeiger auf lokale Variable! */
}
```

- Seit einer Änderung im Juli 2014 gibt der *gcc* in offensichtlichen Fällen eine Warnung aus und liefert stattdessen einen 0-Zeiger zurück.

struct3.c

```
#include <stdio.h>

struct datum {
   unsigned short tag, monat, jahr;
};

void ausgabe(struct datum d) {
   printf("%hu.%hu.%hu\n", d.tag, d.monat, d.jahr);
}

/* init1() & init2() */

int main() {
   struct datum d;
   struct datum* p;

   d = init1();
   ausgabe(d);

   p = init2(); /* Zeiger auf Variable, die nicht mehr existiert! */
   ausgabe(*p); /* wenn's klappt ... dann ist das Glueck! */
   ausgabe(*p); /* sollte eigentliche dasselbe ausgeben :-( */
}
```


höhere Adressen
Wachstumsrichtung des Stapels
Parameter für *main()*
Variable *d*
Variable *p*
Gespeicherte Register und Rücksprungadresse
1900
1
1
Kopie von *p
Parameter für *main()*
Variable *d*
Variable *p*
1900
1
1
Gespeicherte Register und Rücksprungadresse
Parameter für *printf()*
Kopie von *p
Parameter für *main()*
Variable *d*
Variable *p*
?
?
?
Gespeicherte Register und Rücksprungadresse
Parameter für *printf()*
nach dem Aufruf von *init2()*
während des 1. Aufrufs von *ausgabe()*
während des 2. Aufrufs von *ausgabe()*

- Die Variable *d* in der Funktion *init2()* ist eine lokale Variable, die auf dem Laufzeit-Stapel für Funktionen (im Englischen *runtime stack* genannt) lebt.
- Sie existiert nur solange, wie diese Funktion ausgeführt wird. Danach wird dieser Speicherplatz evtl. anderweitig verwendet.
- Nach dem Aufruf von *init2()* ist zwar die Lebenszeit der Daten hinter *p* zwar vorbei, aber sie liegen typischerweise immer noch intakt auf dem Laufzeit-Stapel.
- Entsprechend werden beim ersten Aufruf von *ausgabe()* die Daten noch korrekt kopiert.
- Allerdings werden die von *p* referenzierten Daten dann während des ersten Aufrufs von *ausgabe()* möglicherweise überschrieben. Deswegen werden beim folgenden zweiten Aufruf von *ausgabe()* potentiell undefinierte Werte bei der Parameterübergabe kopiert.

$$
\begin{array}{lcl}
\langle\text{union-type-specifier}\rangle & \longrightarrow & \textbf{union}\ [\ \langle\text{identifier}\rangle\ ]\ \text{„\{“} \\
 & & \langle\text{struct-declaration-list}\rangle\ \text{„\}“} \\
 & \longrightarrow & \textbf{union}\ \langle\text{identifier}\rangle
\end{array}
$$

- Syntaktisch gleichen variante Verbunde den regulären Verbunden – es wird nur das Schlüsselwort **union** anstelle von **struct** verwendet.
- Im Vergleich zu den regulären Verbunden liegen alle Variablen eines varianten Verbunds an der gleichen Position im Speicher.

Wozu variante Verbunde?

In folgenden Szenarien kann der Einsatz von Verbunden sinnvoll sein:

- Variante Verbunde sparen Speicherplatz ein, wenn immer nur eine Variante benötigt wird. In diesem Falle muss (außerhalb des varianten Verbunds) ein Status verwaltet werden, der festhält, welche Variante gerade in Benutzung ist.
- Durch variante Verbunde sind zwei (oder mehr) Sichten durch verschiedene Datentypen auf ein gemeinsames Stück Speicher möglich, ohne dass hierfür jeweils umständliche Konvertierungen notwendig wären. Allerdings ist hier Vorsicht geboten, da dies sehr von der jeweiligen Plattform abhängen kann.

Beispiel eines varianten Verbunds

union.c

```
union IPAddr {
   unsigned int ip;
   unsigned char b[4];
};
```

- Alle Komponenten eines Verbunds liegen an der gleichen Speicheradresse.
- Der Speicherbedarf der größten Komponente bestimmt den Speicherbedarf für den gesamten varianten Verbund.
- In diesem Beispiel sind *ip* und *b* zwei Sichten auf das gleiche Stück Speicher: Einerseits kann eine IP-Adresse als ganze Zahl betrachtet werden, andererseits aber auch als Sequenz von vier Bytes.
- Der Unterschied zwischen *big* und *little endian* ist hier wieder relevant.

Beispiel eines varianten Verbunds

union.c

```
int main() {
   union IPAddr a;

   a.ip = 0x863c4205;  /* bel. IP-Adresse in int-Darst. zuweisen */
   /* Zugriff auf a ueber die Komponente ip */
   printf("%u [%x]\n", a.ip, a.ip);
   /* Zugriff auf a ueber die Komponente b */
   printf("%hhu.%hhu.%hhu.%hhu ", a.b[0], a.b[1], a.b[2], a.b[3]);
   printf("[%02hhx.%02hhx.%02hhx.%02hhx]\n",
      a.b[0], a.b[1], a.b[2], a.b[3]);
   puts("");
   printf("Speicherplatzbedarf: %zd\n", sizeof(a));

   puts("");   /* Anordung im Speicher analysieren */
   puts("Position im Speicher:");
   printf("a:    %p\n", &a);
   printf("ip:   %p\n", &a.ip);
   printf("b[0]: %p\n", &a.b[0]);
   printf("b[1]: %p\n", &a.b[1]);
   printf("b[2]: %p\n", &a.b[2]);
   printf("b[3]: %p\n", &a.b[3]);
}
```

Beispiel eines varianten Verbunds


```
clonard$ uname -m
sun4u
clonard$ gcc -Wall -std=gnu99 union.c -o union
clonard$ union
2252096005 [863c4205]
134.60.66.5 [86.3c.42.05]

Speicherplatzbedarf: 4

Position im Speicher:
a:    ffbff6ac
ip:   ffbff6ac
b[0]: ffbff6ac
b[1]: ffbff6ad
b[2]: ffbff6ae
b[3]: ffbff6af
clonard$
```

- Ausführung auf einer *big endian*-Maschine.

```
thales$ uname -m
i86pc
thales$ gcc -Wall -std=gnu99 union.c -o union
thales$ union
2252096005 [863c4205]
5.66.60.134 [05.42.3c.86]

Speicherplatzbedarf: 4

Position im Speicher:
a:    804729c
ip:   804729c
b[0]: 804729c
b[1]: 804729d
b[2]: 804729e
b[3]: 804729f
thales$
```

- Ausführung auf einer *little endian*-Maschine.

bigendian.c

```
#include <stdbool.h>
#include <stdio.h>
bool big_endian() {
   union {
      unsigned ui;
      unsigned char ch;
   } u = {1};
   return u.ch == 0;
}

int main() {
   if (big_endian()) {
      printf("big endian\n");
   } else {
      printf("little endian\n");
   }
}
```

- Gemäß 6.7.9, Punkt 17, lässt wird bei einer **union** nur die erstgenannte Variante initialisiert (hier also *ui*).

$$
\begin{array}{rcl}
\langle\text{typedef-name}\rangle & \longrightarrow & \langle\text{identifier}\rangle \\
\langle\text{storage-class-specifier}\rangle & \longrightarrow & \textbf{typedef} \\
 & \longrightarrow & \textbf{extern} \\
 & \longrightarrow & \textbf{static} \\
 & \longrightarrow & \textbf{auto} \\
 & \longrightarrow & \textbf{register}
\end{array}
$$

- Einer Deklaration kann das Schlüsselwort **typedef** vorausgehen. Dann wird der Name, der sonst ein Variablen- oder Funktionsname geworden wäre, stattdessen zu einem neu definierten Typnamen. Dieser Typname kann anschließend überall dort verwendet werden, wo die Angabe eines ⟨type-specifier⟩ zulässig ist.

Ein einfaches Beispiel einer Typdefinition


```
typedef int Laenge; /* Vereinbarung des eigenen Typnames "Laenge" */

/* ... */

Laenge i, j; /* Vereinbarung der Variablen i und j vom Typ Laenge */
```

- Hier ist *Laenge* zu einem Synonym für **int** geworden.
- Damit sind **int** *i*, *j*; und *Laenge i*, *j*; äquivalente Vereinbarungen.
- Hier bieten Typdefinitionen die Flexibilität, einen Typ an einer zentralen Stelle zu vereinbaren, um ihn dann bequem für das gesamte Programm verändern zu können.
- Das ist insbesondere sinnvoll bei der Verwendung numerischer Datentypen. Synonyme können auch zur Lesbarkeit beitragen, wenn besonders „sprechende“ Namen verwendet werden.

Typdefinitionen dienen der Vereinfachung


```
typedef char* CharPointer;
typedef int TenIntegers[10];
CharPointer cp1, cp2; // beide sind vom Typ char*
char* cp3, cp4; // cp4 hat nur den Typ char!
TenIntegers a, b; // beides sind Vektoren
int c[10], d; // d hat nur den Typ int!
```

- Typdefinitionen ermöglichen es, komplexere Typen in einen ⟨type-specifier⟩ zu integrieren, die sich sonst nur im Rahmen einer ⟨declaration⟩ formulieren liessen.
- Das betrifft insbesondere Zeiger und Arrays.

```
typedef struct datum {
   unsigned short tag, monat, jahr;
} datum;
datum geburtsdatum; // aequivalent zu struct datum geburtsdatum
datum heute, morgen;
```

- Bei Verbunden werden ebenfalls Typdefinitionen verwendet, um anschließend nur den Namen ohne das Schlüsselwort **struct** verwenden zu können.
- Die Verwendung von Typnamen aus Typdefinitionen bleibt – abgesehen von den syntaktischen Unterschieden – äquivalent zur Verwendung des ursprünglichen Datentyps. Entsprechend entsteht durch eine Typdefinition kein neuer Typ, der nicht mehr mit dem alten Typ kompatibel wäre.

- Durch die unglückliche Aufteilung von Typ-Spezifikationen in ⟨type-specifier⟩ (links stehend) und ⟨declarator⟩ (rechts stehend, sich um den Namen anordnend), werden komplexere Deklarationen rasch unübersichtlich.
- Die Motivation für diese Syntax kam wohl aus dem Wunsch, dass die Deklaration einer Variablen ihrer Verwendung gleichen solle.
- Entsprechend hilft es, sich bei komplexeren Deklarationen die Vorränge und Assoziativitäten der zugehörigen Operatoren in Erinnerung zu rufen.

Beispiel für die Analyse einer Deklaration


```
char* x[10];
```

- Der Vorrangtabelle lässt sich entnehmen, dass der []-Operator einen höheren Vorrang (16) im Vergleich zum $*$-Operator (15) hat.
- Entsprechend handelt es sich bei x um ein Array mit 10 Elementen des Typs Zeiger auf **char**.
- Im Einzelnen:

$x[10]$	Array mit 10 Elementen
$*\ x[10]$	Array mit 10 Zeigern
char$*\ x[10]$	Array mit 10 Zeigern auf Zeichen

Beispiel für die Analyse einer Deklaration


```
char (*x)[10];
```

- Wenn der $*$-Operator Vorrang erhalten soll, so ist in Kombination mit dem []-Operator eine Klammerung notwendig.
- Mit der Klammerung wird x als Zeiger auf ein Array mit 10 Elementen des Types **char** deklariert.
- Im Einzelnen:

$(*x)$	Zeiger
$(*x)[10]$	Zeiger auf ein Array mit 10 Elementen
char $(*x)[10]$	Zeiger auf ein Array mit 10 Elementen des Typs **char**

Zweites Beispiel für die Analyse einer Deklaration


```
int* (*(*x)())[5];
```

- Die Analyse beginnt hier wieder beim Variablennamen x in der Mitte der Deklaration:

$*x$	ein Zeiger
$(*x)()$	ein Zeiger auf eine Funktion
$*(*x)()$	ein Zeiger auf eine Funktion, die einen Zeiger liefert
$(*(*x)())[5]$	ein Zeiger auf eine Funktion, die einen Zeiger auf ein 5-elementiges Array liefert
$*\ (*(*x)())[5]$	ein Zeiger auf eine Funktion, die einen Zeiger auf ein 5-elementiges Array aus Zeigern liefert
int$*\ (*(*x)())[5]$	ein Zeiger auf eine Funktion, die einen Zeiger auf ein 5-elementiges Array aus Zeigern auf **int** liefert

Zweites Beispiel für die Analyse einer Deklaration


```
int* (*(*x)())[5];
```

- An zwei Stellen waren hier Vorränge relevant: Im zweiten Schritt war wesentlich, dass Funktionsaufrufe (Vorrangstufe 16) Vorrang haben vor der Dereferenzierung (Vorrangstufe 15) und im vierten Schritt hatte die Indizierung (Vorrangstufe 16) ebenfalls Vorrang vor der Dereferenzierung.
- Zusammenfassend:
  - [] und () haben einen höheren Rang als $*$.
  - [] und () assoziieren von *links nach rechts*, während $*$ von *rechts nach links* gruppiert.

Abhilfe mit Typdefinitionen


```
int* (*(*x)())[5];
```

- Lesbarer wird dies durch einen stufenweisen Aufbau mit Typdefinitionen:

```
typedef int* intp; // intp = Zeiger auf int
typedef intp intpa[5]; // intpa = Vektor mit 5 Zeigern auf int
typedef intpa* intpap; // intpap = Zeiger auf intpa
typedef intpap (*fp)(); // f = Zeiger auf eine Funktion, die intpap liefert
fp y;
```

Verwendung von Klammern bei Typdeklarationen 195

```
int (*x[10])();
```

- Klammern können verwendet werden, um die Operatoren anders zu gruppieren und damit den Typ entsprechend zu verändern.
- Hier ist x ein 10-elementiges Array von Zeigern auf Funktionen mit Rückgabewerten des Typs **int**. Im Einzelnen:

$x[10]$	x als 10-elementiges Array
$(*x[10])$	x als 10-elementiges Array von Zeigern
$(*x[10])()$	x als 10-elementiges Array von Zeigern auf Funktionen
int $(*x[10])()$	x als 10-elementiges Array von Zeigern auf Funktionen, mit Rückgabewerten des Typs **int**.

Beispiele für unzulässige Typdeklarationen

int *af*[]()	Array von Funktionen, die Rückgabewerte des Typs **int** liefern.
int *fa*()[]	Funktion, die ein Array von ganzen Zahlen liefert; hier wäre **int**∗ *fa*() akzeptabel gewesen.
int *ff*()()	Funktion, die eine Funktion liefert, welche wiederum **int** liefert.

- Anders als in Java gibt es keine automatisierte Speicherverwaltung, die unbenötigte Speicherflächen automatisch freigibt.
- Entsprechend muss in C Speicher nicht nur explizit belegt, sondern auch explizit freigegeben werden.
- Dies ist recht fehleranfällig.
- Hinzu kommt, dass Speicherflächen zunächst nicht mit Datentypen verbunden sind. Sie müssen aufgeteilt, verwaltet und Zeiger müssen je nach Bedarf konvertiert werden. Entsprechend fehlt hier die Sicherheit des Typsystems.
- Zum Ausgleich dafür lässt sich eine Speicherverwaltung in C selbst schreiben.

void∗ *calloc*(*size_t nelem*, *size_t* elsize)

Belegt Speicher für *nelem* Elemente der Größe *elsize* und initialisiert diesen mit 0. Im Erfolgsfall wird der Zeiger darauf geliefert, ansonsten 0.

void∗ *malloc*(*size_t size*)

Belegt Speicher für ein Objekt, das *size* Bytes benötigt. Im Erfolgsfall wird der Zeiger darauf geliefert, ansonsten 0.

void free(**void**∗ *ptr*)

Hier muss *ptr* auf eine zuvor belegte, jedoch noch nicht freigegebene Speicherfläche verweisen. Dann gibt *free* diese Fläche zur andersweitigen Nutzung wieder frei.

void∗ *realloc*(**void**∗ *ptr*, *size_t size*)

Versucht, die Größe der Speicherfläche, auf die *ptr* verweist, auf *size* Bytes anzupassen. Im Erfolgsfalle wird ein Zeiger auf die (möglicherweise neue) Speicherfläche zurückgeliefert, ansonsten 0.

void∗ *aligned_alloc*(*size_t* alignment, *size_t* size)

Neu eingeführt in C11, berücksichtigt Adresskanten.

reverse.c

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

/* lineare Liste ganzer Zahlen */
typedef struct element {
   int i;
   struct element* next;
} element;

int main() {
   element* head = 0;
   int i;

   /* Zahlen einlesen und in der Liste
      in umgekehrter Reihenfolge ablegen */
   while ((scanf("%d", &i)) == 1) {
      element* last = (element*) calloc(1, sizeof(element));
      if (last == 0) {
         fprintf(stderr, "out of memory!\n"); exit(1);
      }
      last->i = i; last->next = head; head = last;
   }
   /* Zahlen aus der Liste wieder ausgeben */
   while (head != 0) {
      printf("%d\n", head->i);
      head = head->next;
   }
}
```

reverse.c

```
element* last = (element*) calloc(1, sizeof(element));
if (last == 0) {
   fprintf(stderr, "out of memory!\n"); exit(1);
}
```

- *calloc* wird hier darum gebeten, für ein Element der Größe **sizeof**(*element*) Speicher zu belegen.
- Das entspricht *element last* = **new** *element*() in Java.
- Falls der gewünschte Speicher nicht belegt werden kann, wird ein 0-Zeiger zurückgeliefert. Entsprechend ist in C immer ein anschließender Test auf 0 erforderlich.
- Wenn es klappt, wird durch *calloc* die Speicherfläche mit Nullen initialisiert.
- *calloc* liefert den generischen Zeiger **void**∗ zurück. Dieser ist zu allen anderen Zeigern kompatibel. Der Cast-Operator (*element*∗) macht diese Typkonvertierung hier explizit.

reverse.c

```
element* last = (element*) malloc(sizeof(element));
if (last == 0) {
   fprintf(stderr, "out of memory!\n"); exit(1);
}
```

- Alternativ kann auch *malloc* aufgerufen werden.
- *malloc* erwartet jedoch nur die Gesamtgröße der belegenden Speicherfläche in Bytes und unterlässt die Initialisierung.
- Der Inhalt des neuen Objekts ist deswegen im Erfolgsfall vollkommen uninitialisiert.

```
void* my_calloc(size_t nelem, size_t elsize) {
   void* ptr = calloc(nelem, elsize);
   if (ptr) return ptr; /* alles OK */
   /* Fehlerbehandlung: */
   fprintf(stderr, "out of memory -- aborting!\n");
   /* Termination mit core dump */
   abort();
}
```

- Die Behandlung des Falls, dass ein 0-Zeiger zurückgeliefert wird, sollte nie vergessen werden.
- Wem das zu mühsam erscheint, kann diese Überprüfung in eine separate Funktion auslagern wie *my_calloc* in diesem Fall.

```
char bigmem[8192]; /* hoffentlich ausreichend */
char* bigmem_free = bigmem;

void* alloc(size_t nbytes) {
   if (bigmem_free + nbytes > bigmem + sizeof bigmem) {
      return 0; /* kein Speicher mehr da */
   }
   void* p = bigmem_free;
   bigmem_free += nbytes;
   return p;
}
```

- Eine triviale Speicherverwaltung, die nur Speicher aus einer Fläche abgibt, ohne freiwerdenden Speicher entgegennehmen zu können, ist einfach zu implementieren.
- Hier dient das große Array *bigmem* als Speicher-Reservoir.
- Der Zeiger *bigmem_free* zeigt auf den Bereich aus dem Array, der noch nicht vergeben ist. Alles davor wurde bereits vergeben. Zu Beginn wird der Zeiger auf den Anfang des großen Arrays gesetzt.

Triviale Speicherverwaltung


```
char bigmem[8192]; /* hoffentlich ausreichend */
char* bigmem_free = bigmem;

void* alloc(size_t nbytes) {
   if (bigmem_free + nbytes > bigmem + sizeof bigmem) {
      return 0; /* kein Speicher mehr da */
   }
   void* p = bigmem_free;
   bigmem_free += nbytes;
   return p;
}
```

- Wenn es in der Speicherverwaltung darum geht, Adressarithmetik zu verwenden, dann wird **char**∗ als Zeigertyp benutzt, da ein **char** die Größe eines Bytes hat, d.h. **sizeof**(**char**)== 1.
- (Theoretisch legt der Standard nur fest, dass ein Byte mindestens 8 Bits hat. Prinzipiell kann ein Byte größer sein, aber der Standard definiert fest, dass ein Byte im Sinne des Standards von **char** repräsentiert wird und dass **sizeof**(**char**)== 1 gilt.)
- Entsprechend ist **char** die kleinste adressierbare Größe.

badalign.c

```
char* cp = alloc(sizeof(char));
int* ip = alloc(sizeof(int));
if (cp && ip) {
   * cp = 'x'; * ip = 1;
} else {
   fprintf(stderr, "alloc failed\n");
}
```

- Was passiert, wenn wir über *alloc* unterschiedlich große Objekte anfordern?
- Dann zeigt *ip* auf einen ungeraden Wert!
- Manchen Plattformen macht das nichts aus, andere hingegen akzeptieren dies nicht:

```
theseus$ uname -a
SunOS theseus 5.10 Generic_147440-09 sun4u sparc SUNW,Sun-Fire-V490
theseus$ gcc -o badalign badalign.c
theseus$ ./badalign
Bus Error (core dumped)
theseus$
```

Alignment

- Manche Architekturen erlauben keinen Zugriff auf größere Datentypen wie etwa **int** oder **double**, wenn sie an irgendwelchen ungeraden Adressen liegen.
- Die SPARC-Architektur beispielsweise besteht darauf, dass **int**-Variablen auf durch vier teilbaren Adressen liegen und **long long int** oder **double** auf durch acht teilbare Adressen.
- Andere Architekturen sind diesbezüglich großzügiger (wie etwa die x86-Plattform), aber mitunter ist die Zugriffszeit größer, wenn keine entsprechend ausgerichtete Adresse benutzt wird.
- Wenn es wegen eines Alignment-Fehlers zu einem Absturz kommt, wird das als „Bus Error“ bezeichnet (mit Verweis auf den Systembus, der sich geweigert hat, auf ein Objekt an einer nicht ausgerichteten Adresse zuzugreifen).
- Eine Speicherverwaltung muss darauf Rücksicht nehmen.

alignment.c

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

int main() {
   printf("alignment for char: %zu\n", alignof(char));
   printf("alignment for int: %zu\n", alignof(int));
   printf("alignment for long long int: %zu\n", alignof(long long int));
   printf("alignment for double: %zu\n", alignof(double));
   printf("alignment for double [10]: %zu\n", alignof(double [10]));
   printf("alignment for long double: %zu\n", alignof(long double));
}
```

- Seit C11 gibt es in C den **alignof**-Operator. Zu beachten ist, dass dieser den **#include** <stdalign.h>-Header benötigt.

```
theon$ ./alignment
alignment for char: 1
alignment for int: 4
alignment for long long int: 8
alignment for double: 8
alignment for double [10]: 8
alignment for long double: 16
theon$
```

Alignment

- Für Alignments wird ebenfalls der Datentyp *size_t* verwendet.
- Alignments sind immer Zweierpotenzen.
- Die Datentypen **char**, **signed char** und **unsigned char** haben das niedrigste Alignment (typischerweise 1).
- *alignof*(*max_align_t*) liefert das maximale Alignment, das bei einer Plattform notwendig sein kann.
- Größere Alignments können zur Verbesserung der Performance in Betracht gezogen werden, etwa wenn Datenstrukturen auf Cache-Line-Kanten ausgerichtet werden sollen.

maxalign.c

```
#include <complex.h>
#include <stdalign.h>
#include <stddef.h>
#include <stdio.h>

int main() {
   printf("alignof(max_align_t) = %zu\n", alignof(max_align_t));
   printf("alignof(long double complex) = %zu\n",
      alignof(long double complex));
   printf("sizeof(long double complex) = %zu\n",
      sizeof(long double complex));
}
```

- Das maximale Alignment ergibt sich typischerweise durch die entsprechende Anforderung des größten elementaren Datentyps. Dies sind **long double** bzw. – sofern komplexe Zahlen unterstützt werden – **long double complex**.

Maximales Alignment


```
theon$ maxalign
alignof(max_align_t) = 16
alignof(long double complex) = 16
sizeof(long double complex) = 32
theon$
```

```
theseus$ maxalign
alignof(max_align_t) = 8
alignof(long double complex) = 8
sizeof(long double complex) = 32
theseus$
```

alignas.c

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

int main() {
   alignas(32) int int32_a;
   alignas(32) int int32_b;
   printf("sizeof(int32) =  %2zu\n", sizeof(int32_a));
   printf("alignof(int32) = %2zu\n", alignof(int32_a));
   printf("&int32_a = %p\n", &int32_a);
   printf("&int32_b = %p\n", &int32_b);
}
```

- Mit **alignas** kann bei einer Deklaration ein gewünschtes Alignment spezifiziert werden (nicht zulässig für Typdeklarationen).
- Zulässig sind 0 (dann bleibt es bei der Voreinstellung), ein natürlich vorkommendes Alignment größer oder gleich der Voreinstellung oder ein Alignment jenseits dem Maximum, wenn dies vom Übersetzer in dem Kontext unterstützt wird.

Berücksichtigung des Alignments


```
void* alloc(size_t nbytes, size_t align) {
   char* p = bigmem_free;
   p = (char*) (((uintptr_t) p + align - 1) & ~(align - 1));
   if (p + nbytes > bigmem + sizeof bigmem) {
      return 0; /* kein Speicher mehr da */
   }
   bigmem_free = p + nbytes;
   return p;
}
```

- Das Problem kann durch einen zusätzlichen Parameter mit der gewünschten Ausrichtung gelöst werden. Hier: *align*.
- Der Zeiger *p* wird dann auf die nächste durch *align* teilbare Adresse gesetzt unter der Annahme, dass *align* eine Zweierpotenz ist.
- Alternativ könnte auch immer *alignof*(*max_align_t*) zugrunde gelegt werden – so wird es von *malloc* und *calloc* umgesetzt.

```
p = (char*) (((uintptr_t) p + align - 1) & ~(align - 1));
```

$(uintptr_t)p$	konvertiere p in eine ganze Zahl
$align - 1$	setzt die n niedrigswertigen Bits, wobei $n = log_2(align)$
$\sim(align - 1)$	bildet das Komplement davon, d.h. alle Bits außer den n niedrigstwertigen sind gesetzt
$\& \sim(align - 1)$	blendet die n niedrigswertigen Bits weg
$(uintptr_t)p + align - 1$	die Addition stellt sicher, dass das Resultat nicht kleiner wird durch das Wegblenden von Bits

- Das setzt voraus, dass *align* eine Zweierpotenz ist. Davon ist bei Alignment-Größen immer auszugehen.
- *uintptr_t* aus <*stdint.h*> ist ein **unsigned**-Typ, der groß genug ist, um Zeigerwerte aufzunehmen.

Berücksichtigung des Alignments


```
#define NEW(T) alloc(sizeof(T), alignof(T))

char* cp = NEW(char);     printf("cp =   %p\n", cp);
int* ip = NEW(int);       printf("ip =   %p\n", ip);
char* cp2 = NEW(char);    printf("cp2 =  %p\n", cp2);
char* cp3 = NEW(char);    printf("cp3 =  %p\n", cp3);
double* dp = NEW(double); printf("dp =   %p\n", dp);
long double* ldp = NEW(long double); printf("ldp =  %p\n", ldp);
```

```
theon$ goodalign
cp =   600ce0
ip =   600ce4
cp2 =  600ce8
cp3 =  600ce9
dp =   600cf0
ldp =  600d00
theon$
```

- *calloc*, *malloc* und *realloc* haben jedoch keine *align*-Parameter.
- In der Praxis wird die Größe des gewünschten Datentyps und die maximal vorkommende Alignment-Größe *alignof*(*max_align_t*) in Betracht gezogen.
- Beginnend ab C11 steht auch

  void∗ *aligned_alloc*(*size_t alignment*, *size_t size*);

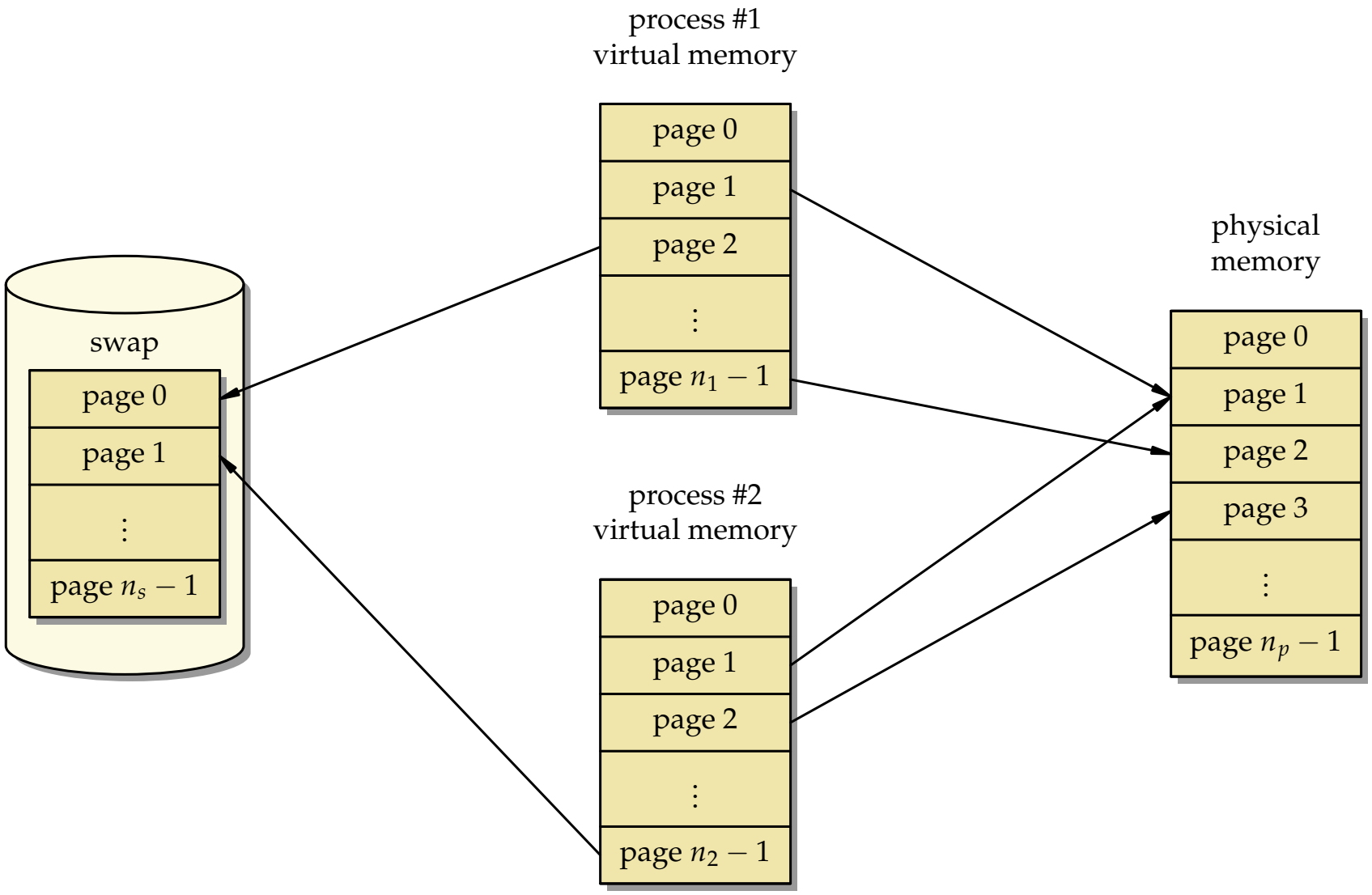
  zur Verfügung. Hierbei muss *alignment* eine vom System tatsächlich verwendete Alignment-Größe sein und *size* ein Vielfaches von *alignment*.
- POSIX bietet auch noch

  int *posix_memalign*(**void**∗∗ *memptr*, *size_t alignment*, *size_t size*);

  an, wobei *alignment* eine Zweierpotenz sein muss und im Erfolgsfalle (Return-Wert gleich 0) der Zeiger hinter *memptr* auf die belegte Speicherfläche gesetzt wird.

Virtueller Adressraum

- Wenn ein Prozess unter UNIX startet, wird zunächst nur Speicherplatz für den Programmtext (also den Maschinen-Code), die globalen Variablen, die Konstanten (etwa die von Zeichenketten) und einem Laufzeitstapel (Stack) angelegt.
- All dies liegt in einem sogenannten virtuellen Adressraum (typischerweise mit 32- oder 64-Bit-Adressen), den das Betriebssystem einrichtet.
- Die Betonung liegt auf virtuell, da die verwendeten Adressen nicht den physischen Adressen entsprechen, sondern dazwischen eine durch das Betriebssystem konfigurierte Abbildung durchgeführt wird.
- Diese Abbildung wird nicht für jedes einzelne Byte definiert, sondern für größere Einheiten, die sogenannten Kacheln (*page*).


process #1
virtual memory
page 0
page 1
page 2
⋮
page $n_1 - 1$
swap
page 0
page 1
⋮
page $n_s - 1$
physical
memory
page 0
page 1
page 2
page 3
⋮
page $n_p - 1$
process #2
virtual memory
page 0
page 1
page 2
⋮
page $n_2 - 1$

getpagesize.c

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

int main() {
   printf("page size = %d\n", getpagesize());
}
```

- Die Größe der Kacheln ist plattformabhängig und kann mit Hilfe des Systemaufrufs *getpagesize()* ermittelt werden.

```
theon$ uname -a
SunOS theon 5.11 11.3 i86pc i386 i86pc
theon$ getpagesize
page size = 4096
theon$
```

```
theseus$ uname -a
SunOS theseus 5.10 Generic_147440-09 sun4u sparc SUNW,Sun-Fire-V490
theseus$ getpagesize
page size = 8192
theseus$
```

- Sei $[0, 2^n - 1] \subset \mathbb{N}_0$ der virtuelle Adressraum, $P = 2^m$ die Größe einer Kachel mit $m < n$ und

$$M : [0, 2^{n-m} - 1] \to [0, 2^{n-m} - 1] \cup \{\text{invalid}\}$$

  die Funktion, die eine virtuelle Kachelnummer in die korrespondierende physische Kachelnummer bzw. in „invalid“ abbildet.

- Dann lässt sich folgendermaßen aus der virtuellen Adresse a_v die zugehörige physische Adresse a_p ermitteln:

$$a_p = \begin{cases} \text{invalid} & \text{falls } M(a_v \textbf{ div } P) = \text{invalid} \\ M(a_v \textbf{ div } P) \times P + a_v \textbf{ mod } P & \text{sonst} \end{cases}$$

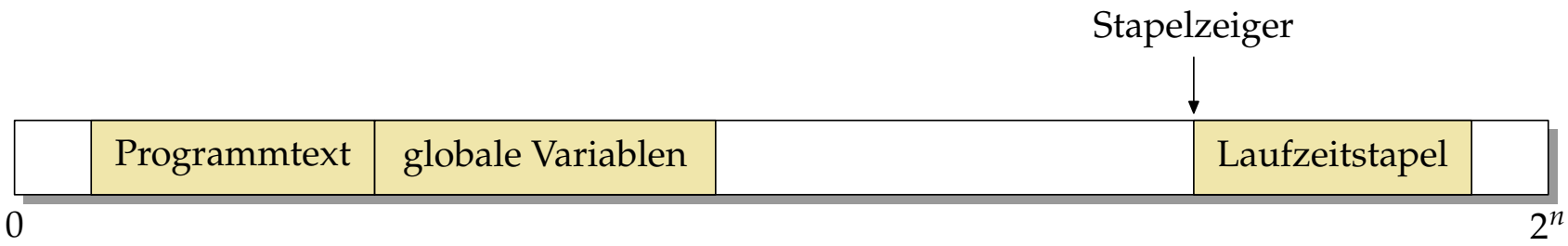
- Eine Abbildung zu „invalid“ führt zu einer Trap, d.h. der Betriebssystemkern übernimmt dann die Kontrolle und erzeugt entweder *SIGSEGV*-Signal (*segmentation violation*) oder passt M an durch das Nachladen von Kacheln von der Platte oder der automatisch erfolgenden Stackvergrößerung.

Abbildung virtueller Adressen

- Die Funktion M wird von der zur Hardware gehörenden MMU (*memory management unit*) implementiert in Abhängigkeit von Tabellen, die das Betriebssystem konfigurieren kann.
- Die Abbildung erfolgt an der Stelle der Speicherhierarchie zwischen der CPU, den einzelnen Cache-Ebenen und dem Hauptspeicher, bei dem zwischen virtuellen und physischen Adressen gewechselt wird.
- Die CPU selbst und die Cache-Ebenen, die nur einer CPU (bzw. nur einem Core) zugeordnet sind, arbeiten mit virtuellen Adressen, während bei gemeinsam genutzten Cache-Ebenen einheitliche physische Adressen notwendig sind.
- Entsprechend erfolgt die Abbildung genau an diesem Übergang, das ist etwa auf den PCs im Poolraum E.44 zwischen L2 und L3.
- Für die zu M gehörende Tabelle gibt es ebenfalls einen Cache, den sogenannten *translation lookaside buffer*.

- Eine erste Aufteilung des Adressraums mit der Platzierung der Segmente mit dem Programmtext und den globalen Variablen wird durch den *ld* vorgenommen.
- Beim Start des Programms wird ein Bereich für den Stack reserviert.
- Der Laufzeitlader bildet gemeinsam genutzte Bibliotheken (*shared libraries*) in den Adressraum ab.
- All dies hinterlässt einen Adressraum, in dem einzelne Bereiche genutzt werden.
- Für weite Teile des Adressraums bleibt M jedoch undefiniert. Ein Versuch, über einen entsprechenden Zeiger zuzugreifen, führt dann zu einem Abbruch des Programms (*segmentation violation*).

Aufteilung des Adressraums



- Wie der Adressraum zu Beginn belegt wird, liegt in der Freiheit des Betriebssystems bzw. des *ld*.
- Fast immer bleibt die Adresse 0 unbelegt, damit Versuche, auf einen 0-Zeiger zuzugreifen, zu einem Fehler führen. Ebenso bleibt der ganz hohe Bereich bei 2^n frei.
- Der Programmtext und die globalen Variablen liegen normalerweise in der Nähe, sind aber unterschiedlich konfiguriert (*read-only/executable* und *read/write*).
- Der Laufzeitstapel (Stack) wird davon getrennt konfiguriert, damit er genügend Platz hat zu wachsen, typischerweise von hohen zu niedrigen Adressen.

end.c

```
#include <stdio.h>
extern const void etext; extern const void edata; extern const void end;
int global[1000] = {1}; // some global initialized data
int main() { char local;
   printf("main ->   %16p\n", main); printf("etext ->  %16p\n", &etext);
   printf("global -> %16p\n", global); printf("edata ->  %16p\n", &edata);
   printf("end ->    %16p\n", &end); printf("local ->  %16p\n", &local);
}
```

- Bei traditionellen *ld*-Konfigurationen werden die Symbole *etext*, *edata* und *end* definiert, die entsprechend auf das Ende des Programmtexts, das Ende der initialisierten Daten und das Ende der uninitialisierten Daten verweisen.

```
theon$ end
main ->              4007c2
etext ->             400856
global ->            600bc0
edata ->             601b60
end ->               601b68
local ->  ffff80ffbfffe6bf
theon$
```

```
theon$ pmap 12175
12175:  endpause
0000000000400000          4K r-x----  /.../endpause
0000000000600000          8K rw-----  /.../endpause
FFFF80FFBF460000       1824K r-x----  /lib/amd64/libc.so.1
FFFF80FFBF638000         68K rw-----  /lib/amd64/libc.so.1
FFFF80FFBF649000          8K rw-----  /lib/amd64/libc.so.1
FFFF80FFBF68D000        352K r-x----  /lib/amd64/ld.so.1
FFFF80FFBF7B0000         24K rwx----    [ anon ]
FFFF80FFBF7C0000         64K rw-----    [ anon ]
FFFF80FFBF7D8000         12K r--s---    [ anon ]
FFFF80FFBF7E5000         16K rwx----  /lib/amd64/ld.so.1
FFFF80FFBF7E9000          4K rwx----  /lib/amd64/ld.so.1
FFFF80FFBF7EB000          4K r--s---    [ anon ]
FFFF80FFBF7FA000          4K r--s---    [ anon ]
FFFF80FFBFFFC000         16K rw-----    [ stack ]
         total         2408K
theon$
```

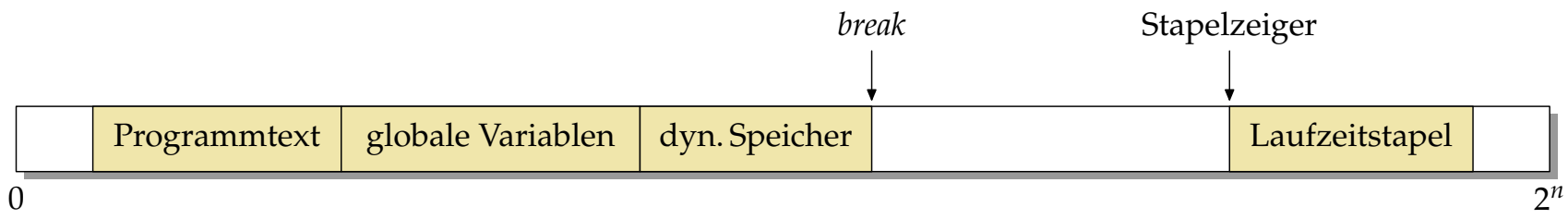
- Solange ein Prozess noch läuft, kann auf Solaris mit Hilfe des *pmap*-Programms der virtuelle Adressraum aufgelistet werden.
- Links steht in Hex jeweils die Anfangsadresse, dann in dezimal die Zahl der belegten Kilobyte, dann die Zugriffsrechte (r = *read*, w = *write*, x = *executable*) und schließlich, sofern vorhanden, die in den Adressraum abgebildete Datei.

Zur vierten Spalte:

.../endpause	Das ausführbare Programm, das gestartet wurde.
/lib/amd64/libc.so.1	Dynamisch ladbare C-Bibliothek, „so“ steht dabei für *shared object*, die „1“ dient als Versionsnummer.
/lib/amd64/ld.so.1	Dynamisches Ladeprogramm (war notwendig, um */lib/amd64/libc.so.1* zu laden).
[stack]	Bis zu einem gewissen Limit automatisch wachsender Stack.
[anon]	Durch die Speicherverwaltung belegter Speicher.

- Jede dynamische Speicherverwaltung benötigt einen Weg, mehr Speicher vom Betriebssystem anzufordern und diesen in einen bislang ungenutzten Bereich des virtuellen Adressraums abzubilden.
- Dafür gibt es im POSIX-Standard zwei Systemaufrufe:
  - *sbrk* – der traditionelle Ansatz: einfach, aber nicht flexibel
  - *mmap* – sehr flexibel, aber auch etwas komplizierter
- Gearbeitet wird in jedem Fall mit ganzen Kacheln.
- Eine Rückgabe von Kacheln an das Betriebssystem scheitert normalerweise an der Fragmentierung. Bei C findet das normalerweise nicht statt, d.h. der belegte Speicher wächst selbst bei einem gleichbleibenden Speichernutzungsumfang dank der zunehmenden Fragmentierung langsam aber stetig.

Der Break



- Der Break ist eine vom Betriebssystem verwaltete Adresse, die mit Hilfe der Systemaufrufe *brk* und *sbrk* manipuliert werden kann.
- *brk* spezifiziert die absolute Position des Break, *sbrk* verschiebt diese relativ.
- Zu Beginn zeigt der Break auf den Anfang des Heaps, konventionellerweise liegt dieser hinter den globalen Variablen.
- Durch das Verschieben des Breaks zu höheren Adressen kann dann Speicher belegt werden.

alloc auf Basis des Breaks

trivial-alloc-break.c

```
void* alloc(size_t nbytes, size_t align) {
   static char* mem_break = 0;
   static char* mem_pos = 0;
   static int pagesize = 0;
   if (!pagesize) {
      mem_break = sbrk(0);
      mem_pos = mem_break;
      pagesize = getpagesize();
      size_t remainder = (uintptr_t) mem_break % pagesize;
      if (remainder) {
         char* oldbreak = sbrk(pagesize - remainder);
         if (oldbreak != mem_break) {
            return 0;
         }
         mem_break += pagesize - remainder;
      }
   }
   char* p = mem_pos;
   p = (char*) (((uintptr_t) p + align - 1) & ~(align - 1));
   if (p + nbytes > mem_break) {
      size_t increment = (nbytes + pagesize - 1) & ~(pagesize - 1);
      char* oldbreak = sbrk(increment);
      if (oldbreak == (char*)-1) {
         return 0;
      }
      mem_break = oldbreak + increment;
   }
   mem_pos = p + nbytes;
   return p;
}
```

trivial-alloc-break.c

```
if (!pagesize) {
   mem_break = sbrk(0);
   mem_pos = mem_break;
   pagesize = getpagesize();
   size_t remainder = (uintptr_t) mem_break % pagesize;
   if (remainder) {
      char* oldbreak = sbrk(pagesize - remainder);
      if (oldbreak != mem_break) {
         return 0;
      }
      mem_break += pagesize - remainder;
   }
}
```

- Der Break liegt nicht notwendigerweise auf einer Kachelkante. Es ist aber zu Beginn sinnvoll, *mem_break* auf eine solche zu setzen, denn dieser Bereich steht ohnehin bereits zur Verfügung.

trivial-alloc-break.c

```
if (p + nbytes > mem_break) {
   size_t increment = (nbytes + pagesize - 1) & ~(pagesize - 1);
   char* oldbreak = sbrk(increment);
   if (oldbreak == (char*)-1) {
      return 0;
   }
   mem_break = oldbreak + increment;
}
mem_pos = p + nbytes;
return p;
```

- Wenn der vorhandene Bereich nicht genügt, wird der Break herausgeschoben – um ein Vielfaches der Kachelgröße.
- Wie bei allen Systemaufrufen üblich, wird bei Fehlern -1 zurückgegeben selbst wenn es sich um Zeiger handelt.

alloc auf Basis von *mmap*

trivial-alloc-mmap.c

```
void* alloc(size_t nbytes, size_t align) {
   static char* mem_end = 0;
   static char* mem_pos = 0;
   static int pagesize = 0;
   if (!pagesize) {
      pagesize = getpagesize();
   }
   char* p = mem_pos;
   p = (char*) (((uintptr_t) p + align - 1) & ~(align - 1));
   if (p + nbytes > mem_end) {
      size_t len = (nbytes + pagesize - 1) & ~(pagesize - 1);
      char* free_space = mmap(0, len,
         PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON, -1, 0);
      if (free_space == MAP_FAILED) {
         return 0;
      }
      if (mem_end != free_space) {
         mem_pos = free_space;
         p = free_space;
      }
      mem_end = free_space + len;
   }
   mem_pos = p + nbytes;
   return p;
}
```

alloc auf Basis von *mmap*

trivial-alloc-mmap.c

```
char* free_space = mmap(0, len,
   PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON, -1, 0);
```

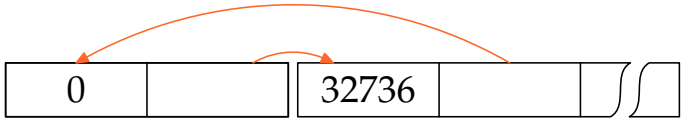
- Beim ersten Parameter kann *mmap* eine Adresse vorgeschlagen werden. Bei 0 wird das ganz dem System überlassen.
- Der zweite Parameter spezifiziert die Länge (sollte ein Vielfaches der Kachelgröße sein).
- Dann folgen die Zugriffsrechte. *PROT_READ* und *PROT_WRITE* geben Lese- und Schreibrechte, gewähren aber keine Ausführungsrechte.
- *MAP_PRIVATE* bedeutet, dass wir diesen Speicherbereich mit niemanden teilen.
- Mit *MAP_ANON* wird implizit */dev/zero* als abzubildende Datei gewählt. Dies gehört nicht zum Umfang von POSIX, wird aber weitgehend unterstützt (einschließlich Linux und Solaris).
- Die letzten beiden Parameter spezifizieren einen Dateideskriptor und einen Offset. In Verbindung mit *MAP_ANON* ist hier -1 und 0 anzugeben.

- Im folgenden wird eine sehr einfache Speicherverwaltung vorgestellt, die
  - das Belegen und Freigeben von Speicher unterstützt,
  - freigegebene Speicherflächen wieder zur Verfügung stellen kann und auch
  - in der Lage ist, mehrere hintereinander freigegebene Speicherflächen zu einer größeren Freifläche zusammenzufügen, um Fragmentierung zu vermeiden.
- Der vorgestellte Algorithmus ist in der Literatur bekannt als *circular first fit*. Eine ähnliche Fassung wurde bereits im ersten C-Buch von Kernighan und Ritchie vorgestellt.

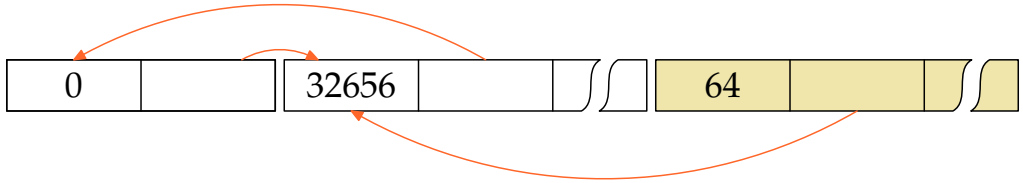
alloc.c

```
typedef struct memnode {
   size_t size;
   struct memnode* next;
} memnode;
```

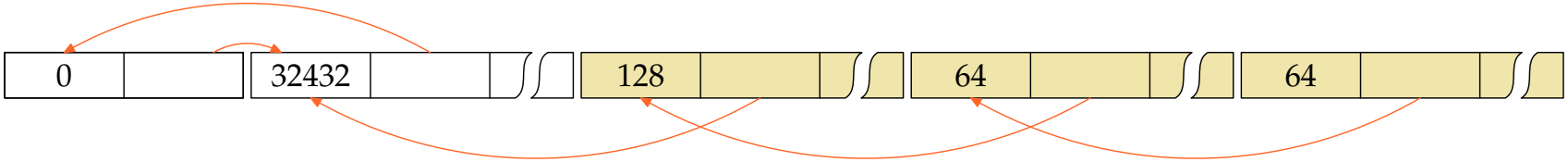
- Allen freien oder belegten Speicherflächen geht ein Verwaltungsobjekt des Typs *memnode* voraus.
- *size* gibt die Größe der Speicherfläche an, die diesem Verwaltungsobjekt unmittelbar folgt, jedoch ohne Einberechnung des Speicherbedarfs für das Verwaltungsobjekt
- *next* verweist
  - bei freien Speicherflächen auf das nächste Verwaltungsobjekt im Ring freier Speicherflächen
  - bei belegten Speicherflächen zum unmittelbar vorangehenden Speicherelement, egal ob dieses frei oder belegt ist.



- Alle freien Speicherflächen sind in einem Ring organisiert.
- Ein Ring ist bei dem gewählten Algorithmus *circular first fit* notwendig, weil nicht immer von Anfang an gesucht wird, sondern dort die Suche begonnen wird, wo sie zuletzt endete.
- Damit sich der Ring nicht auflöst, wenn alle zur Verfügung stehenden Speicherflächen vergeben sind, gibt es ein spezielles Ring-Element, das nur aus einem Verwaltungsobjekt besteht, aber keinen eigentlichen Speicherplatz anbietet.
- Das Diagramm zeigt zwei Ringelemente. Links ist das spezielle Element (mit $size = 0$) und rechts ein Element, das noch 32736 Bytes frei hat.

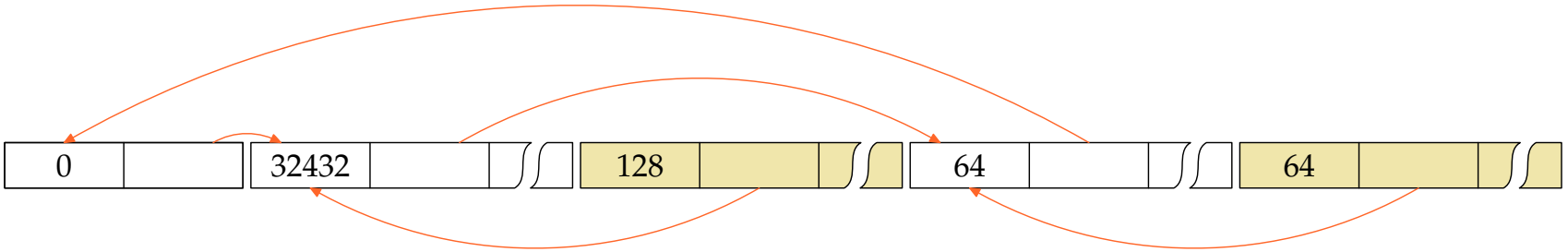


- Wenn nun 64 Bytes belegt werden sollen, wird nach einem Element im Ring der freien Speicherflächen gesucht, das mindestens 64 Bytes anbietet. In diesem Beispiel gab es nur das ganz große Element.
- Da noch Speicher übrigbleibt, wird das Element geteilt: Das Ende wird für die zu vergebende Speicherfläche Platz reserviert mitsamt einem Verwaltungsobjekt und bei dem entsprechenden freien Element wird *size* verkleinert, von 32736 auf 32656 (unter der Annahme, dass ein Verwaltungsobjekt 16 Bytes belegt und somit $64 + 16 = 80$ Bytes benötigt wurden).
- Bei dem Verwaltungsobjekt für die belegte Speicherfläche verweist der *next*-Zeiger auf das im Speicher unmittelbar davorliegende Element; das ist hier noch das Element aus dem Ring der freien Speicherflächen.



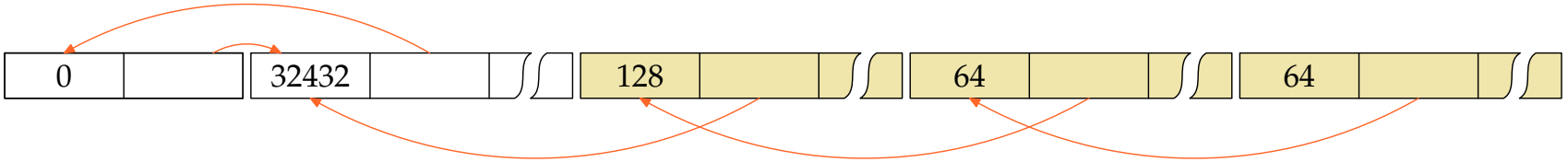
- Inzwischen wurden zwei weitere Speicherflächen belegt, zuerst noch einmal mit 64, dann mit 128 Bytes.
- Diese wurden allesamt dem großen Ringelement der freien Speicherflächen entnommen.
- Zu beachten ist, dass die Verwaltungsobjekte der belegten Speicherflächen jeweils auf das im Speicher unmittelbar davorliegende Element verweisen, egal ob dies frei ist oder nicht. Auf diese Weise ist es bei einer Freigabe recht einfach, ein freigegebenes Element mit einem bereits freien Element in der unmittelbaren Nachbarschaft zu vereinigen.

Freigabe einer Speicherfläche



- Wenn eine belegte Speicherfläche freigegeben wird, wird ausgehend von dem freiwerdenden Element solange die Kette der davorliegenden Elemente verfolgt, bis das erste Ring-Element vorgefunden wird, das eine freie Speicherfläche repräsentiert.
- Die freigegebene Speicherfläche wird unmittelbar dahinter eingehängt.
- Bei dem Ring der freien Speicherflächen bleibt so immer die Ordnung entsprechend der Lage im Speicher erhalten. Nur auf diese Weise ist es später möglich, benachbarte freie Flächen wieder zu größeren freien Flächen zusammenzulegen.

Wie wird eine freie Speicherflächer erkannt?



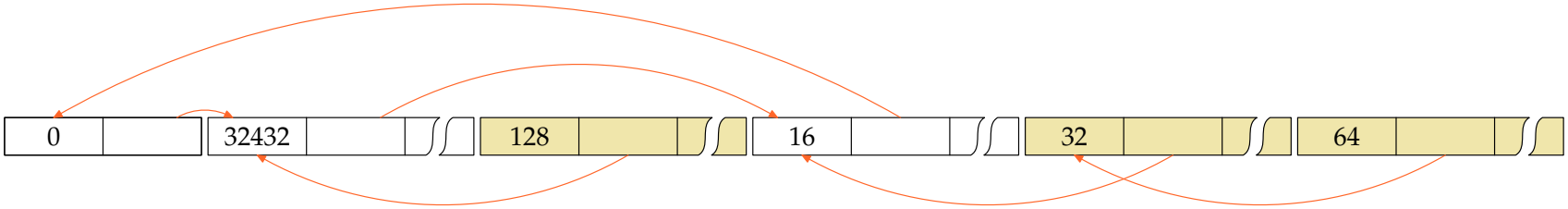
- Wenn bei einer Freigabe das nächste vorangehende freie Element gesucht wird, müssen wir unterscheiden können, ob ein Element frei oder belegt ist.
- Eine Möglichkeit wäre es, etwa bei *size* das niedrigstwertige Bit entsprechend zu setzen. Die Größe muss immer die Alignment-Anforderungen berücksichtigen und entsprechend darf eine Größe nie ungerade sein.
- In diesem einfachen Beispiel mit nur einem großen Speicherblock und einem Spezialelement geht es aber auch ohne diesen Trick...

Wie wird eine freie Speicherflächer erkannt?

alloc.c

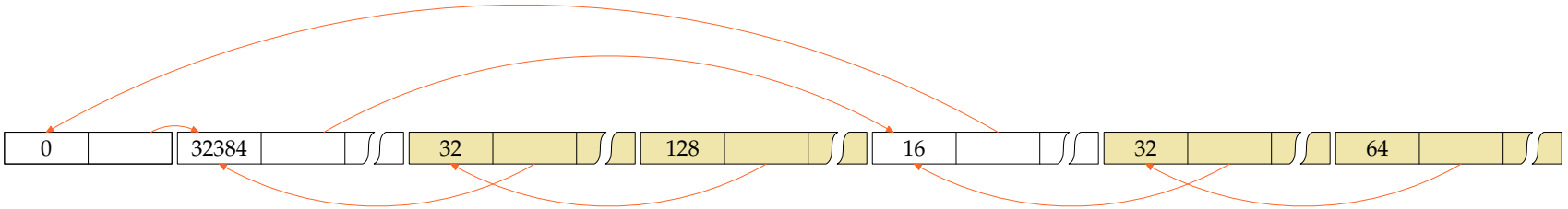
```
bool is_allocated(memnode* ptr) {
   if (ptr == root) return false;
   if (ptr->next > ptr) return false;
   if (ptr->next != root) return true;
   if (root->next > ptr) return true;
   return root->next == root;
}
```

- Das Spezialelement (nennt sich hier *root*) ist immer frei.
- Wenn der Zeiger nach vorne (d.h. zu einer größeren Adresse) weist, dann ist das Element frei.
- Wenn das alles nicht zutrifft und der Zeiger nicht auf das Spezialelement *root* verweist, ist es belegt.
- Wenn *next* auf das Spezialelement *root* verweist, gibt es zwei Fälle:
  - Es ist belegt und das Element liegt unmittelbar hinter dem Spezialelement (am Anfang des großen Blocks) oder
  - es handelt sich um das freie Element mit der höchsten Adresse.

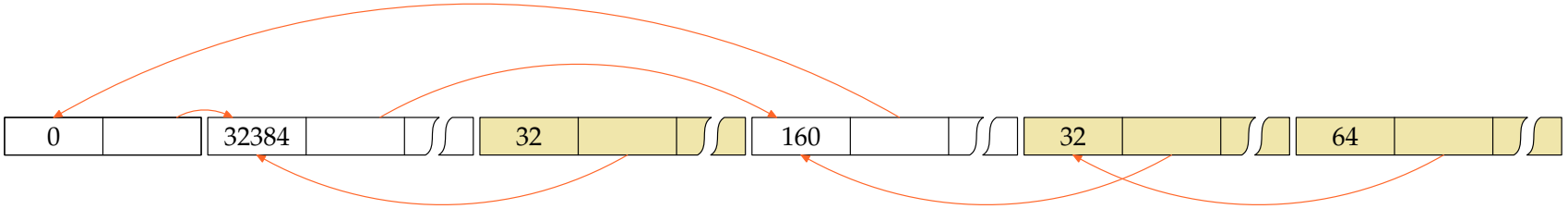


- Wenn nach freien Elementen immer beginnend von dem Spezialelement (*root*) aus gesucht wird, tendiert die Liste der freien Elemente dazu, zu Beginn nur ganz winzige Reste anzubieten, so dass die größeren freien Elemente erst ganz hinten zu finden sind.
- Deswegen wird beim *circular first fit*-Algorithmus die Suche dort fortgesetzt, wo wir zuletzt waren. Und entsprechend dem *first fit* wird die erste Speicherfläche akzeptiert, die genügend Speicherplatz anbietet.
- Das Diagramm zeigt die Situation, wenn 32 Bytes angefordert wurden. Das führte zur Aufspaltung des zuletzt frei gewordenen Elements mit 64 Bytes.

Das Wandern im Ring



- Und wenn erneut Speicher belegt wird, dann beginnt die Suche beim nächsten Element.
- Das ist hier das ganz große freie Element links, von dem wieder etwas am Ende weggenommen wurde.



- Wenn ein Element freigegeben wird, dann kann davor und danach jeweils ein freies Element vorliegen, mit dem das neue Element zusammengelegt werden kann.
- In diesem Beispiel fand sich danach ein freies Element.
- Prinzipiell können bei einer Freigabe bis zu drei Elemente zusammengelegt werden.

Globale Datenstruktur der Speicherverwaltung

alloc.c

```
memnode dynmem[MEM_SIZE] = {
   /* bleibt immer im Ring der freien Speicherflaechen */
   {0, &dynmem[1]},
   /* enthaelt zu Beginn den gesamten freien Speicher */
   {sizeof dynmem - 2*sizeof(memnode), dynmem}
};
memnode* node = dynmem;
memnode* root = dynmem;
```

- In dem einfachen Beispiel wird nur Speicher aus dem großen Array *dynmem* vergeben.
- In diesem liegt gleich zu Beginn das Spezialelement, gefolgt von dem großen Element, dem der restliche freie Speicher gehört.
- *root* zeigt immer auf das Spezialelement.
- *node* ist der im Ring herumwandernde Zeiger, der immer auf ein freies Element im Ring verweist.
- Bei einer ernsthaften Implementierung (siehe Übungen und Wettbewerb!) sind dann sukzessive Kacheln vom Betriebssystem zu holen und zu verwalten.

alloc.c

```
memnode* successor(memnode* p) {
   return (memnode*) ((char*)(p+1) + p->size);
}
```

- Das jeweils im Speicher nachfolgende Element zu finden, ist einfach mit Hilfe der Adressarithmetik.
- Zu beachten ist, dass zuerst die Zeigerarithmetik auf Basis des Zeigertyps *memnode*∗ erfolgt mit $p+1$ und dann, um die Größe in Bytes zu addieren, dieser zwischendurch in einen **char**-Zeiger konvertiert werden muss.
- Bei belegten Elementen ist das vorangehende Element immer über den *next*-Zeiger ermittelbar.
- Wenn wir den Ring der freien Elemente durchlaufen, behalten wir immer noch einen Zeiger auf das freie Element davor. Von dem aus können ggf. mit *successor* noch die dazwischenliegenden belegten Speicherelemente durchlaufen werden.
- All diese Tricks stellen sicher, dass die Verwaltungsobjekte nicht zuviel Speicherplatz belegen.

alloc.c

```
void* my_malloc(size_t size) {
   assert(size >= 0);
   if (size == 0) return 0;
   /* runde die gewuenschte Groesse auf
      das naechste Vielfache von ALIGN */
   if (size % ALIGN) {
      size += ALIGN - size % ALIGN;
   }
   /* Suche und Vergabe ... */
}
```

- *malloc* und analog *my_malloc* müssen darauf achten, dass der vergebene Speicher korrekt ausgerichtet ist (Alignment). Am einfachsten ist es hier, alles auf die maximale Alignment-Anforderung auszurichten, das ist hier *ALIGN*.
- Für *ALIGN* wird *alignof*(*max_align_t*) verwendet.

alloc.c

```
memnode* prev = node; memnode* ptr = prev->next;
do {
   if (ptr->size >= size) break; /* passendes Element gefunden */
   prev = ptr; ptr = ptr->next;
} while (ptr != node); /* bis der Ring durchlaufen ist */
if (ptr->size < size) return 0; /* Speicher ist ausgegangen */
```

- *node* ist der im Ring herumwandernde Zeiger auf ein freies Element.
- Wir setzen *prev* auf *node* und *node* gleich auf das nächste Element. Auf diese Weise kennen wir immer den Vorgänger.
- Danach läuft die Schleife, bis entweder ein passendes freies Element gefunden wurde oder wir den gesamten Ring durchlaufen haben.

alloc.c

```
if (ptr->size < size + 2*sizeof(memnode)) {
   node = ptr->next; /* "circular first fit" */
   /* entferne ptr aus dem Ring der freien Speicherflaechen */
   prev->next = ptr->next;
   /* suche nach der unmittelbar vorangehenden Speicherflaeche;
      zu beachten ist hier, dass zwischen prev und ptr noch
      einige belegte Speicherflaechen liegen koennen
   */
   for (memnode* p = prev; p < ptr; p = successor(p)) {
      prev = p;
   }
   ptr->next = prev;
   return (void*) (ptr+1);
}
```

- Wenn das gefundene freie Element genau passt bzw. zu klein ist, um weiter zerlegt zu werden, muss es aus der Liste der freien Element herausgenommen werden.
- Außerdem muss das korrekte Vorgängerelement gefunden werden, auf das der *next*-Zeiger zu verweisen hat.

alloc.c

```
node = ptr; /* "circular first fit" */
/* lege das neue Element an */
memnode* newnode = (memnode*)((char*)ptr + ptr->size - size);
newnode->size = size; newnode->next = ptr;
/* korrigiere den Zeiger der folgenden Speicherflaeche,
   falls sie belegt sein sollte */
memnode* next = successor(ptr);
if (next < dynmem + MEM_SIZE && next->next == ptr) {
   next->next = newnode;
}
/* reduziere die Groesse des alten Elements
   aus dem Ring der freien Speicherflaechen */
ptr->size -= size + sizeof(memnode);
return (void*) (newnode+1);
```

- Andernfalls ist das gefundene freie Element zu zerlegen in ein weiterhin freies Element am Anfang der Fläche und das neue belegte Element.
- Ferner ist darauf zu achten, dass das folgende Element, falls es belegt ist, auf das neugeschaffene Element davor verweist.

alloc.c

```
void my_free(void* ptr) {
   if (!ptr) return;
   memnode* node = (memnode*) ptr - 1;
   assert(is_allocated(node));
   memnode* prev = node->next;
   while (is_allocated(prev)) {
      prev = prev->next;
   }
   node->next = prev->next; prev->next = node;
   join_if_possible(node, node->next);
   join_if_possible(prev, node);
}
```

- Bei belegten Elementen verweist *next* auf das im Speicher zuvor liegende Element, das frei oder belegt sein kann.
- Wir gehen soweit zurück, bis wir das zuvor liegende freie Element finden, damit wir das aktuelle Element in die verkettete Liste der freien Elemente einhängen können.
- Danach wird überprüft, ob ein Zusammenlegen möglich ist.

alloc.c

```
void join_if_possible(memnode* prev, memnode* ptr) {
   memnode* p = (memnode*)((char*) (prev + 1) + prev->size);
   if (p != ptr) return;
   if (prev->size == 0) return;
   prev->next = ptr->next;
   prev->size += ptr->size + sizeof(memnode);
   memnode* next = successor(ptr);
   if (next < dynmem + MEM_SIZE && next->next == ptr) {
      next->next = prev;
   }
   if (node == ptr) node = prev;
}
```

- *prev* und *ptr* verweisen beide auf freie Elemente.
- Wenn *prev* nicht unmittelbar *ptr* vorangeht, lassen sich die beiden Elemente nicht zusammenlegen.
- Wenn zusammengelegt wird, dann ist beim *ptr* folgenden Element der *next*-Zeiger anzupassen, falls dieser zuvor auf *ptr* verwies.

Bei der Speichervergabe kommt es zwangsläufig zu Fragmentierungen, d.h. zum Verlust nutzbarer Speicherfläche. Dabei wird zwischen *interner* und *externer* Fragmentierung unterschieden:

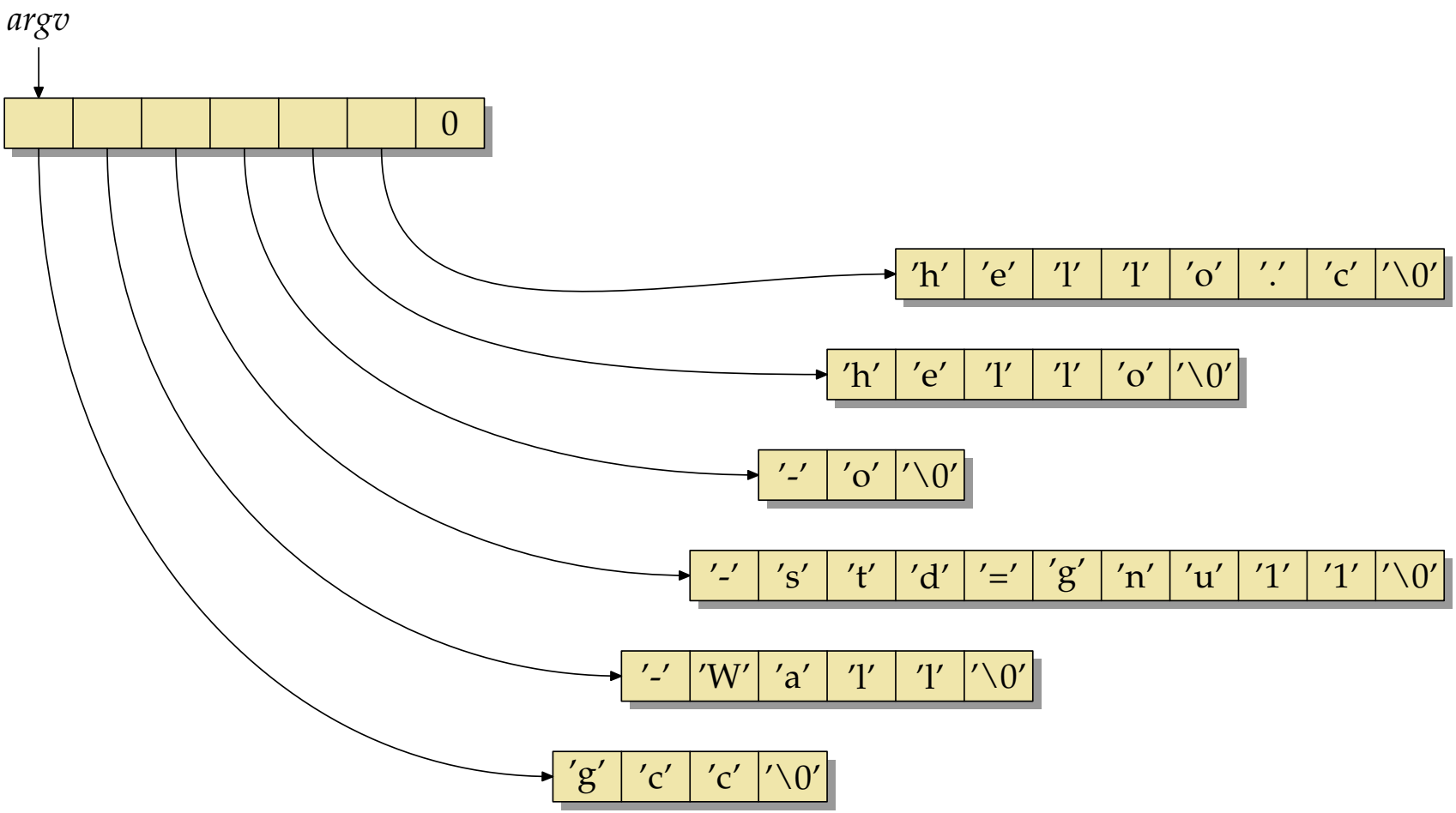
- Interne Fragmentierung entsteht bei der Anwendung des Alignments. Wenn beispielsweise 17 Bytes belegt werden sollen, werden diese bei einer Alignment-Anforderung von 16 auf 32 erhöht, so dass 15 Bytes ungenutzt bleiben. Ebenso zählt die Verwaltungsdatenstruktur (hier 16 Bytes) ebenfalls zur internen Fragmentierung.
- Externe Fragmentierung entsteht, wenn kleine freie Speicherblöcke entstehen, die nicht zu größeren freien Speicherblöcken zusammengelegt werden (können).

Der vorgestellte Algorithmus ist sehr ineffizient:

- Die Zahl der durchzulaufenden freien Elemente ist nur durch die Gesamtzahl der freien Elemente nach oben begrenzt.
- Das Verfolgen einer Zeigerliste ist prinzipiell sehr ineffizient, da die Zugriffe entlang der Zeigerkette sequentiell erfolgen müssen und die folgenden Adressen, von denen geladen wird, nicht von der Hardware „erratbar“ sind (*hardware prefetch*).

```
int main(int argc, char* argv[]) {
   /* ... */
}
```

- *main* erhält gemäß dem Standard zwei Parameter, *argc* und *argv*, die der Übermittlung der Kommandozeilenparameter dienen.
- *argc* enthält die Zahl der Parameter, wobei der Kommandoname mitgezählt wird.
- *argv* ist ein Zeiger auf ein Array von Zeigern, das auf die einzelnen Kommandozeilenparameter verweist.
- *argv*[0] zeigt auf den Namen des Kommandos, *argv*[1] bis *argv*[*argc*−1] zeigen auf die einzelnen Parameter.
- Die Zeigerliste wird durch einen Nullzeiger terminiert, d.h. *argv*[*argc*] ist 0.



- Dies ist die Repräsentierung der Kommandozeilenparameter für „gcc -Wall -std=gnu11 -o hello hello.c“. *argc* hätte hier den Wert 6.

args.c

```
#include <stdio.h>

int main(int argc, char* argv[]) {
   printf("Command name: %s\n", argv[0]);
   printf("Number of command line arguments: %d\n", argc-1);
   for (int i = 1; i < argc; i++) {
      printf("Argument %d: %s\n", i, argv[i]);
   }
}
```

- Dieses Programm gibt den Kommandonamen (in *argv*[0]) und die übrigen Kommandozeilenparameter aus.
- Der Kommandoname ist normalerweise der Name, mit dem ein Kommando aufgerufen worden ist.

args2.c

```
#include <stdio.h>

int main(int argc, char* argv[]) {
   char* cmdname = *argv++; --argc;
   printf("Command name: %s\n", cmdname);
   printf("Number of command line arguments: %d\n", argc);
   while (argc-- > 0) {
      printf("Argument: %s\n", *argv++);
   }
}
```

- Alternativ können die Kommandozeilenparameter auch sukzessive „konsumiert“ werden, indem entsprechend *argc* gesenkt und *argv* weitergesetzt wird.
- Dann sollte aber die Invariante eingehalten werden, dass *argc* die Zahl der noch unter *argv* verbleibenden Parameter angibt.

mygrep.c

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

int main(int argc, char* argv[]) {
   char line[256];

   if (argc != 2) {
      fprintf(stderr, "Usage: %s pattern\n", argv[0]);
      exit(1);
   }

   while (fgets(line, sizeof line, stdin)) {
      if (strstr(line, argv[1])) {
         fputs(line, stdout);
      }
   }
}
```

- *strstr* sucht nach dem ersten Vorkommen des zweiten Parameter in dem ersten Parameter und liefert, falls gefunden, einen Zeiger darauf zurück, ansonsten 0.

- Per Konvention beginnen Optionen in der Kommandozeile mit dem Minuszeichen „-“.
- Hinter dem Minuszeichen können dann ein oder mehrere Optionen folgen, die typischerweise mit nur einem Buchstaben benannt werden.
- Es gibt auch Kommandos, die längere Optionsnamen unterstützen. Dann können aber Optionen nicht mehr in einem Parameter integriert werden oder die Optionen müssen anders beginnen. GNU-Werkzeuge verwenden dafür gerne das doppelte Minuszeichen „--“.
- Zwei alleinstehende Minuszeichen beenden die Folge der Optionen.
- Danach folgen typischerweise Pflichtargumente (etwa der zu suchende Text) und/oder Eingabedateien.
- Diese Konventionen sind im POSIX-Standard festgehalten.

- Zusätzlich zu dem zu suchenden Text soll es möglich sein. Optionen anzugeben (beides in Nachbildung des originalen *grep*-Kommandos):
  - Die Option „-n“ (*number*) soll die jeweilige Zeilennummer mit ausgeben.
  - Die Option „-v“ (*veto*) soll dazu führen, dass nur die Zeilen ausgegeben werden, die den Suchtext *nicht* enthalten.
- Die Optionen sollen kombinierbar sein, d.h. „-n“ und „-v“ können als zwei getrennte Parameter angegeben werden oder auch kombiniert, also etwa „-nv”’ oder ”‘-vn“.
- Der Konvention folgend soll „--“ die Optionen beenden. Bei dem Kommando „mygrep1 -n -- -1“ wird „-1”’ nicht als die (nicht vorhandene) Option ”‘1“ interpretiert, sondern als der Suchtext „-1“.

mygrep1.c

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

int main(int argc, char *argv[]) {
   char *cmdname = *argv++; --argc; /* take command name */
   bool opt_v = false; /* option -v: print non-matching lines */
   bool opt_n = false; /* option -n: emit line numbers */

   /* process options ... */

   /* do the actual work */
   char line[256]; /* input line */
   int lineno = 0; /* current line number */
   while (fgets(line, sizeof line, stdin)) {
      lineno++;
      if (!strstr(line, pattern) == opt_v) {
         if (opt_n) printf("%d: ", lineno);
         fputs(line, stdout);
      }
   }
}
```

Suchkommando mit Optionen

mygrep1.c

```
/* process options */
for(; argc > 0 && **argv == '-'; argc--, argv++) {
   /* per convention we interpret "--" as end of options */
   if (argv[0][1] == '-' && argv[0][2] == 0) {
      argc--; argv++; break;
   }
   if (argv[0][1] == 0) {
      /* got just a "-" without anything following */
      fprintf(stderr, "%s: empty option\n", cmdname);
      return 1;
   }
   /* process individual options within an argument */
   for (char* s = *argv + 1; *s; s++) {
      switch (*s) {
         case 'v' : opt_v = true; break;
         case 'n' : opt_n = true; break;
         default:
            fprintf(stderr, "%s: illegal option '%c'\n", cmdname, *s);
            return 1;
      }
   }
}

/* just one remaining argument with the pattern is expected */
if (argc != 1) {
   fprintf(stderr, "Usage: %s [-nv] pattern\n", cmdname);
   return 1;
}
char* pattern = *argv++; --argc;
```

Unterstützung beliebig langer Zeilen

mygrep2.c

```
char* readline(FILE* fp) {
   int len = 32;
   char* cp = malloc(len);
   if (!cp) return 0;
   int i = 0;
   int ch;
   while ((ch = getc(fp)) != EOF && ch != '\n') {
      cp[i++] = ch;
      if (i == len) {
         /* double the allocated space */
         len *= 2;
         char* newcp = realloc(cp, len);
         if (!newcp) {
            free(cp);
            return 0;
         }
         cp = newcp;
      }
   }
   if (i == 0 && ch == EOF) {
      free(cp);
      return 0;
   }
   cp[i++] = 0;
   return realloc(cp, i); /* free unused space */
}
```

Optionen mit Werten

- Bislang waren die Optionen nur **bool**-wertig.
- Gelegentlich haben diese aber einen größeren Wertebereich, z.B. eine ganze Zahl oder ein Dateiname.
- *grep* kennt z.B. eine Option, die den jeweils anzugebenden Kontext spezifiziert (Zahl der zu zeigenden Zeilen davor und danach).
- Mit „-c 3“ sind beispielsweise drei Zeilen Kontext darzustellen.
- Konventionellerweise ist es aber auch zulässig, „-c3“ anzugeben oder in Kombination: „-nc3“.

Optionen mit Werten

mygrep3.c

```
for (char* s = *argv + 1; *s; s++) {
   switch (*s) {
      case 'c' :
         if (s[1]) {
            ++s;
         } else {
            ++argv; --argc;
            if (!argc) {
               fprintf(stderr,
                  "%s: argument for option 'c' missing\n", cmdname);
            }
            s = *argv;
         }
         /* pick argument from the rest of s[] */
         for (; *s; ++s) {
            if (!isdigit(*s)) {
               fprintf(stderr,
                  "%s: digits expected for option 'c'\n", cmdname);
               return 1;
            }
            context = context * 10 + *s - '0';
         }
         --s; /* break from outer for loop */
         break;
      case 'v' : opt_v = true; break;
      case 'n' : opt_n = true; break;
      default:
         fprintf(stderr, "%s: illegal option '%c'\n", cmdname, *s);
         return 1;
   }
}
```

Verarbeitung der Optionen mit *getopt*

- Es ist unerfreulich und fehleranfällig, die Kommandozeilenbearbeitung von Optionen „per Hand“ vorzunehmen.
- Es gibt daher im Rahmen des POSIX-Standards die Funktion *getopt*, die die Konventionen unterstützt.
- *getopt* erhält *argc* und *argv* (einschließlich dem Kommandonamen) und eine Optionsspezifikation, bestehend aus den Buchstaben der Optionennamen. Steht ein „:“ hinter einem Optionsbuchstaben, so erwartet die entsprechende Option einen Wert.
- Unsere *grep*-Nachimplementierung bräuchte also die Spezifikation „c:nv“.

Verarbeitung der Optionen mit *getopt*

mygrep4.c

```
char usage[] = "Usage: %s [-c context] [-nv] pattern\n";
/* external variables set by getopt() */
extern char* optarg;
extern int optind;
/* process options */
int option;
while ((option = getopt(argc, argv, "c:nv")) != -1) {
   switch (option) {
      case 'c':
         context = atoi(optarg); break;
      case 'n':
         opt_n = true; break;
      case 'v':
         opt_v = true; break;
      default:
         fprintf(stderr, usage, cmdname); return 1;
   }
}
argc -= optind; argv += optind; /* skip options processed by getopt() */
/* just one remaining argument with the pattern is expected */
if (argc != 1) {
   fprintf(stderr, usage, cmdname); return 1;
}
char* pattern = *argv++; --argc;
```

mygrep5.c

```
/* compile pattern */
regex_t regex; /* compiled regular expression */
unsigned int regex_flags = REG_NOSUB;
if (opt_i) {
   regex_flags |= REG_ICASE; /* ignore case */
}
if (opt_e) {
   regex_flags |= REG_EXTENDED; /* supported egrep syntax */
}
unsigned int regex_error = regcomp(&regex, pattern, regex_flags);
if (regex_error) {
   char errbuf[128];
   regerror(regex_error, &regex, errbuf, sizeof errbuf);
   fprintf(stderr, "%s: invalid regular expression: %s\n",
      cmdname, errbuf);
   return 1;
}
```

- Die POSIX-Bibliothek bietet auch eine Bibliothek für reguläre Ausdrücke an. Hier muss zunächst der reguläre Ausdruck in eine interne Datenstruktur übersetzt werden.

mygrep5.c

```
if ((regexec(&regex, line, 0, 0, 0) != 0) == opt_v) {
```

- Statt *strstr* wird dann *regexec* verwendet mit der zuvor einmal erstellten Datenstruktur.
- *regexec* liefert 0 zurück, wenn der reguläre Ausdruck für einen Teil der Zeichenkette zutrifft.
- *regexec* unterstützt auch das Extrahieren von Teilen einer Zeichenkette mit Hilfe von Klammern-Ausdrücken. Davon wird hier aber kein Gebrauch gemacht und deswegen sind zwei Parameter auf 0 gesetzt.

- C unterstützt (wie andere Programmiersprachen auch) sogenannte Übersetzungseinheiten, d.h. ein Programm kann in mehrere separate Teile zerlegt werden, die sich getrennt übersetzen lassen.
- C unterstützt kein gehobenes Modulkonzept, sondern relativ einfache Mechanismen, wie Variablen und Funktionen aus anderen Übersetzungseinheiten benutzt werden können.
- Dennoch ist es mit einer geeigneten Vorgehensweise möglich, Schnittstellen sauber zu definieren und bis zu einem gewissen Rahmen auch die Gewährleistung der Schnittstellensicherheit zu erreichen.
- Dies ist jedoch nicht ohne Hilfsmittel zu bewerkstelligen.

deklvsdef.c

```
int f(int a, int b); /* eine Funktions-Deklaration */

/* eine Funktions-Definition */
int max(int a, int b) {
   return a > b? a: b;
}

extern int i; /* eine Variablendeklaration */
int j = 27; /* eine Variablendefinition */
```

- Eine Deklaration teilt dem Übersetzer alle notwendigen Informationen mit, die eine anschließende Verwendung des deklarierten Namens ermöglicht.
- Eine Definition enthält alle Informationen, die zur Erzeugung des Objekts benötigt werden:
  - Bei Funktionen gehört der Programmtext mit den Anweisungen dazu.
  - Bei Variablen schließt die Definition die Initialisierung ein (falls eine gewünscht wird).

Eindeutigkeit einer Definition

- C hat einen globalen Namensraum für globale Variablen und Funktionen.
- Keine globale Variable oder Funktion darf über alle Übersetzungseinheiten und Bibliotheken hinweg mehr als einmal definiert werden.
- Es sind aber beliebig viele nicht-definierende Deklarationen einer Funktion oder Variablen zulässig.
- Die nicht-definierenden Deklarationen dienen im Kontext der Modularisierung dazu, eine Funktion oder Variable zu deklarieren, die in einer anderen Übersetzungseinheit definiert wird.

main.c

```
#include <stdio.h>

/* Deklarationen */
extern int i;
extern void f();

int main() {
   printf("Wert von i vor dem Aufruf: %d\n", i);
   f();
   printf("Wert von i nach dem Aufruf: %d\n", i);
}
```

lib.c

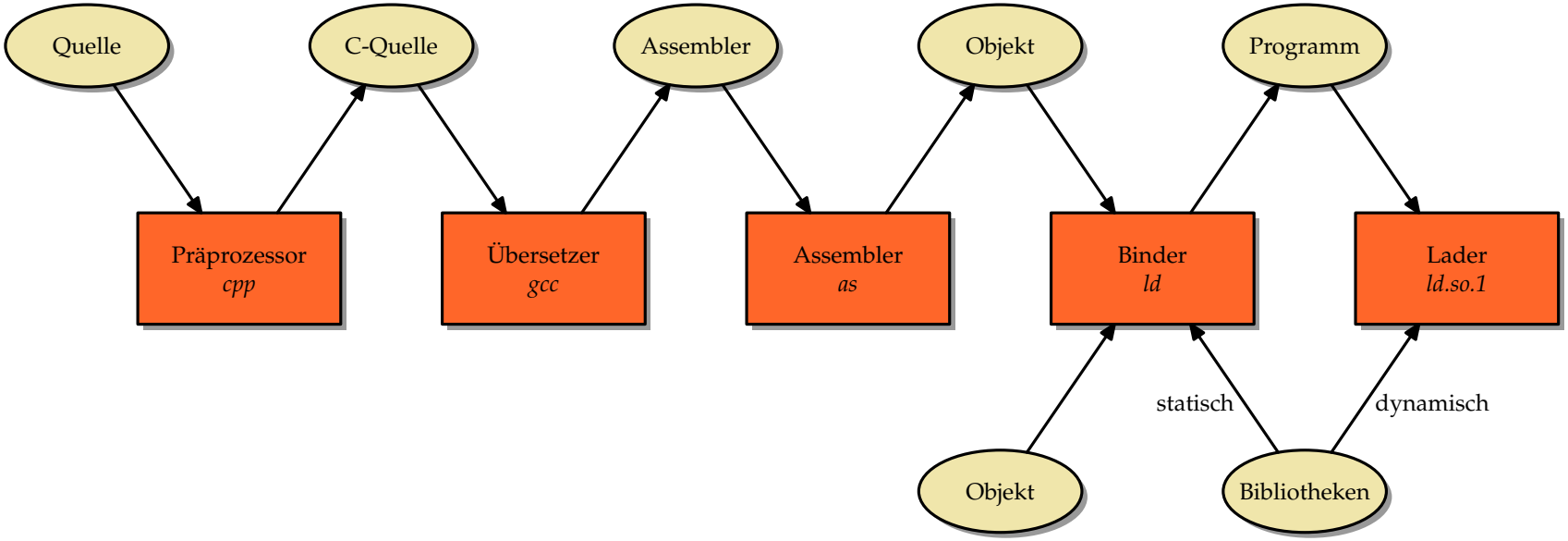
```
int i = 1; /* Definition */

void f() { /* Definition */
   ++i;
}
```

Deklarationen mit extern

main.c

```
extern int i;
extern void f();
```

- Das Schlüsselwort **extern** hat zwei Funktionen:
  - Es macht klar, dass es sich um eine Deklaration und nicht um eine Definition handelt.
  - Es stellt fest, dass es sich um einen Namen aus dem globalen Namensraum handelt.
- Wenn wir mit Variablen- oder Funktionsnamen Objekte einer anderen Übersetzungseinheit ansprechen wollen, müssen diese Namen im globalen Namensraum sein.
- Jede Übersetzungseinheit, die diese Namen benutzen möchte, benötigt eine passende Deklaration.
- Ob Deklaration und Definition aus zwei verschiedenen Übersetzungseinheiten zueinander passen, wird nicht überprüft.



- Der Übersetzer „sieht“ jeweils nur eine Übersetzungseinheit mitsamt allen per **#include** hereinkopierten Deklarationen.
- Der Binder (*ld*) und der dynamische Lader (*ld.so.1*) arbeiten nur mit Namen und der Adressen, die sie repräsentieren.
- Es gibt also keine Überprüfung, ob die Definition und die Nutzung eines Namens entsprechend den Regeln von C konform zueinander sind.
- Entsprechend besitzt C keine Schnittstellensicherheit.

ggt.h

```
int ggt(int a, int b);
```

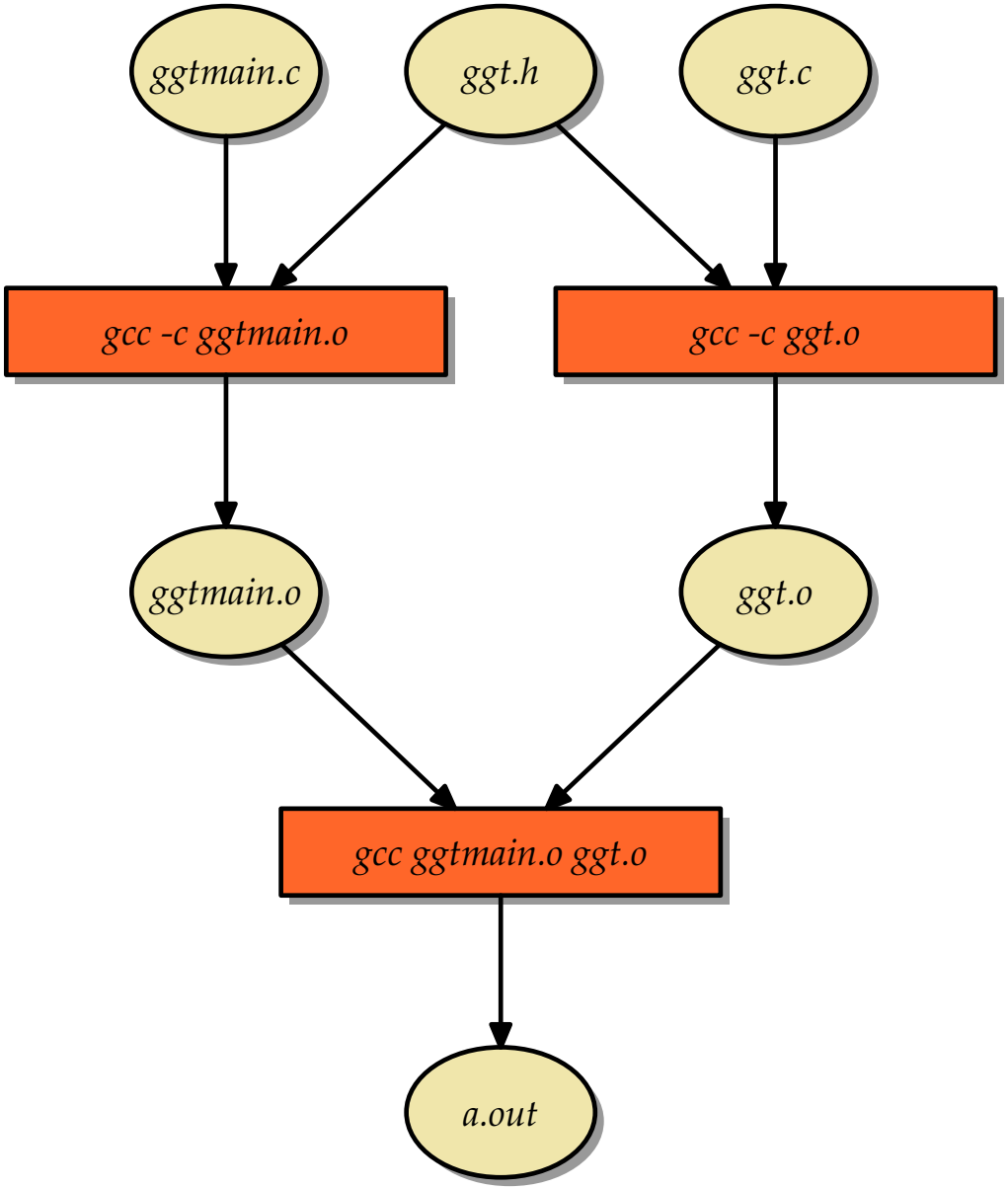
ggt.c

```
#include "ggt.h"

int ggt(int a, int b) {
   while (a != b) {
      if (a > b) {
         a -= b;
      } else {
         b -= a;
      }
   }
   return a;
}
```

- In *ggt.h* wird die Funktion *ggt* deklariert, in *ggt.c* wird sie definiert.
- Zwar könnte das **#include** wegfallen, aber dann fiele auch die Überprüfung weg, ob die Deklaration in *ggt.h* mit der Definition in *ggt.c* übereinstimmt.

ggtmain.c

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

int main(int argc, char* argv[]) {
   char* cmdname = *argv++; --argc;
   char usage[] = "Usage: %s a b\n";
   if (argc != 2) {
      fprintf(stderr, usage, cmdname);
      exit(1);
   }
   int a = atoi(argv[0]);
   int b = atoi(argv[1]);
   if (a > 0 && b > 0) {
      printf("%d\n", ggt(a, b));
   } else {
      fprintf(stderr, usage, cmdname);
      exit(1);
   }
}
```


ggtmain.c
ggt.h
ggt.c
gcc -c ggtmain.o
gcc -c ggt.o
ggtmain.o
ggt.o
gcc ggtmain.o ggt.o
a.out

- Die Header-Datei *ggt.h* wurde sowohl bei der Übersetzung von *ggt.c* als auch der Übersetzung von *ggtmain.c* gelesen.
- Auf diese Weise lässt sich Schnittstellensicherheit erreichen, denn wenn
  - die Nutzung in *ggtmain.c* mit der Deklaration in *ggt.h* konform geht und
  - die Definition von *ggt* in *ggt.c* mit der Deklaration in *ggt.h* übereinstimmt,

  haben wir die Sicherheit, dass die Nutzung in *ggtmain.c* mit der Definition in *ggt.c* konform geht.

- Die Schnittstellensicherheit steht und fällt aber damit, dass bei einer Änderung einer Header-Datei (wie etwa *ggt.h*) die Übersetzungseinheiten neu übersetzt werden, die direkt oder indirekt *ggt.h* einbeziehen.
- Unterbleibt dies, kann es zu Inkonsistenzen kommen.
- Beispiel: *ggt.c* wird übersetzt, *ggt.h* verändert, dann *ggtmain.c* entsprechend angepasst und übersetzt. Wenn dann *ggt.o* und *ggtmain.o* zusammengebaut werden, fällt es nicht mehr auf, dass beide Teile nicht zusammenpassen.

- Stuart Feldman entwickelte 1977 in den Bell Laboratories ein Werkzeug namens *make*, um das Problem zu lösen.
- (2003 erhielt er hierfür den *Software System Award* der ACM.)
- Die prinzipielle Idee ist, dass in einem *makefile* die Abhängigkeiten und die Kommandos zur erneuten Erzeugung einer Datei zusammengestellt werden und dass dann anhand der Existenz und Zeitstempel aller beteiligten Dateien von *make* automatisiert bestimmt wird, was zu tun ist, um eine Datei unter Berücksichtigung aller Abhängigkeiten korrekt neu zu erzeugen.

makefile

```
ggt:            ggt.o ggtmain.o
                gcc -o ggt ggt.o ggtmain.o
ggt.o:          ggt.h ggt.c
                gcc -c -Wall -std=gnu11 ggt.c
ggtmain.o:      ggt.h ggtmain.c
                gcc -c -Wall -std=gnu11 ggtmain.c
```

- Zeilen, die nicht mit einem Tab beginnen, nennen eine erzeugbare Datei (hier *ggt*, *ggt.o* und *ggtmain.o*).
- Dahinter folgt jeweils ein Doppelpunkt, Leerzeichen und noch auf der gleichen Zeile die Abhängigkeiten. *ggtmain.o* hängt hier beispielsweise von *ggtmain.c* und *ggt.h* ab.
- Die darauffolgenden Zeilen beginnen jeweils mit mindestens einem Tab und nennen dann die Kommandos, um die Datei neu zu erzeugen.

1. Sei Z das Ziel. Wenn das Ziel im *makefile* nicht explizit genannt ist, jedoch als Datei existiert, dann ist nichts weiter zu tun. (Falls das Ziel weder als Datei noch als Regel existiert, dann gibt es eine Fehlermeldung.)

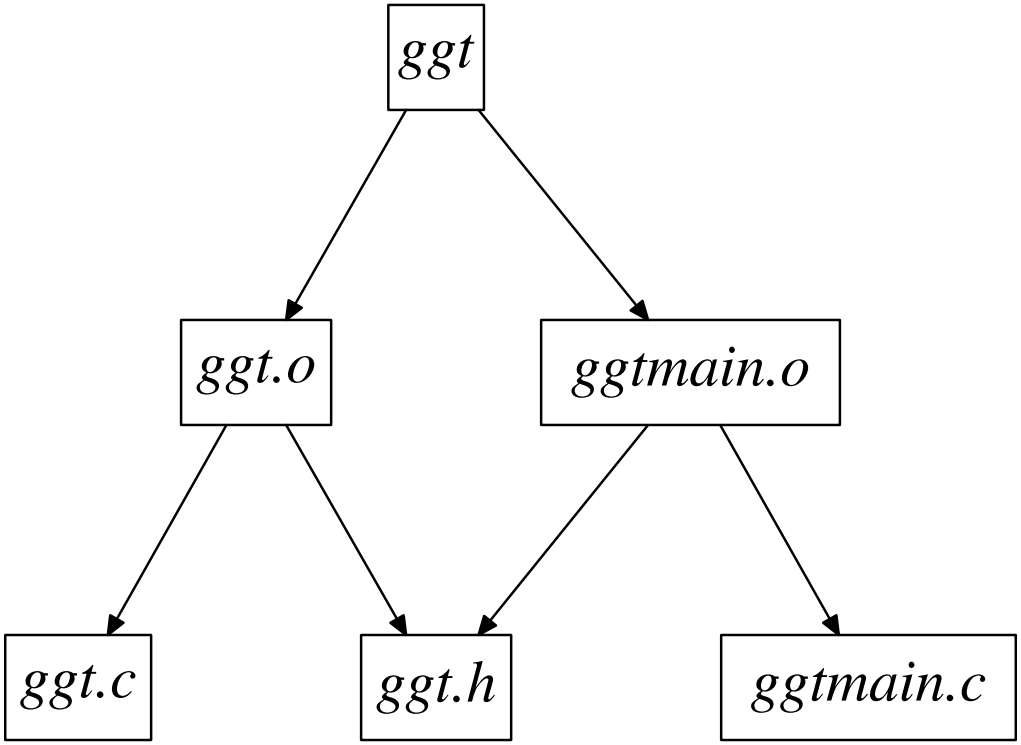
2. Andernfalls ist innerhalb des *makefile* eine Abhängigkeit gegeben in der Form

   $$Z: \; A_1 \ldots A_n,$$

   wobei die Folge $\{A_i\}_1^n$ leer sein kann ($n = 0$). Hier ist der Algorithmus (beginnend mit Schritt 1) rekursiv aufzurufen für jede der Dateien $A_1 \ldots A_n$.

3. Sobald alle Dateien $A_1 \ldots A_n$ in aktueller Form vorliegen, wird überprüft, ob der Zeitstempel von Z (letztes Schreibdatum) jünger ist als jeder der Zeitstempel der Dateien $A_1 \ldots A_n$.

4. Falls es ein A_i gibt, das neueren Datums ist als Z, dann werden die zu Z gehörenden Kommandos ausgeführt, um Z neu zu erzeugen.



- Die Abhängigkeiten lassen sich als gerichteter, antizyklischer Graph (DAG) darstellen.
- Die Quelldateien werden dann durch Blattknoten dargestellt (d.h. Knoten, von denen keine Kanten ausgehen).

- Wenn wir mit *make* oder einem vergleichbaren Werkzeug arbeiten, sind wir einen wesentlichen Schritt weiter. Aber dennoch haben wir nur dann Schnittstellensicherheit, wenn die Abhängigkeiten vollständig im *makefile* erfasst sind.
- Bei größeren Projekten lässt sich das nicht mehr „per Hand“ aktuell und korrekt halten.
- Hier helfen Werkzeuge, die die Abhängigkeiten automatisiert aus den C-Quellen extrahieren und das *makefile* entsprechend aktualisieren.
- Bekannt ist hier das Werkzeug *makedepend*, das sich aber nicht immer konform zum *gcc* verhält.
- Die Abhängigkeiten lassen sich auch mit Hilfe der Option „-M“ durch den *gcc* ermitteln.
- Unser *gcc-makedepend* bietet die Funktionalität von *makedepend* auf Basis von *gcc -M* an.

Privater Namensraum

- Die auf obersten Ebene (d.h. nicht verschachtelt) deklarierten Variablen und Funktionen sind normalerweise global im Namensraum sichtbar, d.h. auch von anderen Übersetzungseinheiten aus nutzbar.
- Wenn das vermieden werden soll, können Variablen und Funktionen auch **static** deklariert werden.
- Wenn innerhalb eines Blocks eine Variable **static** deklariert wird, lebt sie wie eine globale Variable, bleibt aber nur lokal innerhalb des umgebenden Blocks sichtbar.

Generelle Vorgehensweise

Im Normalfall bietet sich folgende Vorgehensweise an:

- Das gesamte Programm ist in geeignete Teile mit jeweils durchdachten Schnittstellen zu zerlegen.
- Abgesehen von dem Hauptprogramm sollte jede Übersetzungseinheit als Paar aufgesetzt werden mit einer Header-Datei mit denen nach außen hin sichtbaren Deklarationen und der zugehörigen Implementierung.
- Spätestens wenn Header-Dateien Typdeklarationen enthalten, sind sie mit **#ifndef** etc. gegen eine Mehrfachinklusion zu schützen.
- Jede Header-Datei sollte selbst per **#include** alle notwendigen Typdeklarationen aus anderen Header-Dateien hereinkopieren.
- Jede Implementierung sollte mindestens die zugehörige Header-Datei mit **#include** hereinkopieren. Alles, was nicht in der Header-Datei genannt wird, sollte mit **static** garantiert privat gehalten werden.
- Deklarationen fremder Übersetzungseinheiten sollten immer konsequent mit **#include** aus der entsprechenden Header-Datei übernommen werden.

function.h

```
#ifndef FUNCTION_H
#define FUNCTION_H

/* remains opaque */
typedef struct function Function;

Function* create_function(const char* name, double (*f)(double x));
const char* get_name(Function* f);
double execute(Function* f, double x);

#endif
```

- Die dynamische Verknüpfung von Objekten mit Methoden kann in C über Funktionszeiger realisiert werden.
- In diesem einfachen Beispiel haben wir einen abstrakten Funktionstyp in einer Veränderlichen, der mit einem Funktionszeiger verknüpft ist.
- **double** $(*f)$(**double** x) ist ein Funktionszeiger, der einen Parameter des Typs **double** erwartet und **double** zurückliefert.

sinus.h

```
#ifndef SINUS_H
#define SINUS_H

#include "function.h"

Function* get_sinus();

#endif
```

- In diesem Fall benötigen wir pro Funktion nur ein Objekt. In Java oder C++ würde man von einer *singleton class* sprechen.

sinus.c

```
#include <math.h>
#include "sinus.h"

Function* get_sinus() {
   static Function* sinus = 0;
   if (!sinus) {
      sinus = create_function("sin", sin);
   }
   return sinus;
}
```

function_reg.h

```
#ifndef FUNCTION_REGISTRY_H
#define FUNCTION_REGISTRY_H

#include <stdbool.h>
#include "function.h"

typedef struct FunctionRegistry FunctionRegistry;

FunctionRegistry* create_registry();
bool add_function(FunctionRegistry* freg, Function* f);
Function* lookup(FunctionRegistry* freg, const char* name);
void delete_registry(FunctionRegistry* freg);

#endif
```

- Hier ist es sinnvoll, in einer Registratur die Funktionsnamen mit den Funktionsobjekten zu verknüpfen.

testit.c

```
int main() {
   FunctionRegistry* freg = create_registry();
   if (freg) {
      add_function(freg, get_sinus());
      add_function(freg, get_cosinus());

      for(;;) {
         printf(": ");
         char* line = readline(stdin);
         if (!line) break;
         int fname_start = 0; int fname_end = 0; double x = 0;
         if (sscanf(line, " %n%*s%n %lf", &fname_start, &fname_end, &x) != 1 ||
               fname_start == fname_end) {
            printf("usage: function value\n");
            free(line);
            continue;
         }
         char* fname = line + fname_start; line[fname_end] = 0;
         Function* f = lookup(freg, fname);
         if (f) {
            double result = execute(f, x);
            printf("%g\n", result);
         } else {
            printf("no such function: %s\n", fname);
         }
         free(line);
      }
      delete_registry(freg);
   }
}
```

Schnittstelle für das dynamische Nachladen


```
#include <dlfcn.h>
#include <link.h>

void* dlopen(const char* pathname, int mode);
char* dlerror(void);
```

- *dlopen* lädt ein Modul (*shared object*, typischerweise mit der Dateiendung „.so“), dessen Dateiname bei *pathname* spezifiziert wird.
- Der Parameter *mode* legt zwei Punkte unabhängig voneinander fest:
  - Wann werden die Symbole aufgelöst? Entweder sofort (*RTLD_NOW*) oder so spät wie möglich (*RTLD_LAZY*). Letzteres wird normalerweise bevorzugt.
  - Sind die geladenen globalen Symbole für später zu ladende Module sichtbar (*RTLD_GLOBAL*) oder wird ihre Sichtbarkeit lokal begrenzt (*RTLD_LOCAL*)? Hier wird zur Vermeidung von Konflikten typischerweise *RTLD_LOCAL* gewählt.
- Wenn das Laden nicht klappt, dann kann *dlerror* aufgerufen werden, um eine passende Fehlermeldung abzurufen.

```
#include <dlfcn.h>

void* dlsym(void* restrict handle, const char* restrict name);
int dlclose(void* handle);
```

- Die Funktion *dlsym* erlaubt es, Symbolnamen in Adressen zu konvertieren. Im Falle von Funktionen lässt sich auf diese Weise ein Funktionszeiger gewinnen. Zu beachten ist hier, dass nur bei C-Funktionen davon ausgegangen werden kann, dass der C-Funktionsname dem Symbolnamen entspricht. Bei C++ ist das ausgeschlossen. Als *handle* wird der **return**-Wert von *dlopen* verwendet, *name* ist der Symbolname.
- Mit *dlclose* kann ein nicht mehr benötigtes Modul wieder entfernt werden.

sinus.c

```
#include <math.h>
#include "function.h"

Function* get_function() {
   static Function* sinus = 0;
   if (!sinus) {
      sinus = create_function("sin", sin);
   }
   return sinus;
}
```

- Die dynamischen Module haben keine zugehörige Header-Datei mehr.
- Stattdessen steht eine Funktion mit einem einheitlichem Namen (hier: *get_function* zur Verfügung, um ein entsprechendes Objekt zu erzeugen.

function_reg.c

```
Function* lookup(FunctionRegistry* freg, const char* name) {
   FunctionRegistryEntry* entry = freg->entries;
   while (entry && strcmp(get_name(entry->f), name)) {
      entry = entry->next;
   }
   if (entry) return entry->f;

   // attempt to load it dynamically
   size_t sofile_len = strlen(name) + 5;
   char* sofile = malloc(sofile_len);
   if (!sofile) return 0;
   strcpy(sofile, "./"); strcat(sofile, name); strcat(sofile, ".so");
   void* handle = dlopen(sofile, RTLD_LAZY | RTLD_LOCAL);
   if (!handle) return 0;
   Function* (*gf)() = (Function* (*)()) dlsym(handle, "get_function");
   if (!gf) {
      dlclose(handle); return 0;
   }
   Function* f = gf();
   if (!add_function_with_handle(freg, f, handle)) {
      dlclose(handle); return 0;
   }
   return f;
}
```

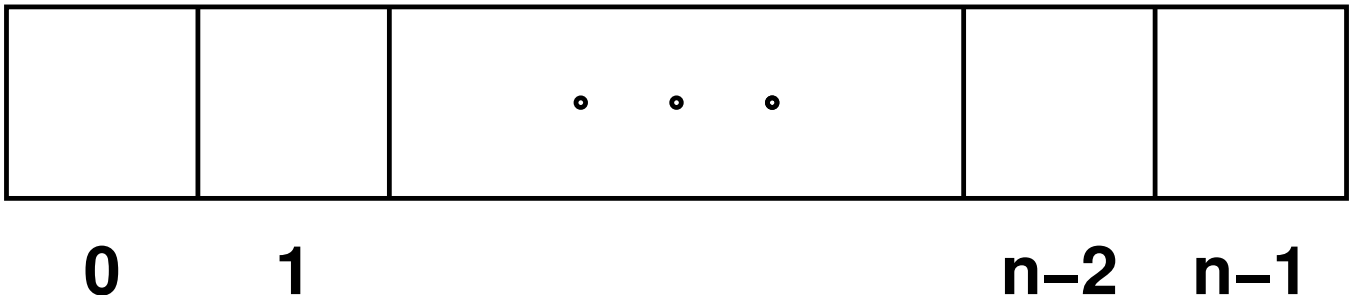
Zu den fundamentalen Abstraktionen des POSIX-Standards gehört auch die Schnittstelle für das Dateisystem. Hierzu gehört u.a.

- das Modell einer gewöhnlichen Datei (als Array von Bytes),
- spezielle Dateien (u.a. Verzeichnisse),
- die einer Datei zugeordneten Attribute (u.a. Besitzer, Gruppe und darauf Bezug nehmende Zugriffsrechte),
- Dateinamen,
- atomare Operationen auf Dateien und
- die hierarchische Organisation.

Aus IEEE Std 1003.1 (POSIX):

> *An object that can be written to, or read from, or both. A file has certain attributes, including access permissions and type. File types include regular file, character special file, block special file, FIFO special file, symbolic link, socket, and directory. Other types of files may be supported by the implementation.*

Modell einer gewöhnlichen Datei



- Eine gewöhnliche Datei entspricht einem Array aus Bytes.
- Wenn eine Datei eine Länge von n Bytes hat, sind diese über die Positionen 0 bis $n - 1$ abrufbar.
- Eine Dateiverbindung hat eine aktuelle Position p.
- Wenn ein Byte über eine Verbindung gelesen oder geschrieben wird, dann erfolgt der Zugriff auf der aktuellen Position p, die anschließend, falls die Operation erfolgreich war, um eins erhöht wird.
- Lese-Operationen bei einer Position von n sind nicht erfolgreich.

- Unix verlangt und unterstellt bei regulären Dateien *keinerlei Struktur* und unterstützt auch keine.
- Die Konzepte variabel oder konstant langer Datensätze (Records) sind im Kernel von UNIX nicht implementiert.
- Entsprechend sind gewöhnliche Dateien ganz schlichte Byte-Arrays.
- Die einzige Besonderheit ist, dass Dateien unter Unix „Löcher“ haben dürfen, d.h. einzelne Indexbereiche des Arrays können unbelegt sein. Diese werden dann als Nullbytes ausgelesen.

Verwaltungsinformationen einer Datei

Zu einer Datei gehören

- der *Inhalt* und *Aufbewahrungsort* (Menge von Blöcken auf der Platte, etc.) und
- *Verwaltungsinformationen* (Besitzer, erlaubter Zugriff, Zeitstempel, Länge, Dateityp, etc.).

Eine Datei kann beliebig viele Namen haben:

- Es ist auch zeitweise möglich, dass eine Datei überhaupt keinen Namen besitzt.
- Die Namen werden unabhängig von der Datei in Verzeichnissen verwaltet.
- Namenseinträge in einem Verzeichnis sind ähnlich wie Zeiger zu betrachten, die auf die Verwaltungsinformationen einer Datei verweisen.

- Neben den gewöhnlichen Dateien gibt es unter Unix weitere Dateiformen.
- Verzeichnisse bilden Namen auf Dateien ab. Verzeichnisse werden mit speziellen Bibliotheksfunktionen gelesen.
- Neben den Verzeichnissen gibt es insbesondere Dateivarianten, die der Interprozess-Kommunikation oder direkten Schnittstelle zu Treibern des Betriebssystems dienen.
- Diese weichen in der Semantik von dem Byte-Array ab und bieten beispielsweise uni- oder bidirektionale Kommunikationskanäle.

- Gerätedateien erlauben die direkte Kommunikation mit einem (das jeweilige Gerät repräsentierenden) Treiber.
- Sie erlauben beispielsweise den direkten Zugriff auf eine Festplatte vorbei an dem Dateisystem.
- Für Gerätedateien gibt es zwei verschiedene Schnittstellen:
  - **Zeichenweise arbeitende Geräte** (*character devices / raw devices*):
    Diese Dateien erlauben einen ungepufferten zeichenweisen Lese- und/oder Schreibzugriff.
  - **Blockweise arbeitende Geräte** (*block devices*):
    Diese Dateien erlauben Lese- und Schreibzugriffe nur für vollständige Blöcke. Diese Zugriffe laufen implizit über den Puffer-Cache von Unix.
- Neben den üblichen Lese- und Schreib-Operationen gibt es bei Geräten auch den Systemaufruf *ioctl*, der gerätespezifisch weitere Funktionalitäten zur Verfügung stellt.

Beispiel für ein *character device*


```
theon$ ls -lL /dev/random
crw-r--r--   1 root     sys      195,  0 Oct  8 17:56 /dev/random
theon$ dd if=/dev/random bs=4 count=1 2>/dev/null | od -S | sed -
n 's/.* //; 1p'
-1228706201
theon$
```

- Bei Gerätedateien wird von *ls* nicht die Größe in Bytes gegeben (wie sonst bei normalen Dateien), sondern zwei Zahlen, die *major* und *minor number*, hier konkret 195 und 0.
- Die *major number* indiziert im Kernel eine globale Tabelle aller zeichenbasierten Geräte; die *minor number* wird implizit als zusätzlicher Parameter bei den Operationen auf dem Gerät mitgegeben.
- */dev/random* gehört bislang nicht zum POSIX-Standard, wird aber von allen gängigen Unix-ähnlichen Systemen unterstützt. Lesenderweise erlaubt es das Auslesen einer Serie zufälliger Bytes, die mit Hilfe der im Kernel beobachtbaren Entropie erzeugt werden.
- Konkret werden im Beispiel mit *dd* vier Bytes ausgelesen und dann mit Hilfe von *od* als eine ganze Zahle dargestellt.

Auf eine Festplatte kann typischerweise auf drei verschiedene Weisen zugegriffen werden:

- Über ein Dateisystem.
- Über die zugehörige blockweise arbeitende Gerätedatei indirekt über den Puffer-Cache.
- Über die zugehörige zeichenweise arbeitende Gerätedatei.

Intern im Betriebsystem liegt die gleiche Schichtenstruktur der Schnittstellen vor: Zugriffe auf ein Dateisystem werden abgebildet auf Zugriffe auf einzelne Blöcke innerhalb des Puffer-Cache. Wenn der gewünschte Block zum Lesen nicht vorliegt oder ein veränderter Block im Cache zu schreiben ist, dann wird der zugehörige Treiber direkt kontaktiert.

Arten von Dateisystemen

Prinzipiell lassen sich Dateisysteme in vier Gruppen unterteilen:

- *Plattenbasierte Dateisysteme:*
  Die Daten des Dateisystems liegen auf einer lokalen Platte.
- *Netzwerk-Dateisystem:*
  Das Dateisystem wird von einem anderen Rechner über das Netzwerk angeboten. Beispiele: NFS, AFS und Samba.
- *Meta-Dateisysteme:*
  Das Dateisystem ist eine Abbildungsvorschrift eines oder mehrerer anderer Dateisysteme. Beispiele: *tfs* und *unionfs*. Ebenso können Software-RAIDs auf Basis von Meta-Dateisystemen realisiert werden.
- *Pseudo-Dateisystem:*
  Das Dateisystem ist nicht mit persistenten Daten verbunden, sondern dient als Sicht zu Datenstrukturen im Kernel. Beispiel: Das *procfs* unter */proc*, das die einzelnen aktuell laufenden Prozesse repräsentiert.

Manche moderne Dateisysteme kombinieren auch mehrere Aspekte wie beispielsweise ZFS.

- Gegeben ist die abstrakte Schnittstelle eines Arrays von Blöcken. (Dies kann eine vollständige Platte sein, eine Partition davon oder eine virtuelle Platte, wie sie etwa bei diversen RAID-Verfahren entsteht.)
- Zu den Aufgaben eines plattenbasierten Dateisystems gehört es, ein Array von Blöcken so zu verwalten, dass
  - über ein hierarchisches Namenssystem
  - Dateien (bis zu irgendeinem Maxium) frei wählbarer Länge
  - gespeichert und gelesen werden können.

Aus dem Buch *Advanced UNIX Programming* von Marc J. Rochkind, S. 29, zum Umgang mit einer Schreib-Operation:

> *I've taken note of your request, and rest assured that your file descriptor is OK,*
>
> *I've copied your data successfully, and there's enough disk space. Later, when it's convenient for me, and if I'm still alive, I'll put your data on the disk where it belongs.*
>
> *If I discover an error then I'll try to print something on the console, but I won't tell you about it (indeed, you may have terminated by then).*
>
> *If you, or any other process, tries to read this data before I've written it out, I'll give it to you from the buffer cache, so, if all goes well, you'll never be able to find out when and if I've completed your request.*
>
> *You may ask no further questions. Trust me. And thank me for the speedy reply.*

Was passiert, wenn dann mittendrin der Strom ausfällt?

- Blöcke einer Datei oder gar ein Verwaltungsblock sind nur teilweise beschrieben.
- Verwaltungsinformationen stimmen nicht mit den Dateiinhalten überein.

Im Laufe der Zeit gab es mehrere Entwicklungsstufen bei Dateisystemen in Bezug auf die Integrität:

- Im Falle eines Falles muss die Integrität mit speziellen Werkzeugen überprüft und ggf. hergestellt werden. Beispiele: Alte Unix-Dateisysteme wie UFS (alt), ext2 oder aus der Windows-Welt die Familie der FAT-Dateisysteme.
- Ein Journalling erlaubt normalerweise die Rückkehr zu einem konsistenten Zustand. Beispiele: Neuere Versionen von UFS, ext3 und reiser3.
- Das Dateisystem ist immer im konsistenten Zustand und arbeitet entsprechend mit Transaktionen analog wie Datenbanken. Hinzu kommen Überprüfungssummen und Selbstheilungsmechanismen (bei redundanten RAID-Verfahren). Beispiele: ZFS, btrfs, ext4, reiser4 und NTFS.

Organisation moderner Dateisysteme

Moderne Dateisysteme wie ZFS und btrfs werden als B-Bäume organisiert:

- B-Bäume sind sortierte und balancierte Mehrwege-Bäume, bei denen Knoten so dimensioniert werden, dass sie in einen physischen Block auf der Platte passen. (Wobei es Verfahren gibt, die mit dynamischen Blockgrößen arbeiten.)
- Entsprechend besteht jeder Block aus einer Folge aus Schlüsseln, Zeigern auf zugehörige Inhalte und Zeiger auf untergeordnete Teilbäume.
- Es werden nie in Benutzung befindliche Blöcke verändert. Stattdessen werden sie zunächst kopiert, angepasst, geschrieben und danach der neue statt dem alten Block verwendet bzw. verlinkt (*copy on write*).
- Alte Versionen können so auch bei Bedarf problemlos erhalten bleiben (*snapshots*), indem auf das Aufräumen verzichtet wird.

Hierarchie der Dateisysteme

- Jedes Dateisystem enthält eine Hierarchie der Verzeichnisse.
- Darüber hinaus gibt es auch eine Hierarchie der Dateisysteme.
- Es beginnt mit der Wurzel / und dem die Wurzel repräsentierenden Wurzel-Dateisystem. (Dies ist das erste Dateisystem, das verwendet wird und das auch das Betriebssystem oder zumindest wesentliche Teile davon enthält.)
- Weitere Dateisysteme können bei einem bereits existierenden Verzeichnis eingehängt werden.
- So entsteht eine globale Hierarchie, die sich über mehrere Dateisysteme erstreckt.

```
doolin$ cd /
doolin$ df -h .
Filesystem              size   used  avail capacity  Mounted on
/dev/dsk/c0t0d0s0       7.9G   3.5G   4.3G    46%    /
doolin$ cd var
doolin$ df -h .
Filesystem              size   used  avail capacity  Mounted on
/dev/dsk/c0t0d0s7       7.9G   1.9G   5.9G    25%    /var
doolin$ cd run
doolin$ df -h .
Filesystem              size   used  avail capacity  Mounted on
swap                    1.6G    48K   1.6G     1%    /var/run
doolin$
```

- Das Werkzeug *df* listet die Dateisysteme auf mitsamt verwendeten bzw. freien Speicherplatz und dem Verzeichnis, bei dem es montiert wurde. (Hier zu sehen bei einem alten System mit rein plattenbasierten Dateisystemen.)
- Wenn *df* einen Pfadnamen erhält, gibt es die Angaben zu dem Dateisystem aus, auf den der Pfadname verweist.

Boot–block	Super–block	Inode–Liste	Datenblöcke

- In den 70er Jahren (bis einschließlich UNIX Edition VII) hatte ein Unix-Dateisystem einen sehr einfachen Aufbau, bestehend aus
  - dem Boot-Block (reserviert für den Boot-Vorgang oder ohne Verwendung),
  - dem Super-Block (mit Verwaltungsinformationen für die gesamte Platte),
  - einem festdimensionierten Array von Inodes (Verwaltungsinformationen für einzelne Dateien) und
  - einem Array von Blöcken, die entweder für Dateiinhalte oder (im Falle sehr großer Dateien) für Verweise auf weitere Blöcke einer Datei verwendet werden.

- Das heutige UFS (*UNIX file system*) geht zurück auf das von Marshall Kirk McKusick, William N. Joy, Samuel J. Leffler und Robert S. Fabry Anfang der 80er-Jahre entwickelte Berkeley Fast File System.
- Gegenüber dem historischen Aufbau enthält es einige wesentliche Veränderungen:
  - Die Verwaltungsinformationen einer Datei und der Dateiinhalt werden so auf der Platte abgelegt, dass sie möglichst schnell hintereinander gelesen werden können.
  - Dazu wird die Platte entsprechend ihrer Geometrie in Zylindergruppen aufgeteilt. Zusammenhängende Inodes und Datenblöcke liegen dann möglichst in der gleichen oder der benachbarten Zylindergruppe.
  - Die Blockgröße wurde vergrößert (von 1k auf 8k) und gleichzeitig wurden für kleine Dateien fragmentierte Blöcke eingeführt.
  - Damit der Verlust des Super-Blocks nicht katastrophal ist, gibt es zahlreiche Sicherungskopien des Super-Blocks an Orten, die sich durch die Geometrie ableiten lassen.
- Das unter Linux lange Zeit populäre ext2-Dateisystem hatte UFS als Vorbild.

- Eine Inode enthält sämtliche Verwaltungsinformationen, die zu einer Datei gehören.
- Jede Inode ist (innerhalb eines Dateisystems) eindeutig über die Inode-Nummer identifizierbar.
- Die Namen einer Datei sind *nicht* Bestandteil der Inode. Stattdessen bilden Verzeichnisse Namen in Inode-Nummern ab.
- U.a. finden sich folgende Informationen in einer Inode:
  - Eigentümer und Gruppe
  - Dateityp (etwa gewöhnliche Datei oder Verzeichnis oder einer der speziellen Dateien)
  - Zeitstempel: Letzter Lesezugriff, letzter Schreibzugriff und letzte Änderung der Inode.
  - Anzahl der Verweise aus Verzeichnissen
  - Länge der Datei in Bytes (bei gewöhnlichen Dateien und Verzeichnissen)
  - Blockadressen (bei gewöhnlichen Dateien und Verzeichnissen)

Auslesen eines Verzeichnisses

- In der Unix-Welt gibt es keine standardisierten Systemaufrufe, die ein Auslesen eines Verzeichnisses ermöglichen.
- Der Standard IEEE Std 1003.1 bietet jedoch die Funktionen *opendir*, *readdir* und *closedir* als portable Schnittstelle oberhalb der (nicht portablen) Systemaufrufe an.
- Alle anderen Funktionalitäten (Auslesen des öffentlichen Teils einer Inode, Wechseln des Verzeichnisses und sonstige Zugriffe auf Dateien) sind auch auf der Ebene der Systemaufrufe standardisiert.

dir.c

```
#include <dirent.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>

int main(int argc, char* argv[]) {
   char* cmdname = *argv++; --argc;
   char usage[] = "Usage: %s [directory]\n";
   if (argc > 1) {
      fprintf(stderr, usage, cmdname);
      exit(1);
   }
   char* dirname;
   if (argc > 0) {
      dirname = *argv;
   } else {
      dirname = ".";
   }
   /**** Auslesen von dirname ****/
}
```

dir.c

```
if (chdir(dirname) < 0) {
   perror(dirname);
   exit(1);
}
DIR* dir = opendir(".");
if (!dir) {
   perror(dirname);
   exit(1);
}
```

- Mit *chdir()* ist es möglich, das aktuelle Verzeichnis zu wechseln. Dies betrifft den aufrufenden Prozess (und wird später an neu erzeugte Prozesse weiter vererbt).
- *chdir()* wird hier verwendet, um im weiteren Verlauf den Zusammenbau zusammengesetzter Pfade aus dem Verzeichnisnamen und dem darin enthaltenen Dateinamen zu vermeiden.
- Nach dem Aufruf von *chdir()* ist das gewünschte (dann aktuelle) Verzeichnis unter dem Namen . erreichbar.

Auslesen eines Verzeichnisses

dir.c

```
struct dirent* entry;
while ((entry = readdir(dir))) {
   printf("%s: ", entry->d_name);
   struct stat statbuf;
   if (lstat(entry->d_name, &statbuf) < 0) {
      perror(entry->d_name); exit(1);
   }
   if (S_ISREG(statbuf.st_mode)) {
      printf("regular file with %jd bytes\n",
         (intmax_t) statbuf.st_size);
   } else if (S_ISDIR(statbuf.st_mode)) {
      puts("directory");
   } else if (S_ISLNK(statbuf.st_mode)) {
      char buf[1024];
      ssize_t len = readlink(entry->d_name, buf, sizeof buf);
      if (len < 0) {
         perror(entry->d_name); exit(1);
      }
      printf("symbolic link pointing to %.*s\n", len, buf);
   } else {
      puts("special");
   }
}
closedir(dir);
```

dir.c

```
struct dirent* entry;
while ((entry = readdir(dir))) {
   printf("%s: ", entry->d_name);
```

- *readdir* liefert einen Zeiger auf eine (statische) Struktur mit Informationen über die nächste Datei aus dem Verzeichnis.
- Die Struktur mag mehrere systemabhängige Komponenten haben. Relevant und portabel ist jedoch nur der Dateiname in dem Feld *d_name*

dir.c

```
struct stat statbuf;
if (lstat(entry->d_name, &statbuf) < 0) {
   perror(entry->d_name); exit(1);
}
```

- Es gibt mehrere Systemaufrufe, die den öffentlichen Teil einer Inode auslesen können.
- Dazu gehört *lstat*, das einen Dateinamen erhält und dann in dem per Zeiger referenzierten Struktur die gewünschten Informationen aus der Inode ablegt.
- Im Unterschied zu *stat*, das genauso aufgerufen wird, folgt *lstat* nicht implizit symbolischen Links, so dass wir hier die Chance haben, diese als solche zu erkennen.

dir.c

```
      if (S_ISREG(statbuf.st_mode)) {
         printf("regular file with %jd bytes\n",
            (intmax_t) statbuf.st_size);
```

- Das Feld *st_mode* aus der von *lstat()* gefüllten Datenstruktur enthält in kombinierter Form mehrere Informationen über eine Datei:
  - den Dateityp,
  - die Zugriffsrechte (rwx) für Besitzer, Gruppe und den Rest der Welt und
  - eventuelle weitere besondere Attribute wie etwa das Setuid-Bit oder das Sticky-Bit.
- Damit der Zugriff weniger kompliziert ist, gibt es standardisierte Makros im Umgang mit *st_mode*. So liefert etwa *S_ISREG* den Wert **true**, falls es sich um eine gewöhnliche Datei handelt.

dir.c

```
      } else if (S_ISLNK(statbuf.st_mode)) {
         char buf[1024];
         ssize_t len = readlink(entry->d_name, buf, sizeof buf);
         if (len < 0) {
            perror(entry->d_name); exit(1);
         }
         printf("symbolic link pointing to %.*s\n", len, buf);
```

- Wäre *stat()* an Stelle von *lstat()* verwendet worden, würde dieser Fall nie erreicht werden, da normalerweise symbolischen Links implizit gefolgt wird.
- Mit *readlink()* kann der Link selbst ausgelesen werden.
- Das Ziel eines symbolischen Links muss nicht notwendigerweise existieren. Falls das Ziel nicht existiert, liefert *stat()* einen Fehler, während *lstat()* uns unabhängig von der Existenz das Ziel nennt.

- Bei Systemaufrufen sind, soweit sie von Privilegien und/oder einem Zugriffsschutz abhängig sind, folgende u.a. folgende vier Identitäten von Belang:

effektive Benutzernummer	*geteuid()*
effektive Gruppennummer	*getegid()*
reale Benutzernummer	*getuid()*
reale Gruppennummer	*getgid()*

- Normalerweise gleichen sich die effektiven und realen Nummern. Im Falle von Programmen mit dem s-bit werden die effektiven Identitätsnummern von dem Besitzer des Programmes übernommen, während die realen Nummern gleichbleiben.
- In Bezug auf Zugriffe im Dateisystem sind die effektiven Nummern von Belang.

Zugriffsschutz bei Dateien

- Zu jeder Inode gehören die elementaren Zugriffsrechte, die Lese-, Schreib- und Ausführungsrechte angeben für den Besitzer, die Gruppe und den Rest der Welt.
- Wenn die effektive Benutzernummer die 0 ist, dann ist alles erlaubt (Super-User-Privilegien).
- Falls die effektive Benutzernummer mit der der Datei übereinstimmt, dann sind die Zugriffsrechte für den Besitzer relevant.
- Falls nur die effektive Gruppennummer mit der Gruppenzugehörigkeit der Datei übereinstimmt, dann sind die Zugriffsrechte für die Gruppe relevant.
- Andernfalls gelten die Zugriffsrechte für den Rest der Welt.

Zugriffsschutz bei Verzeichnissen

- Lese-, Schreib- und Ausführungsrechte haben bei Verzeichnissen besondere Bedeutungen.
- Das Leserecht gibt die Möglichkeit, das Verzeichnis mit *opendir* und *readdir* anzusehen, aber noch *nicht* das Recht, *stat* für eine darin enthaltene Datei aufzurufen.
- Das Ausführungsrecht lässt die Verwendung des Verzeichnisses in einem Pfad zu, der an einem Systemaufruf weitergereicht wird.
- Das Schreibrecht gewährt die Möglichkeit, Dateien in dem Verzeichnis zu entfernen (*unlink*), umzutaufen (*rename*) oder neu anzulegen. Das Ausführungsrecht ist aber eine Voraussetzung dafür.

Weitere Bits in *st_mode*

- Zusätzlich gibt es noch drei weitere Bits:

  Set-UID-Bit Bei einer Ausführung wird die effektive Benutzernummer (UID) gesetzt auf die Benutzernummer des Besitzers.

  Set-GID-Bit Entsprechend wird auch die effektive Gruppennummer (GID) gesetzt. Bei Verzeichnissen bedeutet dies, dass neu angelegte Dateien die Gruppe des Verzeichnisses erben.

  Sticky-Bit Programme mit dem Sticky-Bit bleiben im Speicher. Verzeichnisse mit diesem Bit schränken die Schreibrechte für fremde Dateien ein – nützlich für gemeinsam genutzte Verzeichnisse wie etwa */tmp*.

- Systemnahe Software ist in vielen Fällen in Besitz von Privilegien und gleichzeitig im Kontakt mit potentiell gefährlichen Nutzern, denen diese Privilegien nicht zustehen.
- Daher muß bei der Entwicklung systemnaher Software nicht nur auf die korrekte Implementierung der gewünschten Funktionalitäten geachtet werden, sondern auch auf die umfassende Verhinderung nicht gewünschter Zugriffe.
- Dazu ist die Kenntnis der typischen Angriffstechniken notwendig und die konsequente Verwendung von Programmiertechniken, die diese zuverlässig abwehren.

Beispiel: Das Werkzeug *pubfile*

- Das Werkzeug *pubfile* soll dazu dienen, Dateien im Verzeichnis *pub* unterhalb meines nicht-öffentlichen Heimatkataloges zur Verfügung zu stellen.
- So könnte *pubfile* übersetzt und in */tmp* öffentlich zur Verfügung gestellt werden:

```
theon$ id
uid=120(borchert) gid=200(sai)
theon$ gcc -Wall -o pubfile pubfile.c
theon$ cp pubfile /tmp
theon$ cat ~/pub/READ_ME
This is the file named READ_ME in the directory /home/borchert/pub.
theon$ /tmp/pubfile READ_ME
This is the file named READ_ME in the directory /home/borchert/pub.
theon$
```

Beispiel: Das Werkzeug *pubfile*


```
theon$ id
uid=819(gast) gid=207(guest)
theon$ /tmp/pubfile READ_ME
/home/borchert/pub/READ_ME: Permission denied
theon$ cat ~borchert/pub/READ_ME
cat: cannot open /home/borchert/pub/READ_ME: Permission denied
theon$
```

- Im Normalfall operiert ein Programm mit den Privilegien des aufrufenden Benutzers.
- Das gilt auch dann, wenn das Programm einem anderen Benutzer gehört.
- Hier operiert */tmp/pubfile* mit den Privilegien von *gast*.

Setzen des s-Bits für ein Programm


```
theon$ ls -l /tmp/pubfile
-rwxrwxr-x   1 borchert sai         8600 Jan 16 10:52 /tmp/pubfile
theon$ chmod u+s /tmp/pubfile
theon$ ls -l /tmp/pubfile
-rwsrwxr-x   1 borchert sai         8600 Jan 16 10:52 /tmp/pubfile
theon$
```

- Das läßt sich aber ändern, wenn der Eigentümer des Programmes dem Programm das sogenannte „s-bit“ spendiert.
- Hierbei steht „s“ für **setuid**.
- Konkret bedeutet dies, dass das Programm mit den Privilegien des Programmeigentümers operiert und nicht mit denen des Aufrufers.

```
theon$ id
uid=819(gast) gid=207(guest)
theon$ /tmp/pubfile READ_ME
This is the file named READ_ME in the directory /home/borchert/pub.
theon$
```

- Nun klappt es für andere Benutzer.

- Wir haben nun den Fall, dass das Programm Privilegien besitzt, die der Aufrufer normalerweise nicht hat.
- Natürlich sollte so ein Programm nicht all seine Privilegien (im Beispiel die Rechte von *borchert*) dem Aufrufer preisgeben.
- Stattdessen hatte der Autor von *pubfile* die Absicht, dass nur die Dateien aus dem Unterverzeichnis *pub* der Öffentlichkeit zur Verfügung stehen sollen. Wenn es möglich ist, auf andere Dateien zuzugreifen oder gar beliebige Privilegien des Programmeigentümers ausnutzen zu können, dann würden Sicherheitslücken vorliegen.

Die erste Lösung für *pubfile*

pubfile.c

```
/*
 * Display files within my pub directory.
 * Usage: pubfile {file}
 * WARNING: This program has several security flaws.
 * afb 2/2003
 */

#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <unistd.h>

const int BUFFER_SIZE = 8192;
const char* pubdir = "/home/borchert/pub";

int main(int argc, char** argv) {
   *argv++; --argc; /* skip command name */
   while (argc-- > 0) {
      /* ... process *argv++ ... */
   }
}
```

pubfile.c

```
/* process *argv++ */
char pathname[BUFFER_SIZE];
char buffer[BUFFER_SIZE];
int fd;
int count;

strcpy(pathname, pubdir);
strcat(pathname, "/");
strcat(pathname, *argv++);

if ((fd = open(pathname, O_RDONLY)) < 0) {
   perror(pathname); exit(1);
}
while ((count = read(fd, buffer, sizeof buffer)) > 0) {
   if (write(1, buffer, count) != count) {
      perror("write to stdout"); exit(1);
   }
}
if (count < 0) {
   perror(pathname); exit(1);
}
close(fd);
```

```
theon$ id
uid=819(gast) gid=207(guest)
theon$ /tmp/pubfile ../.ssh/id_rsa
-----BEGIN RSA PRIVATE KEY-----
[...]
-----END RSA PRIVATE KEY-----
theon$
```

- Unter Angabe eines relativen Pfadnamens können beliebige Dateien mit den Rechten des Benutzers *borchert* betrachtet werden.
- In diesem Beispiel wird der private RSA-Schlüssel ausgelesen, mit dessen Hilfe möglicherweise ein passwortloser Zugang auf andere Systeme mit den dortigen Privilegien von *borchert* eröffnet werden kann. Gelegentlich funktioniert das sogar auf dem gleichen System. Und hierfür genügte nur ein zu weitreichender Lesezugriff!

pubfile.c

```
/* process *argv++ */
char pathname[BUFFER_SIZE];
/* ... */
strcpy(pathname, pubdir);
strcat(pathname, "/");
strcat(pathname, *argv++);
```

- Hier wird der lokale Puffer *pathname* gefüllt, ohne auf die Größe des Puffers zu achten.
- Zwar mag *BUFFER_SIZE* großzügig gewählt sein, aber ein Argument auf der Kommandozeile kann deutlich länger sein.
- Die Frage ist ganz einfach: Was kann passieren, wenn der Indexbereich verlassen wird? Die Sprachdefinition von C selbst gibt keine Antwort darauf, abgesehen davon, dass das Verhalten dann als „undefiniert“ deklariert wird. Bei den gängigen Implementierungen mit einem rückwärts wachsenden Stack besteht die Möglichkeit, die Rücksprungadresse zu modifizieren und damit statt zum Aufrufer zu einem eingeschleusten Code springen zu lassen. Typischerweise kann der Code innerhalb des überlaufenden Puffers untergebracht werden.

Kommen solche Sicherheitslücken noch vor?

- Ja. Aktuelles Beispiel: OP-TEE Trusted OS
- Zitat aus der Projektbeschreibung: „OP-TEE is a Trusted Execution Environment (TEE) designed as companion to a non-secure Linux kernel running on Arm [..]“
- Das Buffer-Overrun-Problem bestand bei dem Systemaufruf *syscall_asymm_verify* (liegt der Funktion *TEE_AsymmetricVerifyDigest* der TEE Internal Core API zugrunde).
- Die Lücke wurde als CVE-2019-1010298 erfasst und hat den CVSS-Score 10.0 – das ist das Maximum, d.h. die Verletzung der sicheren bzw. vertrauenswürdigen Laufzeitumgebung (TEE) gelingt vollumfänglich und ohne vorherige Authentifizierung.

Siehe diesen Commit:

core/tee/tee_svc_cryp.c

```
TEE_Result syscall_asymm_verify(unsigned long state,
                        const struct utee_attribute *usr_params,
                        size_t num_params,
                        const void *data, size_t data_len,
                        const void *sig, size_t sig_len)
{
        /* ... */
        params = malloc(sizeof(TEE_Attribute) * num_params);
        if (!params)
                return TEE_ERROR_OUT_OF_MEMORY;
        res = copy_in_attrs(utc, usr_params, num_params, params);
        /* ... */
}
```

- Der Aufrufer kann *num_params* und *usr_params* beliebig preparieren. Insbesondere sind auch beliebig hohe Werte bei *num_params* möglich.
- Was passiert nun, wenn *num_params* so hoch gesetzt wird, dass es bei der Multiplikation zu einem Überlauf kommt?

Weiteres Beispiel: Contiki-NG

- Contiki-NG: „The OS for Next Generation IoT Devices“.
- Zitat aus der Projektbeschreibung: „It focuses on dependable (secure and reliable) low-power communication and standard protocols, such as IPv6/6LoWPAN, 6TiSCH, RPL, and CoAP.“
- Das Buffer-Overrun-Problem bestand bei AQL-Abfragen (*Antelope Query Language*). Es genügt, wenn ein einzelnes lexikalisches Symbol (Token) mehr als 16 Zeichen umfasst.
- Die Lücke wurde als CVE-2018-1000804 erfasst und hat den CVSS-Score 10.0 – das ist das Maximum, d.h. der Einbruch gelingt vollumfänglich und dies gelingt über das Netzwerk ohne Voraussetzungen oder auch nur eine Authentifizierung. Beim *Internet of Things* (IoT) ist dies natürlich in besonderer Weise fatal, da die Software auf solchen Geräten häufig sich nur sehr schwer aktualisieren lässt – falls überhaupt.

Beschreibung übernommen von Issue #594:

db_options.h

```
#define DB_MAX_ELEMENT_SIZE 16
```

aql.h

```
typedef char value_t[DB_MAX_ELEMENT_SIZE];
```

aql-lexer.c

```
static int
next_token(lexer_t *lexer, const char *s)
{
  size_t length;

  length = strcspn(s, separators);
  /* ... */
  /* note: lexer->value is of type value_t */
  memcpy(lexer->value, s, length);
  (*lexer->value)[length] = '\0';
  return 1;
}
```

Und das ist kein uralter Code, sondern weniger als 10 Jahre alt.

- Das Heartbeat-Protokoll wurde in Ergänzung zum SSL-Protokoll definiert: RFC 6520
- Das Protokoll soll zwei Probleme lösen:
  - Eine schnellere Alternative zu *SO_KEEPALIVE*
  - Ein alternativer Ansatz zur *Path MTU Discovery*, nachdem die ursprünglich dafür gedachten ICMP-Pakete allzu häufig von Firewalls weggefiltert werden
- Im Rahmen des Protokolls können Pings geschickt werden mit Daten (Payload) und einer zufällig gewählten Ergänzung. Solche Pings werden dann beantwortet, wobei der Payload zusammen mit anderen zufälligen Daten zurückgeschickt wird.
- Der Payload hat eine variable Länge. Deswegen findet sich im Header eines Heartbeat-Pakets ein Feld mit zwei Bytes, das den Umfang der Payload-Daten spezifiziert.

ssl/ssl3.h

```
typedef struct ssl3_record_st
        {
/*r */  int type;               /* type of record */
/*rw*/  unsigned int length;    /* How many bytes available */
/*r */  unsigned int off;       /* read/write offset into 'buf' */
/*rw*/  unsigned char *data;    /* pointer to the record data */
/*rw*/  unsigned char *input;   /* where the decode bytes are */
/*r */  unsigned char *comp;    /* only used with decompression - malloc()ed */
/*r */  unsigned long epoch;    /* epoch number, needed by DTLS1 */
/*r */  unsigned char seq_num[8]; /* sequence number, needed by DTLS1 */
        } SSL3_RECORD;
```

- Eine typische Datenstruktur für einen Kommunikationspuffer, entnommen aus openssl-1.0.1f
- *data* zeigt auf (die bereits entschlüsselten) Daten, die wir über das Netzwerk erhalten haben.
- *length* gibt an, wieviele Bytes in *data* zum Lesen zur Verfügung stehen.

ssl/d1-both.c

```
unsigned char *p = &s->s3->rrec.data[0], *pl;
/* ... */
/* Read type and payload length first */
hbtype = *p++;
n2s(p, payload);
pl = p;
```

- *s−>s3−>rrec* ist vom Typ *SSL3_RECORD* und repräsentiert das eingelesene Datenpaket, in dem sich ein Heartbeat-Paket befindet.
- *p* zeigt auf den Anfang des Datenbereichs des eingelesenen Pakets.
- Dort ist zu Beginn der Typ des Heartbeat-Pakets (ein Byte) und der Umfang des beigefügten Payloads (zwei Bytes).
- *n2s* konvertiert zwei Bytes vom Netzwerk in *network byte order* in eine ganze Zahl (*short*).
- *payload* kann hier ein beliebiger Wert zwischen 0 und 65535 sein, der vollkommen frei von der anderen Seite gewählt werden kann.

OpenSSL: Antwortpaket

ssl/d1-both.c

```
buffer = OPENSSL_malloc(1 + 2 + payload + padding);
bp = buffer;

/* Enter response type, length and copy payload */
*bp++ = TLS1_HB_RESPONSE;
s2n(payload, bp);
memcpy(bp, pl, payload);
bp += payload;
/* Random padding */
RAND_pseudo_bytes(bp, padding);

r = dtls1_write_bytes(s, TLS1_RT_HEARTBEAT,
   buffer, 3 + payload + padding);
```

- Hier wird ein Antwort-Paket geschnürt (in Reaktion zu einem Ping), bei der die erhaltene Payload zurückzuschicken ist mitsamt einer Ergänzung aus zufälligen Daten (*padding*).
- Mit Hilfe von *memcpy* wird von *pl* (zeigt an den Anfang der erhaltenen Payload) nach *bp* kopiert.
- Kopiert werden *payload* Bytes. Es wird nirgends überprüft, ob noch *payload* Bytes hinter *pl* belegt sind...

OpenSSL: Wo ist das Problem?

Kann ein Lesen (und Weitergeben) des Speicherinhalts jenseits des Eingabe-Puffers ein Problem darstellen?

- Ja! Ziemlich anschaulich erklärt es Randall Munroe in xkcd: `http://www.xkcd.com/1354/`
- Bruce Schneier dazu:

  "Catastrophic" is the right word. On the scale of 1 to 10, this is an 11.

In der Programmiersprache C hat es bereits erfolgreiche Einbrüche aufgrund folgender Programmierfehler gegeben:

- Unzureichende Überprüfung von Argumenten beim Eröffnen von Dateien, Ausführen von Kommandos oder anderen Systemaufrufen.
- Fehlende Einhaltung der Index-Grenzen eines Arrays. Gefahr besteht hier sowohl bei Arrays auf dem Stack als auch auf dem Heap (also per *malloc()* beschafft). Gefahr droht hier auch bei beliebten Funktionen der Bibliothek wie *strcpy*, *strcat*, *sprintf* und *gets*.
- Doppelte Freigabe eines Zeigers mit *free()*.
- Benutzung eines Zeigers, nachdem er bereits freigegeben worden ist.
- Weglassen des Formats bei *printf*. Statt *printf*(*s*) sollte besser *printf*("%s", *s*) verwendet werden.

- Leider ist die Vermeidung dieser Fehler nicht einfach.
- Selbst bei sicherheitsrelevanter Software wie der ssh (*secure shell*) oder der SSL-Bibliothek (*secure socket layer*) wurden immer wieder neue Fehler bei aufwendigen Untersuchungen des Programmtexts gefunden.
- Deswegen ist es bei C sinnvoll, bei systemnaher Software auf die Standard-Bibliotheken von C teilweise zu verzichten und stattdessen auf Alternativen auszuweichen, die die Verwendung sicherer Techniken unterstützen.

- Die Unterstützung dynamischer Zeichenketten in C ist nicht sehr ausgeprägt.
- Zwar ist es leicht möglich, mit *malloc()* ein Array der gewünschten Länge zu erhalten, aber danach gibt es keine zuverlässige Längeninformation mehr.
- *strlen* ist nur sinnvoll im Falle wohldefinierter Zeichenketten, da es nach dem Nullbyte sucht.
- Entsprechend haben Standardfunktionen wie *strcpy* oder *sprintf* keine Möglichkeit zu überprüfen, ob genügend Platz für das Ergebnis vorhanden ist.
- Folglich muß die Abschätzung dem Programmierer im Vorfeld überlassen werden, die dann häufig unterlassen wird oder fehlerhaft ist.

Beispiel: Einlesen einer Zeile in eine Zeichenkette

readline.c

```
/*
 * Read a string of arbitrary length from a
 * given file pointer. LF is accepted as terminator.
 * 0 is returned in case of errors.
 * afb 3/2003
 */

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

static const int INITIAL_LEN = 8;

char* readline(FILE* fp) {
   /* ... */
}
```

- Der Umgang mit Zeichenketten ist in C sehr umständlich, wenn die benötigte Länge nicht zu Beginn bekannt ist, wie dieses Beispiel demonstriert.

Beispiel: Einlesen einer Zeile in eine Zeichenkette

readline.c

```
size_t len = 0; /* current length of string */
size_t alloc_len = INITIAL_LEN; /* allocated length */
char* buf = malloc(alloc_len);
int ch;

if (buf == 0) return 0;
while ((ch = getc(fp)) != EOF && ch != '\n') {
   if (len + 1 >= alloc_len) {
      alloc_len *= 2;
      char* newbuf = realloc(buf, alloc_len);
      if (newbuf == 0) {
         free(buf);
         return 0;
      }
      buf = newbuf;
   }
   buf[len++] = ch;
}
buf[len++] = '\0';
return realloc(buf, len);
```

Anforderungen an eine Alternative

Ein Ausweg besteht in der Schaffung einer alternativen Bibliothek für dynamische Zeichenketten in C, die folgende Anforderungen erfüllen sollte:

- Neben der eigentlichen Zeichenkette muß auch eine Längenangabe vorliegen.
- Bibliotheksfunktionen analog zu *strcpy()* und *strcat()* müssen unterstützt werden. Diese Funktionen müssen entweder die Längenangabe einhalten oder automatisch die Zeichenketten in ihrer Größe anpassen.
- Hinzu kommen Funktionen für die Initialisierung und die Freigabe von Zeichenketten.

Bei der Semantik gibt es zwei grundsätzliche Ansätze:

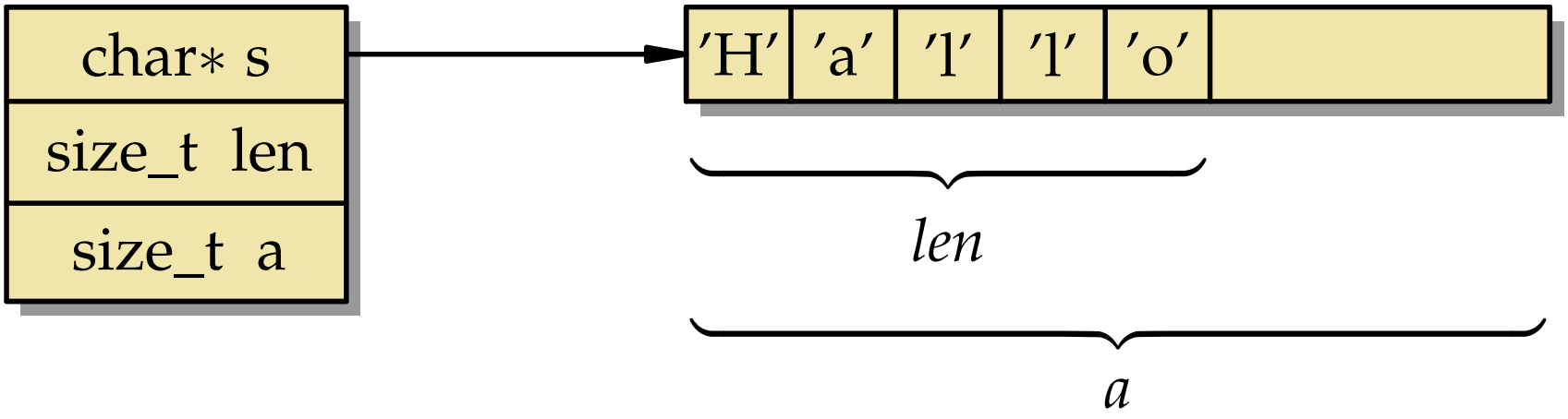
- Jede Zeichenkette ist in ihrer Repräsentierung unabhängig von allen anderen Zeichenketten und kann daher auch jederzeit frei verändert werden. Dies entspricht der traditionellen Vorgehensweise in C und der *string*-Template-Klasse in C++.
- Jede Zeichenkette ist konstant. Daher kann bei einer Operation analog zu *strcpy()* auf das Kopieren verzichtet werden. Änderungen erfordern hingegen das vorherige Anfertigen von Kopien. Dies entspricht der Vorgehensweise von Java. Hierfür wird entweder eine *garbage collection* oder der Einsatz von Referenzzählern benötigt.

- Eine C-Bibliothek, die dem ersten Ansatz folgt, wurde von Dan J. Bernstein entwickelt (u.a. für das Qmail-Paket).
- Später wurde sie von Felix von Leitner nachprogrammiert, um die Bibliothek unter der GPL (GNU General Public License) zur Verfügung stellen zu können.
- Zu finden ist sie unter `http://www.fefe.de/libowfat/`.

stralloc.h

```
typedef struct stralloc {
  char* s;
  size_t len;
  size_t a;
} stralloc;
```

- Diese öffentlich einsehbare Datenstruktur wird von Bernsteins Bibliothek verwendet.
- Der Zeiger *s* darf gleich 0 sein, um eine leere Zeichenkette zu repräsentieren. Dann müssen *len* und *a* ebenfalls 0 sein.



- *s* verweist auf einen Puffer der Länge *a*, in dem eine Zeichenkette der Länge *len* untergebracht ist. Es gilt: $len \leq a$.
- Im Gegensatz zu den normalen Zeichenketten unter C dürfen diese auch Nullbytes enthalten. Entsprechend gibt es keine Nullbyte-Terminierung.
- Bei Bedarf versuchen die *stralloc*-Funktionen automatisch, den Puffer (mit Hilfe von *realloc*) zu vergrößern (und geben **true** zurück, falls erfolgreich).

```
stralloc sa = {0};
```

- Wichtig ist die korrekte Initialisierung einer Variablen vom Typ *stralloc*. C sieht bei lokalen Variablen keine automatische Initialisierung vor, so dass hier die Initialisierung nicht vergessen werden darf.
- Damit wird übrigens nicht nur *sa.s* auf 0 initialisiert, sondern auch gleichzeitig *sa.len* und *sa.a* auf 0 gesetzt.

sareadline.c

```
/*
 * Read a string of arbitrary length from a
 * given file pointer. LF is accepted as terminator.
 * 1 is returned in case of success, 0 in case of errors.
 * afb 4/2003
 */

#include <stralloc.h>
#include <stdio.h>

int readline(FILE* fp, stralloc* sa) {
   sa->len = 0;
   for(;;) {
      if (!stralloc_readyplus(sa, 1)) return 0;
      if (fread(sa->s + sa->len, sizeof(char), 1, fp) <= 0) return 0;
      if (sa->s[sa->len] == '\n') break;
      ++sa->len;
   }
   return 1;
}
```

sareadline.c

```
int readline(FILE* fp, stralloc* sa) {
   sa->len = 0;
   /* ... */
}
```

- Hier wird zunächst die Länge auf 0 gesetzt.
- Es ist nicht unüblich, in einfachen Fällen direkt auf die *stralloc*-Datenstruktur zuzugreifen.
- Alternativ wäre es auch möglich gewesen mit *stralloc_copys*(*sa*, "") eine leere Zeichenkette zu kopieren.
- Generell dient *stralloc_copys* dazu, traditionelle nullbyte-terminierte Zeichenketten in C zu einem *stralloc*-Objekt zu kopieren.

sareadline.c

```
for(;;) {
   if (!stralloc_readyplus(sa, 1)) return 0;
   if (fread(sa->s + sa->len, sizeof(char), 1, fp) <= 0) return 0;
   if (sa->s[sa->len] == '\n') break;
   ++sa->len;
}
```

- Die **for**-Schleife behandelt das zeichenweise Einlesen, bis entweder das Zeilenende erkannt wird oder ein Fehler auftritt.
- Die Funktion *stralloc_readyplus* sorgt dafür, dass in *sa->s* mindestens ein Byte mehr Platz vorhanden ist, als die augenblickliche Länge *sa->len* beträgt.
- Nicht vergessen werden sollte die Überprüfung des Rückgabewerts. Bei 1 war die Operation erfolgreich, bei 0 konnte nicht genügend Speicher belegt werden.
- Wenn dies sichergestellt ist, kann mit *fread* das nächste Zeichen an der Position *sa->len* abgelegt werden.
- Wenn dies ein Zeilentrenner war, wird die **for**-Schleife beendet. Ansonsten wird das Zeichen akzeptiert, indem die Länge der Zeichenkette um 1 erhöht wird.

spubfile.c

```
while (argc-- > 0) {
   stralloc pathname = {0};
   char buffer[BUFFER_SIZE];
   int fd;
   int count;

   if (**argv == '.' || strchr(*argv, '/')) {
      fprintf(stderr, "invalid filename: %s\n", *argv);
      exit(1);
   }

   stralloc_copys(&pathname, pubdir);
   stralloc_cats(&pathname, "/");
   stralloc_cats(&pathname, *argv++);
   stralloc_0(&pathname);

   if ((fd = open(pathname.s, O_RDONLY)) < 0) {
      perror(pathname.s); exit(1);
   }
   /* ... copy contents of fd to stdout ... */
   close(fd);
}
```

spubfile.c

```
    stralloc_copys(&pathname, pubdir);
    stralloc_cats(&pathname, "/");
    stralloc_cats(&pathname, *argv++);
    stralloc_0(&pathname);
```

- Hinzugekommen ist hier die Funktion *stralloc_cats*, die eine traditionelle Zeichenkette an ein *stralloc*-Objekt anhängt.
- Die Funktion *stralloc_0* hängt genau ein Nullbyte an das *stralloc*-Objekt. Dies erlaubt es, *pathname.s* als traditionelle Zeichenkette in C zu verwenden — beispielsweise bei der Übergabe an die Funktion *open()*.
- Darüber hinaus wird in der korrigierten Version jeder Dateiname dahingehend überprüft, ob er mit einem Punkt beginnt, um sich insbesondere gegen die Verwendung von "." und ".." zu schützen, und ob er einen Schrägstrich enthält, um sich gegen die Angabe relativer Pfadnamen zu schützen.

stralloc sa = {0};	Initialisierung einer Zeichenkette.
stralloc_ready(*sa*, *len*)	Bereitstellung von *len* Bytes.
stralloc_readyplus(*sa*, *len*)	Bereitstellung von *len* weiteren Bytes.
stralloc_free(*sa*)	Freigabe von *sa*.
sa.s	Direkter Zugriff auf den Zeiger.
sa.len	Länge der Zeichenkette.
stralloc_copys(*sa*, *s*)	Kopieren von *s* nach *sa*.
stralloc_copy(*sa1*, *sa2*)	Kopieren von *sa2* nach *sa1*.
stralloc_cats(*sa*, *s*)	Anhängen von *s* an *sa*.
stralloc_cat(*sa1*, *sa2*)	Anhängen von *sa2* an *sa1*.
stralloc_0(*sa*)	Anhängen eines Nullbytes an *sa*.
stralloc_starts(*sa*, *s*)	Findet sich *s* zu Beginn von *sa*?
stralloc_diff(*sa1*, *sa2*)	Vergleich analog zu *strcmp*.
stralloc_diffs(*sa*, *s*)	Vergleich analog zu *strcmp*.

- *s* repräsentiert hier **char**∗, *sa* hingegen *stralloc*∗.

- Sicherheit sollte von Anfang an ein Kriterium sein. Es ist meistens ein hoffnungsloses Unterfangen, erst später Sicherheitsüberprüfungen einbauen zu wollen.
- Sicherheit sollte bei jedem Programm relevant sein, da sich sonst die Verwendung in einem sicherheitskritischen Kontext ausschließt. Nur bei temporären Wegwerf-Programmen können Sicherheitsbedenken wegfallen.
- Programme sollten nur ein Minimum an Privilegien erhalten. Häufig ist es ratsam, nicht nur auf `root`-Privilegien zu verzichten, sondern auch noch zusätzliche Restriktionen aufzunehmen wie die Limitierung des Ressourcen-Verbrauches und die Verwendung von `chroot`-Gefängnissen.
- Falls das Arbeiten mit Privilegien unverzichtbar ist, sollte das Aufteilen in mehrere Programme mit unterschiedlichen Privilegien in Betracht gezogen werden (*privilege separation*).

- Grundsätzlich sollte nichts und niemanden getraut werden, was von außen kommt.
- Bei der Überprüfung von Benutzereingaben sind Positivlisten (was ist erlaubt?) besser als Negativlisten (was ist gefährlich?).
- Sicherheit beruht auf Verantwortlichkeiten. Damit klar ist, welcher Programmteil für welche Überprüfungen verantwortlich ist, sollten entsprechende Vorgaben und Annahmen klar dokumentiert sein. So sollte beispielsweise innerhalb eines Programmes immer klar hervorgehen, wo mit ungeprüften Eingaben zu rechnen ist.
- Der wohldefinierte Bereich einer Programmiersprache sollte auf keinen Fall verlassen werden, unabhängig davon wie schwierig es sein mag, für Verletzungen passende Einbruchstechniken zu finden.

- Alle angebotenen automatischen Überprüfungen zur Übersetz- und Laufzeit sind zu verwenden.
- Wenn die Programmiersprache oder die Bibliothek nicht genügend automatische Überprüfungen mit sich bringen, ist es ratsam, Bibliotheken zu verwenden, die die Überprüfungen entweder durchführen oder überflüssig machen (Beispiel: **stralloc**-Bibliothek).
- Besser als das stille Abschneiden (Beispiel: *snprintf*()) ist die prinzipielle Unterstützung beliebig langer Eingaben. Der Speicherbedarf wird besser zentral limitiert als bei jeder einzelnen Eingabe.

- Die Grenzen aller Sicherheitsbemühungen sollten nicht vergessen werden.
- Das sicherste Programm nützt nichts, wenn die Bibliothek, der Compiler, das Betriebssystem oder die Hardware Sicherheitslücken aufweisen, die das Programm betreffen. Oder wenn sich über Nebeneffekte (wie etwa der Nutzung des L2-Caches) sicherheitsrelevante Inhalte erschließen lassen. Jüngste Beispiele sind Meltdown und Spectre.
- Ebenso ist der korrekte Umgang mit einer sicherheitskritischen Anwendung relevant. Das schwächste Glied in der Kette ist allzu häufig der Mensch (*social engineering*).

- Die Systemschnittstelle für Ein- und Ausgabe dient primär zwei Zielen:
  - Sie sollte möglichst gut abstrahieren und somit Anwendungen befreien von Hardware-Abhängigkeiten und bis zu einem gewissen Umfange auch von den Besonderheiten eines Dateisystems.
  - Sie sollte eine höchstmögliche Effizienz erlauben bis hin zum Verzicht auf jegliche zusätzliche Kopieraktionen zwischen dem Betriebssystem und dem Adressraum des Prozesses (*zero copy*).

- Dateideskriptoren sind ganzzahlige Werte aus dem Bereich $[0, N-1]$, wobei N typischerweise eine Zweierpotenz ist (etwa 512 oder 1024).
- Dateideskriptoren werden innerhalb des Betriebssystems als Indizes für Verwaltungstabellen verwendet.
- Dateideskriptoren referenzieren somit vom Betriebssystem verwaltete Objekte.
- Für jeden Prozess verwaltet das System eine eigene Tabelle. Entsprechend kann beispielsweise der Dateideskriptor 2 bei zwei Prozessen mit völlig unterschiedlichen Objekten verbunden sein.
- Die so referenzierten Objekte sind typischerweise Dateien, können aber auch Netzwerkverbindungen, Verbindungen zu anderen Prozessen, Geräte und Speicherbereiche sein.
- In C wird für Dateideskriptoren der Datentyp **int** verwendet.

Wieviel Dateideskriptoren gibt es?

openmax.c

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

int main() {
   long maxfds = sysconf(_SC_OPEN_MAX);
   printf("maximal number of open file descriptors: %ld\n", maxfds);
}
```

- Der Systemaufruf *sysconf* erlaubt die Abfrage zahlreicher Größen, von denen auch einige erst zur Laufzeit festliegen.
- Der Parameter *_SC_OPEN_MAX* liefert die maximale Zahl offener Dateien und die damit die Größe der systeminternen Tabelle der Objekte für diesen Prozess.

```
theon$ gcc -Wall -std=c11 -o openmax openmax.c
theon$ ./openmax
maximal number of open file descriptors: 512
theon$
```

- Wenn ein neuer Prozess erzeugt wird, dann wird die Tabelle mit den Dateideskriptoren kopiert.
- Entsprechend kann die Shell einige Dateideskriptoren für ein Programm, das sie startet, vorbereiten.
- Wenn nichts anderes spezifiziert wird, sind dies folgende Dateideskriptoren:

  0 Standard-Eingabe
  1 Standard-Ausgabe
  2 Standard-Fehlerausgabe

- Die Bourne-Shell und die von ihr abgeleiteten Shells erlauben das Öffnen und Schließen beliebiger Dateideskriptoren. Folgendes Beispiel ruft a.out auf, wobei 0 geschlossen wird, 7 zum Schreiben geöffnet wird auf die Datei *out* und 10 zum Lesen für die Datei *in* eröffnet wird:

```
a.out 0<&- 7>out 10<in
```

int *open*(**const char**∗ *path*, **int** *oflag*, . . .)

Eröffnet die Datei *path* mit den unter *oflag* angegebenen Modi. Ein weiterer Parameter ist nur bei neu anzulegenden Dateien notwendig. Der Systemaufruf liefert im Erfolgsfalle (d.h. ≥ 0) einen Dateideskriptor zurück.

Für *oflag* wird zunächst (normalerweise) eines der folgenden Flags ausgewählt:

O_RDONLY	nur zum Lesen eröffnen
O_WRONLY	nur zum Schreiben eröffnen
O_RDWR	zum Lesen und Schreiben eröffnen

int *open*(**const char**∗ *path*, **int** *oflag*, . . .)

oflag kann mit einem oder mehreren der folgenden Flags kombiniert werden:

O_APPEND	vor jeder Schreiboperation wird die Dateiposition implizit an das Ende gesetzt
O_CREAT	lege die Datei neu an, falls sie noch nicht existiert
O_EXCL	in Kombination mit *O_CREAT* kommt es zu einem Fehler, falls die Datei bereits existiert
O_TRUNC	die Datei wird, falls sie bereits existiert, auf die Länge 0 gekürzt

Optional gibt es noch einen dritten Parameter für den Fall, dass es eine Datei neu angelegt wird. Dieser gibt dann die initialen Zugriffsrechte an, z.B. 0666 für „rw-rw-rw-“.

ssize_t read(**int** fd, **void**∗ *buf*, *size_t nbytes*)

Liest bis zu *nbytes* Bytes von der Dateiverbindung *fd* und legt diese bei *buf* ab. Die Zahl der gelesenen Bytes wird zurückgeliefert (falls 0: Eingabeende, ≤ 0: Fehler).

ssize_t write(**int** fd, **const void**∗ *buf*, *size_t nbytes*)

Schreibt bis zu *nbytes* Bytes bei *buf* in die mit *fd* verbundene Datei. Die Zahl der geschriebenen Bytes wird zurückgeliefert (kann niedriger sein als *nbytes*, -1 bei Fehlern).

int *close*(**int** *fd*)

Schließt eine Dateiverbindung.

off_t lseek(**int** *filedes*, *off_t offset*, **int** *whence*)

Verändert die Dateiposition *pos* in Abhängigkeit von *whence* und ggf. von der Länge der Datei *len*:

SEEK_SET	$pos = offset$
SEEK_CUR	$pos \mathrel{+}= offset$
SEEK_END	$pos = len + offset$

Der Returnwert liefert die neue Dateiposition bzw. -1 bei einem Fehler.

int *ftruncate*(**int** *filedes*, *off_t length*)

Verändert die Länge der Datei, die sowohl verkleinert als auch vergrößert werden kann. Bei einer Vergrößerung entstehen potentiell „Löcher“, die als Nullbytes ausgelesen werden.

Die beiden ganzzahligen Datentypen *size_t* und *ssize_t* haben die gleiche Größe, die an die jeweilige Plattform angepasst ist:

size_t ist Teil des C-Standards (Rückgabetyp des **sizeof**-Operators!) und steht in Standard-Headern wie beispielsweise **#include** <stdlib.h> zur Verfügung. Der Datentyp ist ohne Vorzeichen.

ssize_t ist Teil des POSIX-Standards (**#include** <unistd.h>) als Rückgabetyp von u.a. *read* und *write*. Der Datentyp ist mit Vorzeichen und erlaubt so die Rückgabe von -1.

off_t ist ebenfalls Teil des POSIX-Standards (**#include** <unistd.h>) und ist ein vorzeichenbehafteter ganzzahliger Datentyp, der für Dateipositionen verwendet wird. (*off_t* kann auch größer sein als *ssize_t*, wenn z.B. eine 32-Bit-Anwendung auf einem 64-Bit-System zur Ausführung kommt.)

„Hello, world!“ mit *write*

syshello.c

```
#include <unistd.h>

int main() {
   const char hello[] = "Hello, world!\n";
   write(1, hello, sizeof hello - 1);
}
```

- Der Systemaufruf *write* gibt im Normalfall die angegebene Zahl von Bytes aus (möglicherweise auch weniger). Dies schließt auch die Ausgabe beliebiger binärer Inhalte einschließlich des Nullbytes ein. Bei der Ausgabe einer Zeichenkette ist daher immer die Länge ohne das Nullbyte anzugeben.
- Bei Zeichenkettenkonstanten ist implizit immer ein abschließendes Nullbyte enthalten, das in der Größenangabe von **sizeof** natürlich einberechnet wird.
- Da das Nullbyte nicht auszugeben ist, ist das von der auszugebenden Länge abzuziehen.

Kopieren mit der *stdio*

scopy.c

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

int main(int argc, char* argv[]) {
   char* cmdname = argv[0];
   if (argc != 3) {
      fprintf(stderr, "Usage: %s infile outfile\n", cmdname);
      exit(1);
   }
   char* infile = argv[1]; char* outfile = argv[2];
   FILE* in = fopen(infile, "r"); if (!in) perror(infile), exit(1);
   FILE* out = fopen(outfile, "w"); if (!out) perror(outfile), exit(1);
   int ch;
   while ((ch = getc(in)) != EOF) {
      if (putc(ch, out) == EOF) perror(outfile), exit(1);
   }
   fclose(in);
   if (fclose(out) == EOF) perror(outfile), exit(1);
}
```

bcopy.c

```
#include <errno.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stralloc.h>
#include <string.h>
#include <unistd.h>

char* cmdname;
/* ... */

int main(int argc, char* argv[]) {
   cmdname = argv[0];
   if (argc != 3) {
      stralloc usage = {0};
      if (stralloc_copys(&usage, "Usage: ") &&
            stralloc_cats(&usage, cmdname) &&
            stralloc_cats(&usage, " infile outfile\n")) {
         write(2, usage.s, usage.len);
      }
      exit(1);
   }
   /* ... */
}
```

bcopy.c

```
char* infile = argv[1]; char* outfile = argv[2];

int infd = open(infile, O_RDONLY);
if (infd < 0) die(infile);
int outfd = open(outfile, O_WRONLY|O_CREAT|O_TRUNC, 0666);
if (outfd < 0) die(outfile);

for(;;) {
   char ch;
   ssize_t res = read(infd, &ch, 1);
   if (res < 0) die(infile);
   if (res == 0) break;
   res = write(outfd, &ch, 1);
   if (res <= 0) die(outfile);
}

close(infd);
if (close(outfd) < 0) die(outfile);
```

bcopy.c

```
int infd = open(infile, O_RDONLY);
if (infd < 0) die(infile);
int outfd = open(outfile, O_WRONLY|O_CREAT|O_TRUNC, 0666);
if (outfd < 0) die(outfile);
```

- Mit dem Systemaufruf *open* kann eine Datei eröffnet werden. Im Erfolgsfalle wird ein (zuvor unbenutzter) Dateideskriptor zurückgeliefert.
- Der zweite Parameter gibt an, wie die Datei zu eröffnen ist. Hier wird die Datei nur zum Schreiben eröffnet. Wenn die Datei noch nicht existiert wird sie angelegt, andernfalls wird sie auf die Länge 0 gekürzt. Das entspricht genau der Semantik des Parameters "w" bei *fopen* oder einer normalen Ausgabe-Umlenkung in der Shell.
- Der optionale dritte Parameter wird nur hinzugefügt, falls bei dem zweiten Parameter *O_CREAT* mit angegeben wurde. Er legt die Zugriffsrechte fest. 0666 steht für rw-rw-rw.

Kopierschleife mit *read* und *write*

bcopy.c

```
for(;;) {
   char ch;
   ssize_t res = read(infd, &ch, 1);
   if (res < 0) die(infile);
   if (res == 0) break;
   res = write(outfd, &ch, 1);
   if (res <= 0) die(outfile);
}
```

- Sowohl bei *read* als auch *write* ist hier als Puffer nur eine **char**-Variable angegeben.
- Somit wird hier zeichenweise kopiert.

bcopy.c

```
void die(char* filename) {
   stralloc msg = {0};
   if (stralloc_copys(&msg, cmdname) &&
         stralloc_cats(&msg, ": ") &&
         stralloc_cats(&msg, strerror(errno)) &&
         stralloc_cats(&msg, ": ") &&
         stralloc_cats(&msg, filename) &&
         stralloc_cats(&msg, "\n")) {
      write(2, msg.s, msg.len);
   }
   exit(1);
}
```

- *strerror* aus **#include** <string.h> liefert die Fehlermeldung passend zu *errno*. Die bislang bekannte Funktion *perror* basiert auf *strerror*.

Zeitvergleich


```
theon$ mkfile 10m 10m
theon$ time bcopy 10m out && rm -f out

real 1m8.829s
user 0m2.738s
sys 1m6.105s
theon$ time scopy 10m out && rm -f out

real 0m0.137s
user 0m0.130s
sys 0m0.007s
theon$
```

- Das Kopierprogramm mit Systemaufrufen schneidet katastrophal schlecht ab im Vergleich zu der Fassung auf Basis der *stdio*. Warum?

copy.c

```
char buf[8192]; ssize_t nbytes;
while ((nbytes = read(infd, buf, sizeof buf)) > 0) {
   ssize_t count;
   for (ssize_t written = 0; written < nbytes; written += count) {
      count = write(outfd, buf + written, nbytes - written);
      if (count <= 0) die(outfile);
   }
}
if (nbytes < 0) die(infile);
```

- Durch die Pufferung reduzieren wir deutlich die Zahl der benötigten Systemaufrufe.
- Bei *write* akzeptieren wir die Möglichkeit, dass weniger Bytes geschrieben werden als verlangt. Das ist immer möglich – etwa durch eine Unterbrechung. Entsprechend ist eine Schleife notwendig, die darauf Rücksicht nimmt.

```
theon$ time scopy 10m out && rm -f out

real 0m0.137s
user 0m0.130s
sys 0m0.007s
theon$ time copy 10m out && rm -f out

real 0m0.027s
user 0m0.002s
sys 0m0.025s
theon$
```

- Jetzt schneidet die auf Systemaufrufen basierende Fassung deutlich besser ab.
- Das liegt insbesondere daran, dass die *stdio*-Fassung zeichenweise operiert. Zwar gibt es ebenfalls wenige Systemaufrufe dank der Pufferung der *stdio*, aber der CPU-Aufwand für das zeichenweise Kopieren bleibt dennoch.

mcopy.c

```
struct stat statbuf; if (fstat(infd, &statbuf) < 0) die(infile);
off_t nbytes = statbuf.st_size;
char* buf = (char*) mmap(0, nbytes, PROT_READ, MAP_SHARED, infd, 0);
if (buf == MAP_FAILED) die(infile);
ssize_t count;
for (ssize_t written = 0; written < nbytes; written += count) {
   count = write(outfd, buf + written, nbytes - written);
   if (count <= 0) die(outfile);
}
```

- Der Systemaufruf *mmap* (*memory map*) erlaubt es, den Inhalt des Puffer-Cache, der zu einer Datei gehört, direkt in den eigenen Adressraum zu legen.
- Auf diese Weise entfällt das Kopieren des Inhalts der zu kopierenden Datei in den Adressraum des Kopierprogramms.
- Allerdings sind auch die Anpassungen des virtuellen Adressraums und die laufenden *page faults* auch nicht ohne Kosten.

Vergleich der Kopierprogramme


```
theon$ time scopy 10m out && rm -f out

real 0m0.137s
user 0m0.130s
sys 0m0.007s
theon$ time copy 10m out && rm -f out

real 0m0.027s
user 0m0.002s
sys 0m0.025s
theon$ time mcopy 10m out && rm -f out

real 0m0.019s
user 0m0.001s
sys 0m0.017s
theon$
```

Aufgabenstellung:

- Die Zeilen aus der Standard-Eingabe sind in einer zufälligen Reihenfolge auszugeben.
- Dies sollte möglichst effizient und mit geringem Speicherplatzbedarf geschehen.
- Die Standard-Eingabe muss nicht notwendigerweise eine Datei sein – sie könnte auch beispielsweise aus einer Pipeline kommen.
- Die Zeilen sollen beliebig lange sein können.
- Alle Permutationen sollen mit gleicher Wahrscheinlichkeit ausgewählt werden. Dies ist nicht-trivial, da die Zahl der Permutationen ($n!$ für n Zeilen) rasch die Zahl der möglichen Seed-Werte eines Pseudo-Zufallszahlengenerators übersteigt.

Vorgehensweise:

- Zunächst wird die gesamte Eingabe gelesen und dabei Buch geführt über alle gefundenen Zeilen, jeweils mit Anfangsposition und Zeilenlänge.
- Dies ist die einzige dynamische Datenstruktur, die im Speicher verbleibt.
- Danach werden Zeilen zufällig ausgewählt und ausgegeben.
- Da letzteres nur für Dateien funktioniert, wird bei Bedarf die gesamte Eingabe im ersten Durchgang in eine temporäre Datei kopiert, aus der dann später gelesen wird.

lposlist.h

```
#ifndef LPOSLIST_H
#define LPOSLIST_H

#include <stdbool.h>
#include <sys/types.h>
#include <unistd.h>

typedef struct lpos {
   off_t pos;
   ssize_t len; /* length without line terminator */
} lpos;

typedef struct lposlist {
   int allocated; /* number of allocated entries */
   int length; /* number of actually used entries */
   lpos* line; /* pointer to array of entries */
} lposlist;

bool add_lpos(lposlist* list, off_t pos, ssize_t len);
#endif
```

lposlist.c

```
#include <stdlib.h>
#include <unistd.h>
#include "lposlist.h"

bool add_lpos(lposlist* list, off_t pos, ssize_t len) {
   if (list->length == list->allocated) {
      int allocated = (list->allocated << 1) + 16;
      lpos* new = realloc(list->line, allocated * sizeof(lpos));
      if (!new) return false;
      list->line = new; list->allocated = allocated;
   }
   list->line[list->length++] = (lpos) {pos, len};
   return true;
}
```

- Wenn das erste Argument von *realloc* ein Nullzeiger ist, dann ist der Aufruf äquivalent zu *malloc*.
- Beginnend mit C99 können Strukturen auch innerhalb eines Ausdrucks konstruiert werden. Die Syntax gleicht der Initialisierung. Hinzukommen muss jedoch der Datentyp in Klammern vor den geschweiften Klammern.

rval.h

```
#ifndef RGEN_H
#define RGEN_H

#include <stdbool.h>

bool get_rval(int* rval);
#endif
```

- Die Aufgabe dieser Funktion ist die Generierung von Pseudo-Zufallszahlen, die nicht von einem begrenzten Seed-Wert abhängen.
- Die Funktion liefert 0 zurück, falls es nicht geklappt hat. Ansonsten wird der Zufallswert hinter dem Zeiger abgelegt und 1 zurückgeliefert.

rval.c

```
#include <fcntl.h>
#include <unistd.h>
#include "rgen.h"

bool get_rval(int* rval) {
   static int fd = 0;
   if (fd == 0) {
      fd = open("/dev/urandom", O_RDONLY);
      if (fd < 0) return false;
   }
   ssize_t nbytes = read(fd, rval, sizeof(int));
   return nbytes == sizeof(int);
}
```

- Es bietet sich die spezielle Gerätedatei */dev/urandom* an, die sich aus dem Entropie-Pool des Betriebssystems bedient.
- Alternativ gibt es auch */dev/random*, das aber solange blockiert, bis genügend Zufallswerte höchster Qualität (in Bezug auf Unvorhersehbarkeit) zur Verfügung stehen. Im Vergleich dazu blockiert */dev/urandom* nicht und überbrückt stattdessen mit einem gewöhnlichen Pseudo-Zufallszahlengenerator.

tmpfile.h

```
#ifndef TMPFILE_H
#define TMPFILE_H

int get_tmpfile();
#endif
```

- Es ist möglich, eine Datei anzulegen, sie mit *unlink* sofort wieder aus dem Verzeichnis zu entfernen und den Dateideskriptor zu behalten.
- Auch wenn dann die Datei nirgends im Dateisystem zu sehen ist, so bleibt sie dennoch erhalten, bis der letzte auf sie verweisende Dateideskriptor geschlossen wird.
- Die Funktion *get_tmpfile* legt eine entsprechende temporäre Datei an, entfernt sie gleich wieder und liefert den Dateideskriptor zurück. Die Datei ist sinnvollerweise zum Lesen und Schreiben geöffnet.
- Zugriffsrechte ungleich 0 sind nur sinnvoll, wenn kein *unlink* unmittelbar folgt und andere Prozesse per Namen noch darauf zugreifen sollen.

tmpfile.c

```
#include <fcntl.h>
#include <stralloc.h>
#include <unistd.h>
#include "tmpfile.h"
#include "rgen.h"

int get_tmpfile() {
   stralloc tmpfile = {0};
   for (int attempt = 0; attempt < 10; ++attempt) {
      if (!stralloc_copys(&tmpfile, "/tmp/tmp.")) return -1;
      int rval;
      if (!get_rval(&rval)) return -1;
      if (!stralloc_catint(&tmpfile, rval)) return -1;
      if (!stralloc_0(&tmpfile)) return -1;
      int outfd = open(tmpfile.s, O_RDWR|O_CREAT|O_EXCL, 0);
      if (outfd >= 0) {
         if (unlink(tmpfile.s) < 0) { close(outfd); return -1; }
         return outfd;
      }
   }
   return -1;
}
```

tmpfile.c

```
for (int attempt = 0; attempt < 10; ++attempt) {
   if (!stralloc_copys(&tmpfile, "/tmp/tmp.")) return -1;
   int rval;
   if (!get_rval(&rval)) return -1;
   if (!stralloc_catint(&tmpfile, rval)) return -1;
   if (!stralloc_0(&tmpfile)) return -1;
   int outfd = open(tmpfile.s, O_RDWR|O_CREAT|O_EXCL, 0);
   if (outfd >= 0) {
      if (unlink(tmpfile.s) < 0) { close(outfd); return -1; }
      return outfd;
   }
}
```

- Da */tmp* von vielen gleichzeitig genutzt wird, ist es sinnvoll, möglichst noch nicht gewählte Dateinamen auszuwählen. Dafür bietet sich etwa die Prozess-ID oder eine Zufallszahl an. Mehrere Versuche sollten aber besser einkalkuliert werden.

Einige Alternativen zu *get_tmpfile* aus dem POSIX-Standard:

int *mkstemp*(**char**∗ *template*)

Liefert analog zu *get_tmpfile* einen Dateideskriptor zurück auf eine neu angelegte temporäre Datei, die zum Lesen und Schreiben eröffnet ist. Der Parameter *template* spezifiert den Pfadnamen, der am Ende mit sechs aufeinanderfolgenden „X“ enden sollte. Dieser Parameter wird dann mit dem tatsächlich angelegten Dateinamen aktualisiert. Der Aufruf von *unlink* muss selbst vorgenommen werden.

FILE∗ *tmpfile*(**void**)

Legt analog zu *get_tmpfile* eine temporäre Datei in */tmp* an, eröffnet sie zum Lesen und Schreiben und entfernt den Namenslink in */tmp*.

Wichtige Punkte bei temporären Dateien:

- Sie werden typischerweise in */tmp* oder */var/tmp* angelegt. Auf beide Verzeichnisse haben alle Benutzer Zugriffsrechte.
- Wenn das Anlegen der Datei nicht mit *O_EXCL* erfolgt, besteht die Gefahr eines Symlink-Angriffs, d.h. ein Angreifer könnte einen symbolischen Link in */tmp* anlegen mit Namen, die möglicherweise beim Öffnen verwendet werden. Dann kann es zum unbeabsichtigten Schreiben auf anderswo liegende Dateien kommen.
- Das Aufräumen angelegter temporärer Dateien sollte sichergestellt werden. Wenn möglich und sinnvoll, geschieht dies am besten dadurch, indem ein *unlink* unmittelbar folgt.

lscan.h

```
#ifndef LSCAN_H
#define LSCAN_H

#include <stdbool.h>
#include "lposlist.h"

bool scan_lines(int fd, int out, lposlist* list);
#endif
```

- Die Funktion *scan_lines*
  - liest die gesamte Eingabe aus *fd*,
  - kopiert sie nach *out*, falls *out* nicht-negativ ist, und
  - legt die gefundenen Zeilen unter *list* ab.

lscan.c

```
#include <sys/stat.h>
#include "lposlist.h"
#include "lscan.h"

static off_t get_blocksize(int fd) {
   struct stat statbuf;
   if (fstat(fd, &statbuf) < 0) return 0;
   return statbuf.st_blksize;
}

bool scan_lines(int fd, int out, lposlist* list) {
   off_t blocksize;
   if (out >= 0) {
      blocksize = get_blocksize(out);
   } else {
      blocksize = get_blocksize(fd);
   }
   if (!blocksize) return false;

   char buf[blocksize];
   // ...
}
```

lscan.c

```
static off_t get_blocksize(int fd) {
   struct stat statbuf;
   if (fstat(fd, &statbuf) < 0) return 0;
   return statbuf.st_blksize;
}
```

- Bei regulären Dateien lässt sich über das Feld *st_blksize* die vom zugehörigen Dateisystem für I/O-Operationen bevorzugte Blockgröße ermitteln.
- Diese liegt traditionellerweise bei 4096 oder 8192 Bytes.
- Bei neueren Dateisystemen wie etwa ZFS kann diese Blockgröße bei jeder Datei unterschiedlich (und sehr viel größer) sein.

lscan.c

```
off_t pos = 0;
ssize_t llen = 0; /* length of current line */
off_t blockpos = pos; /* keep track of current position */
ssize_t nbytes;
while ((nbytes = read(fd, buf, blocksize)) > 0) {
   if (out >= 0) {
      ssize_t written = write(out, buf, nbytes);
      if (written < nbytes) return false;
   }
   for (char* cp = buf; cp < buf + nbytes; ++cp) {
      if (*cp == '\n') {
         add_lpos(list, pos, llen);
         pos = blockpos + cp - buf + 1;
         llen = 0;
      } else {
         ++llen;
      }
   }
   blockpos += nbytes;
}
if (nbytes < 0) return false;
if (llen) add_lpos(list, pos, llen);
return true;
```

shuffle.c

```
#include <errno.h>
#include <stdbool.h>
#include <stdlib.h>
#include <stralloc.h>
#include <string.h>
#include "lscan.h"
#include "lposlist.h"
#include "rgen.h"
#include "tmpfile.h"

char* cmdname;
static void memerr() { /* ... */ }
static void die(char* filename, char* message) { /* ... */ }
static int select_line(int noflines) { /* ... */ }
static void print_line(int fd, off_t pos, ssize_t len) { /* ... */ }
static bool seekable(int fd) { /* ... */ }

int main(int argc, char* argv[]) { /* ... */ }
```

shuffle.c

```
static bool seekable(int fd) {
   return lseek(fd, 0, SEEK_CUR) >= 0;
}

int main(int argc, char* argv[]) {
   cmdname = argv[0];
   int fd = 0;
   int out = -1;
   if (!seekable(fd)) {
      out = get_tmpfile();
      if (out < 0) die("tmpfile", 0);
   }
   lposlist list = {0};
   if (!scan_lines(fd, out, &list)) die("scan_lines", 0);
   if (out >= 0) fd = out;
   while (list.length > 0) {
      int i = select_line(list.length);
      print_line(fd, list.line[i].pos, list.line[i].len);
      int j = list.length - 1;
      if (i != j) list.line[i] = list.line[j];
      --list.length;
   }
}
```

shuffle.c

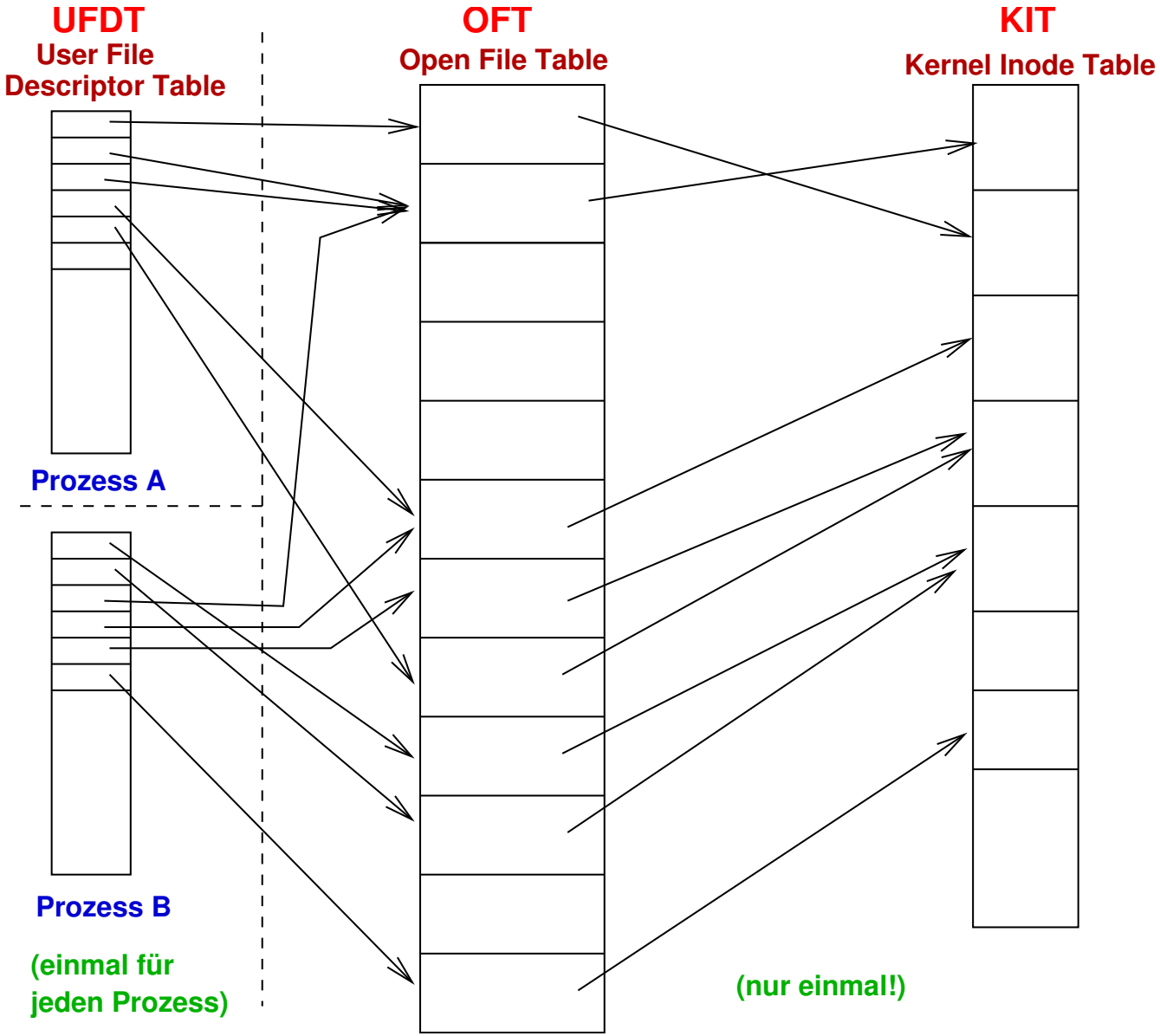
```
static int select_line(int noflines) {
   int selected;
   if (!get_rval(&selected)) die("get_rval", 0);
   if (selected < 0) {
      selected = -(selected+1);
   }
   selected %= noflines;
   return selected;
}
```

- Zu beachten ist hier, dass die Werte von *get_rval* negativ sein können und der Modulo-Operator in C nicht genügt, um den Wert in den Bereich $[0, \text{noflines} - 1)$ zu bringen.

shuffle.c

```
static void print_line(int fd, off_t pos, ssize_t len) {
   char buf[len+1];
   ssize_t copied = 0;
   ssize_t nbytes;
   if (len > 0) {
      if (lseek(fd, pos, SEEK_SET) < 0) die("lseek", 0);
      while (copied < len &&
            (nbytes = read(fd, buf + copied, len - copied)) > 0) {
         copied += nbytes;
      }
      if (nbytes < 0) die("read", 0);
      if (nbytes == 0) die("read", "unexpected end of file");
   }
   buf[len++] = '\n';
   copied = 0;
   while (copied < len &&
         (nbytes = write(1, buf + copied, len - copied)) > 0) {
      copied += nbytes;
   }
   if (nbytes < 0) die("write", 0);
}
```

Die internen Datenstrukturen für Dateien



User File Descriptor Table (UFDT)

- Diese Tabelle gibt es für jeden Prozess.
- Dateideskriptoren dienen als Index zu dieser Tabelle.
- Als Werte hat die Tabelle
  - einen Zeiger in die systemweite *Open File Table* und
  - Optionen, die nur dem Dateideskriptor zugeordnet sind – das ist momentan nur *FD_CLOEXEC*, mit dem Dateideskriptoren automatisiert beim Aufruf des Systemaufrufs *exec* geschlossen werden können. (Diese Option kann mit dem Systemaufruf *fcntl* und dem Parameter *F_GETFD* bzw. *F_SETFD* angesehen bzw. verändert werden).

Open File Table (OFT)

- Diese Tabelle gibt es nur einmal global im Betriebssystem.
- Zu einem Eintrag gehören folgende Komponenten:
  - Ein Zeiger in die *Kernel Inode Table*.
  - Die Optionen, die bei *open* angegeben wurden und später durch *fcntl* und dem Parameter *F_GETFL* bzw. *F_SETFL* angesehen bzw. verändert werden können.
  - Die aktuelle Dateiposition.
  - Eine ganze Zahl, die die Zahl der Verweise aus der UFDT auf den jeweiligen Eintrag spezifiziert. Geht diese Zahl auf 0 zurück, kann der entsprechende Eintrag freigegeben werden.

Kernel Inode Table (KIT)

- Diese Tabelle gibt es nur einmal global im Betriebssystem.
- Jede geöffnete Datei ist in dieser Tabelle genau einmal vertreten.
- Zu einem Eintrag gehören folgende Komponenten:
  - Eine vollständige Kopie der Inode von der Platte.
  - Eine ganze Zahl, die die Zahl der Verweise aus der OFT auf den jeweiligen Eintrag spezifiziert. Solange diese positiv ist, bleibt die Inode auch auf der Platte enthalten, selbst wenn der Referenzzähler innerhalb der Inode auf 0 ist, weil die Datei aus sämtlichen Verzeichnissen entfernt wurde.

Konkurrierende Zugriffe

- Prinzipiell erlaubt Unix den konkurrierenden Zugriff mehrerer Prozesse auf die gleiche Datei.
- Das u.U. notwendige gegenseitige Ausschließen und die Atomizität von Änderungen ergeben sich dabei nicht von selbst, sondern sind Aufgabe der parallel zugreifenden Anwendungen.
- Es gibt aber einige Systemaufrufe, die hier eine Hilfestellung leisten können.

Es ist ein kleines Werkzeug *unique* zu entwickeln, das einen Dateinamen als Parameter erhält und folgende Anforderungen erfüllt:

- Die Zahl in der gegebenen Datei ist auszulesen, um eins zu erhöhen, wieder in die Datei zu schreiben und auf der Standardausgabe auszugeben.
- Gegenseitiger Ausschluss: Jeder Wert darf höchstens einmal ausgegeben werden, egal wieviele Instanzen des Programms gleichzeitig auf die Datei zugreifen.
- Atomizität: Die Datei muss immer einen gültigen Inhalt haben, selbst wenn inmitten einer Operation der Strom ausfällt.

Wenn mehrere gleichzeitig zugreifende Prozesse sich gegenseitig ausschließen möchten, kommen folgende auf dem Dateisystem basierende Techniken in Frage, die alle ohne Interprozess-Kommunikation auskommen:

- Option *O_EXCL* zusammen mit *O_CREAT* bei *open* setzen. Dann ist *open* nur erfolgreich, wenn die Datei vorher noch nicht existiert.
- Mit *link* zu einer existierende Datei einen weiteren Namen hinzufügen. Dies ist nur erfolgreich, wenn der neue Name noch nicht existiert.
- Mit *lockf* können bei einem gegebenen Deskriptor einzelne Bereiche reserviert werden. Jedoch wird *lockf* nicht überall unterstützt oder ist (wie bei NFS) nicht ausreichend zuverlässig.

- Wenn das Ergebnis einer Schreib-Operation abgesichert werden soll, dann empfiehlt sich *fsync*, das einen Dateideskriptor erhält und im Erfolgsfalle wartet, bis der aktuelle Stand auf die Platte gesichert ist.
- Datenbanken und andere Anwendungen arbeiten bei Transaktionen mit mehreren Versionen (der alten und der neuen). Erst wenn die neue Version mit *fsync* abgesichert worden ist, wird ein Versionszeiger in der Datei so aktualisiert, dass er auf die neue Fassung verweist.

Der ISO-Standard für C stellt folgende Funktion zum Umbenennen einer Datei zur Verfügung:

int *rename*(**const char**∗ *old*, **const char**∗ *new*)

Die Datei mit dem Namen *old* wird umbenannt in *new*. Wenn der Name *new* bereits existiert, wird er entfernt (dies ist anders als bei *link*!). Die Atomizität der Operation wird zugesichert. Beide Dateinamen müssen im gleichen Dateisystem liegen.

Dieser Systemaufruf erlaubt es, Dateinamen als „Versionszeiger“ zu verwenden. In Verbindung mit *fsync* lassen sich so Transaktionen umsetzen.

Gegeben sei ein Dateiname, der als Versionszeiger verwendet wird. Dann kann eine Transaktion so ablaufen:

- Eine vollständig neue Version der Daten wird in einer temporären Datei (im gleichen Verzeichnis) erstellt. (Wenn Prozesse sich gegenseitig ausschließen sollen bei dem Erstellen einer neuen Version, sollte der Name der temporären Datei festgelegt werden und das Eröffnen mit *O_EXCL* erfolgen.)
- Diese Daten werden mit *fsync* auf die Platte gesichert.
- Dann wird mit *rename* die temporäre Datei umbenannt in den Dateinamen, der als Versionszeiger verwendet wird.

Beobachter, die den Versionszeiger eröffnen, sehen entweder den konsistenten Stand der alten Version oder den der neuen Version und nie irgendeinen Übergangszustand.

Beispiel: *unique*

unique.c

```
#include <ctype.h>
#include <errno.h>
#include <fcntl.h>
#include <stdbool.h>
#include <stdlib.h>
#include <stralloc.h>
#include <string.h>
#include <unistd.h>

char* cmdname;
stralloc tmpfile = {0}; bool tmpfile_created = false;

/* print an out of memory message to standard error and exit */
void memerr() { /* ... */ }

/* print a error message to standard error and exit;
   include "message" in the output message, if not 0,
   otherwise strerror(errno) is being used
*/
void die(char* filename, char* message) { /* ... */ }

int main(int argc, char* argv[]) { /* ... */ }
```

Beispiel: *unique*

unique.c

```
/* print an out of memory message to standard error and exit */
void memerr() {
   static char memerrmsg[] = "out of memory error\n";
   write(2, memerrmsg, sizeof(memerrmsg) - 1);
   if (tmpfile_created) unlink(tmpfile.s);
   exit(1);
}
```

- Sollte tatsächlich der Speicher ausgehen, dann sollte die Ausgabe der zugehörigen Fehlermeldung ohne dynamische Speicheranforderungen auskommen.
- Von **sizeof**(*memerrmsg*) wird 1 abgezogen, weil das Nullbyte nicht auszugeben ist.
- Wenn die Ausführung abgebrochen wird, sollten ggf. temporäre Dateien aufgeräumt werden. Mit *unlink* kann eine Verweis aus einem Verzeichnis auf eine Datei entfernt werden.

Beispiel: *unique*

unique.c

```
/* print a error message to standard error and exit;
   include "message" in the output message, if not 0,
   otherwise strerror(errno) is being used
*/
void die(char* filename, char* message) {
   stralloc msg = {0};
   if (stralloc_copys(&msg, cmdname) &&
         stralloc_cats(&msg, ": ") && (
            message?
               stralloc_cats(&msg, message)
            :
               stralloc_cats(&msg, strerror(errno))
         ) && stralloc_cats(&msg, ": ") &&
         stralloc_cats(&msg, filename) &&
         stralloc_cats(&msg, "\n")) {
      write(2, msg.s, msg.len);
   } else {
      memerr();
   }
   if (tmpfile_created) unlink(tmpfile.s);
   exit(1);
}
```

Beispiel: *unique*

unique.c

```
int main(int argc, char* argv[]) {
   /* process command line arguments */

   /* try to open the temporary file which also serves as a lock */

   /* determine current value of the counter */

   /* increment the counter and write it to the tmpfile */

   /* update counter file atomically by a rename */

   /* write counter value to stdout */
}
```

- Vorgehensweise: Wir erhalten einen Dateinamen als Argument, leiten daraus den Namen einer temporären Datei ab, eröffnen diese exklusiv zum Schreiben, lesen den alten Zählerwert aus, erhöhen diesen um eins, schreiben den neuen Zählerwert in die temporäre Datei, taufen diese in den gegebenen Dateinamen um und geben am Ende den neuen Zählerwert aus.

Beispiel: *unique*

unique.c

```
/* process command line arguments */
cmdname = argv[0];
if (argc != 2) {
   stralloc usage = {0};
   if (stralloc_copys(&usage, "Usage: ") &&
         stralloc_cats(&usage, cmdname) &&
         stralloc_cats(&usage, " counter\n")) {
      write(2, usage.s, usage.len);
   } else {
      memerr();
   }
   exit(1);
}
char* counter_file = argv[1];
```

- Genau ein Dateiname wird als Argument erwartet. In dieser Datei wird der Zähler verwaltet.

Beispiel: *unique*

unique.c

```
/* try to open the temporary file which also serves as a lock */
if (!stralloc_copys(&tmpfile, counter_file) ||
      !stralloc_cats(&tmpfile, ".tmp") ||
      !stralloc_0(&tmpfile)) {
   memerr();
}
int outfd;
for (int tries = 0; tries < 10; ++tries) {
   outfd = open(tmpfile.s, O_WRONLY|O_CREAT|O_EXCL, 0666);
   if (outfd >= 0) break;
   if (errno != EEXIST) break;
   sleep(1);
}
if (outfd < 0) die(tmpfile.s, 0);
tmpfile_created = true;
```

- Den Namen der temporären Datei gewinnen wir durch ein Anhängen der Endung ».tmp« an den übergebenen Dateinamen.
- Damit liegt die temporäre Datei im gleichen Verzeichnis wie die angegebene Datei und damit auch auf dem gleichen Dateisystem.
- Das Nullbyte am Ende der Zeichenkette *tmpfile* wird für *open* benötigt.

Beispiel: *unique*

unique.c

```
int outfd;
for (int tries = 0; tries < 10; ++tries) {
   outfd = open(tmpfile.s, O_WRONLY|O_CREAT|O_EXCL, 0666);
   if (outfd >= 0) break;
   if (errno != EEXIST) break;
   sleep(1);
}
```

- Die Option *O_EXCL* lässt den Aufruf von *open* scheitern, wenn die Datei bereits existiert. In diesem Falle hat *errno* den Wert *EEXIST*.
- Wenn *open* aus diesem Grunde schiefgeht, wird die Operation mit Zeitverzögerung wiederholt. *sleep* erlaubt ein sekundengenaues Suspendieren des eigenen Prozesses.
- Sobald der Aufruf von *open* erfolgreich ist, schließen wir alle Konkurrenten aus.

Beispiel: *unique*

unique.c

```
/* determine current value of the counter */
int current_value;
int infd = open(counter_file, O_RDONLY);
if (infd >= 0) {
   char buf[512];
   ssize_t nbytes = read(infd, buf, sizeof buf);
   if (nbytes <= 0) die(counter_file, 0);
   current_value = 0;
   for (char* cp = buf; cp < buf + nbytes; ++cp) {
      if (!isdigit(*cp)) die(counter_file, "decimal digits expected");
      current_value = current_value * 10 + *cp - '0';
   }
} else if (errno != ENOENT) {
   die(counter_file, 0);
} else {
   /* start a new counter */
   current_value = 0;
}
```

- Sobald wir einen exklusiven Zugriff haben, lohnt es sich, den bisherigen Zählerstand auszulesen.
- Falls die Datei noch nicht existiert, gehen wir von einem bisherigen Zählerwert von 0 aus.

Beispiel: *unique*

unique.c

```
/* increment the counter and write it to the tmpfile */
++current_value;
stralloc outbuf = {0};
if (!stralloc_copys(&outbuf, "") ||
      !stralloc_catint(&outbuf, current_value)) {
   memerr();
}
int nbytes = write(outfd, outbuf.s, outbuf.len);
if (nbytes < outbuf.len) die(tmpfile.s, 0);
if (fsync(outfd) < 0) die(tmpfile.s, 0);
if (close(outfd) < 0) die(tmpfile.s, 0);
```

- Der um eins erhöhte Zählerwert wird in die temporäre Datei geschrieben.
- Mit *fsync* wird der Inhalt der temporären Datei mit der Festplatte synchronisiert.

Beispiel: *unique*

unique.c

```
/* update counter file atomically by a rename */
if (rename(tmpfile.s, counter_file) < 0) die(counter_file, 0);
tmpfile_created = false;
```

- Mit *rename* wird der Verweis auf die Zieldatei, falls dieser zuvor existierte, implizit mit *unlink* entfernt und danach die temporäre Datei in die Zieldatei umgetauft.
- IEEE Std 1003.1 verlangt ausdrücklich, dass *rename* atomar ist. Dies ist eine Präzisierung im Vergleich zu ISO 9989-2011 (C11-Standard), das den Fall, dass die Zieldatei existiert, ausdrücklich offen lässt.

unique.c

```
/* write counter value to stdout */
if (!stralloc_cats(&outbuf, "\n")) memerr();
nbytes = write(1, outbuf.s, outbuf.len);
if (nbytes < outbuf.len) die("stdout", 0);
```

- Am Ende wird hier, falls alles soweit erfolgreich war, der neue Zählerwert auf der Standard-Ausgabe ausgegeben.

Folgende Nachteile sind mit dem vorgestellten Beispiel verbunden:

- Sollte das Programm gewaltsam terminiert werden, während die temporäre Datei noch existiert, kommt keine weitere Instanz mehr zum Zuge, da alle darauf warten, dass diese irgendwann verschwindet. Das Problem kann dahingehend angegangen werden, dass die anderen Instanzen überprüfen, ob derjenige, der dem die Datei gehört, noch lebt. Dies ist möglich, wenn die Prozess-ID bekannt ist und der Prozess auf dem gleichen Rechner läuft. Andernfalls läuft es nur über Netzwerkprotokolle oder über Heuristiken, die eine zeitliche Beschränkung einführen.
- Eine Wartezeit von einer Sekunde ist recht grob. Kleinere Wartezeiten sind mit Hilfe des Systemaufrufs *poll* möglich.

unique2.c

```
void randsleep() {
   static int invocations = 0;
   if (invocations == 0) {
      srand(getpid());
   }
   ++invocations;
   /* determine timeout value (in milliseconds) */
   int timeout = rand() % (10 * invocations + 100);
   if (poll(0, 0, timeout) < 0) die("poll", 0);
}
```

- *poll* blockiert den aufrufenden Prozess bis zum Eintreffen eines Ereignisses (aus einer Menge gegebener Ereignisse im Kontext von Dateideskriptoren) oder wenn ein Zeitlimit abgelaufen ist.
- Das Zeitlimit wird in Millisekunden als ganze Zahl spezifiziert.
- Im einfachsten Falle kann *poll* wie hier auch als reines Suspendierungs-Werkzeug verwendet werden, das im Gegensatz zu *sleep* Zeitangaben in Millisekunden akzeptiert.
- Wie genau das jedoch aufgelöst wird, hängt vom Betriebssystem ab.

Konkurrierende Zugriffe auf eine Datei

- Grundsätzlich können beliebig viele Prozesse gleichzeitig auf die gleiche Datei zugreifen.
- Eine Synchronisierung oder Koordinierung bleibt grundsätzlich den Anwendungen überlassen.
- Es gibt aber einen entscheidenden Punkt: Arbeiten die konkurrierenden Prozesse mit unabhängig voneinander geöffneten Dateideskriptoren oder sind die Dateideskriptoren gemeinsamen Ursprungs?
- Dateideskriptoren können vererbt werden. Bei der Shell wird dies intensiv ausgenutzt, um beispielsweise die Standard-Kanäle im gewünschten Sinne vorzubereiten.
- Zu jedem Dateideskriptor gibt es eine aktuelle Position. Wenn Dateideskriptoren vererbt werden, arbeiten alle Erben mit der gleichen Position.

write10.c

```
int main(int argc, char* argv[]) {
   cmdname = argv[0];
   for (int i = 1; i <= 10; ++i) {
      stralloc text = {0};
      if (!stralloc_copys(&text, "")) memerr();
      if (!stralloc_catint(&text, getpid())) memerr();
      if (!stralloc_cats(&text, ": ")) memerr();
      if (!stralloc_catint(&text, i)) memerr();
      if (!stralloc_cats(&text, "\n")) memerr();
      ssize_t nbytes = write(1, text.s, text.len);
      if (nbytes < text.len) die("stdout", 0);
   }
}
```

- Dieses Programm ruft 10 mal *write* auf, um die eigene Prozess-ID zusammen mit einer laufenden Nummer auf der Standardausgabe auszugeben.
- Dies dient im folgenden als Testkandidat.

Testfall 1

test1

```
#!/bin/sh

rm -f out

./write10 >out & ./write10 >out & ./write10 >out &
./write10 >out & ./write10 >out & ./write10 >out &
./write10 >out & ./write10 >out & ./write10 >out &
./write10 >out & ./write10 >out & ./write10 >out &
./write10 >out & ./write10 >out & ./write10 >out &
./write10 >out & ./write10 >out & ./write10 >out &
./write10 >out & ./write10 >out & ./write10 >out &
./write10 >out & ./write10 >out & ./write10 >out &
./write10 >out & ./write10 >out & ./write10 >out &
./write10 >out & ./write10 >out & ./write10 >out &
```

- Hier wird das Testprogramm 30 mal aufgerufen und dabei jeweils individuell die Ausgabedatei zum Schreiben eröffnet.
- Das Eröffnen erfolgt durch die Shell mit den Optionen *O_WRONLY*, *O_CREAT* und *O_TRUNC*.
- Jedes Programm arbeitet mit einem eigenen unabhängigen Dateideskriptor, der jeweils ab Position 0 beginnt.

Testfall 2

test2

```
#!/bin/sh

rm -f out

./write10 >>out & ./write10 >>out & ./write10 >>out &
./write10 >>out & ./write10 >>out & ./write10 >>out &
./write10 >>out & ./write10 >>out & ./write10 >>out &
./write10 >>out & ./write10 >>out & ./write10 >>out &
./write10 >>out & ./write10 >>out & ./write10 >>out &
./write10 >>out & ./write10 >>out & ./write10 >>out &
./write10 >>out & ./write10 >>out & ./write10 >>out &
./write10 >>out & ./write10 >>out & ./write10 >>out &
./write10 >>out & ./write10 >>out & ./write10 >>out &
./write10 >>out & ./write10 >>out & ./write10 >>out &
```

- Hier wird von der Shell die Ausgabedaei wiederum jeweils individuell zum Schreiben eröffnet.
- Aber diesmal fällt die Option *O_TRUNC* weg.
- Stattdessen positioniert die Shell den Dateideskriptor an das aktuelle Ende.
- Nach wie vor arbeitet jeder der aufgerufenen Prozesse mit einer eigenen Dateiposition.

Testfall 3

test3

```
#!/bin/sh

rm -f out
exec >out

./write10 & ./write10 & ./write10 &
./write10 & ./write10 & ./write10 &
./write10 & ./write10 & ./write10 &
./write10 & ./write10 & ./write10 &
./write10 & ./write10 & ./write10 &
./write10 & ./write10 & ./write10 &
./write10 & ./write10 & ./write10 &
./write10 & ./write10 & ./write10 &
./write10 & ./write10 & ./write10 &
./write10 & ./write10 & ./write10 &
```

- Hier eröffnet die Shell die Ausgabedatei genau einmal zu Beginn im Rahmen der *exec*-Anweisung.
- Dieser Dateideskriptor wird danach an alle aufgerufenen Prozesse vererbt.
- Entsprechend arbeiten alle Prozesse mit einer gemeinsamen Dateiposition.

```
turing$ ./testit1
turing$ wc -l out
     10 out
turing$ ./testit2
turing$ wc -l out
     50 out
turing$ ./testit2
turing$ wc -l out
     29 out
turing$ ./testit3
turing$ wc -l out
    300 out
turing$
```

- Nur im dritten Falle geht hier keine Ausgabe verloren. Allerdings könnte diese bunt gemischt sein.
- IEEE Std 1003.1 garantiert allerdings hier nicht die Verlustfreiheit, weil das Betriebssystem nicht notwendigerweise entsprechend intern synchronisiert. In der Praxis kann dies allerdings bei *write* dennoch klappen. Bei *read* wird aus Performance-Gründen jedoch weitgehend darauf verzichtet.

Atomares Schreiben mit *O_APPEND*

write10app.c

```
int fd = open(argv[1], O_WRONLY|O_APPEND|O_CREAT, 0666);
```

- Wenn das Mischen von Ausgaben atomar erfolgen soll, ohne dass es zu Verlusten kommt, empfiehlt sich die Verwendung des Flags *O_APPEND*, das bei *open* gesetzt werden kann.
- Zitat aus dem IEEE Std 1003.1:

  > *If the O_APPEND flag of the file status flags is set, the file offset shall be set to the end of the file prior to each write and no intervening file modification operation shall occur between changing the file offset and the write operation.*

- Die Shell verwendet bei „>>“ jedoch nicht das Flag *O_APPEND*, sondern positioniert sich nur mit *lseek* an das Ende der Datei.

Gepufferte Ein- und Ausgabe ist sinnvoll, um die Zahl der Systemaufrufe zu reduzieren. Standardmäßig wird in C als Lösung hierfür die *stdio* verwendet.

Frage: Ist *stdio* gut genug für alle Bedürfnisse?

Antwort: Nein, da sie keine Funktion anbietet, die analog zu dem Systemaufruf *read* sich darauf beschränkt, genau das zu liefern, was bereits ohne Zeitverzug vorliegt.

Ja, das ist in der Tat ein Problem beim Einlesen über das Netzwerk, beim nicht-blockierenden Lesen und auch bei Einlese-Operationen, die mit Hilfe von regulären Ausdrücken spezifiziert werden.

Gepufferte Eingabe minimalistisch

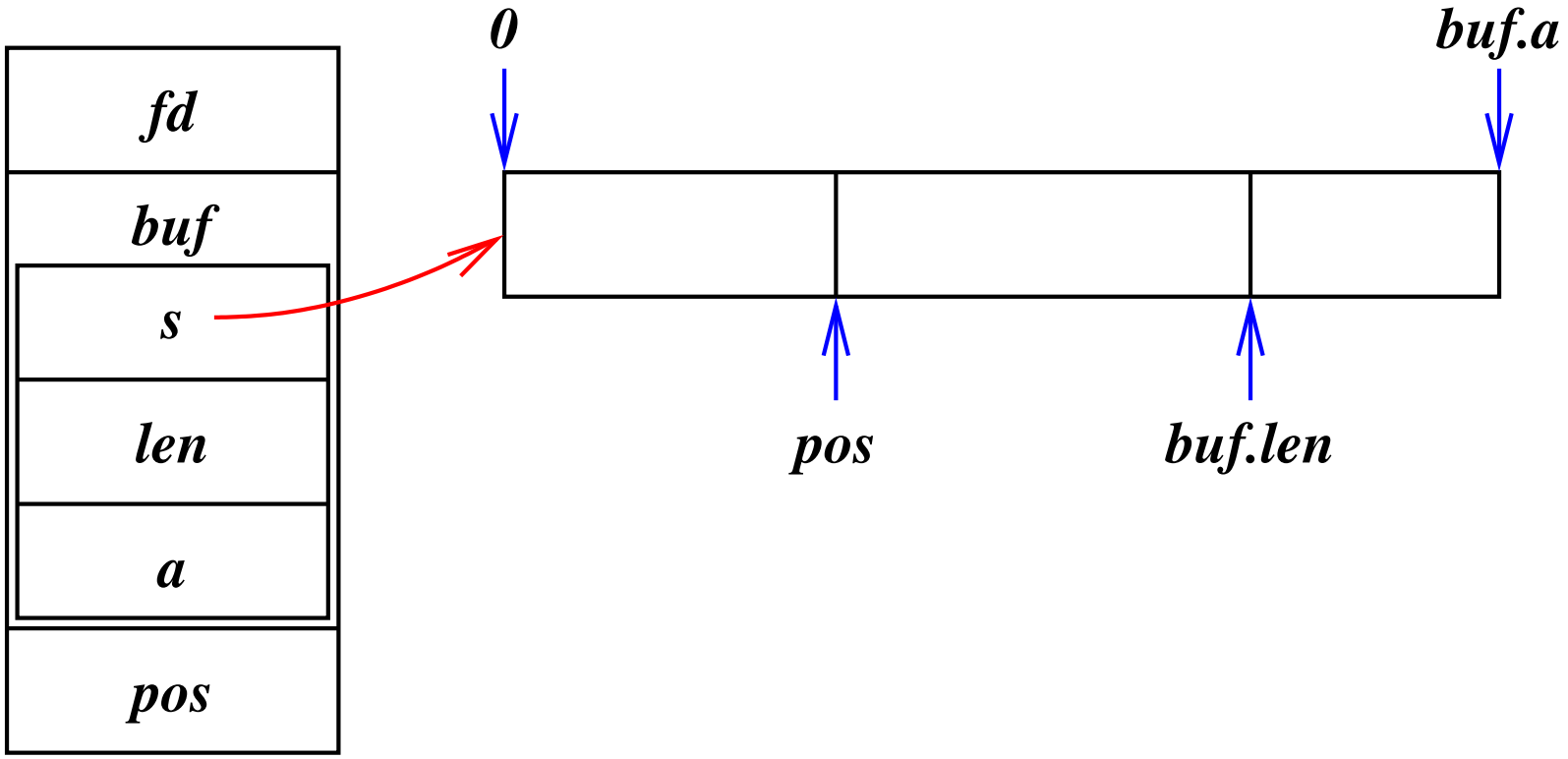
Was ist minimal notwendig für eine gepufferte Eingabe?

- Ein Puffer mit Informationen, wie groß dieser ist, wieweit dieser gefüllt wurde und bis wohin dieser gelesen worden ist.
- Eine Funktion, die einen leeren Puffer auffüllt mit genau einem Systemaufruf von *read*.
- Eine Funktion analog zu *read*, die, falls notwendig, den leeren Puffer auffüllt, und dann den Aufrufer nur aus dem gefüllten Puffer bedient.

- Entsprechend ist eine gepufferte Eingabe notwendig, bei der die Eingabe-Operationen aus einem Puffer versorgt werden, der, wenn er leer wird, mit Hilfe einer *read*()-Operation aufzufüllen ist.
- Die Datenstruktur für einen Eingabe-Puffer benötigt entsprechend einen Dateideskriptor, einen Puffer und einen Positionszeiger innerhalb des Puffers:

inbuf.h

```
typedef struct inbuf {
   int fd;
   stralloc buf;
   size_t pos;
} inbuf;
```


0
buf.a
fd
buf
s
len
a
pos
pos
buf.len

Schnittstelle für eine gepufferte Eingabe

inbuf.h

```
#ifndef INBUF_H
#define INBUF_H

#include <stddef.h>
#include <stralloc.h>
#include <unistd.h>

typedef struct inbuf {
   int fd;
   stralloc buf;
   size_t pos;
} inbuf;

/* set size of input buffer */
int inbuf_alloc(inbuf* ibuf, size_t size);

/* works like read(2) but from ibuf */
ssize_t inbuf_read(inbuf* ibuf, void* buf, size_t size);

/* works like fgetc but from ibuf */
int inbuf_getchar(inbuf* ibuf);

/* move backward one position */
int inbuf_back(inbuf* ibuf);

/* release storage associated with ibuf */
void inbuf_free(inbuf* ibuf);

#endif
```

inbuf.c

```
/* set size of input buffer */
int inbuf_alloc(inbuf* ibuf, size_t size) {
   return stralloc_ready(&ibuf->buf, size);
}

/* works like read(2) but from ibuf */
ssize_t inbuf_read(inbuf* ibuf, void* buf, size_t size) {
   if (size == 0) return 0;
   if (ibuf->pos >= ibuf->buf.len) {
      if (ibuf->buf.a == 0 && !inbuf_alloc(ibuf, 512)) return -1;
      /* fill input buffer */
      ssize_t nbytes;
      do {
         errno = 0;
         nbytes = read(ibuf->fd, ibuf->buf.s, ibuf->buf.a);
      } while (nbytes < 0 && errno == EINTR);
      if (nbytes <= 0) return nbytes;
      ibuf->buf.len = nbytes;
      ibuf->pos = 0;
   }
   ssize_t nbytes = ibuf->buf.len - ibuf->pos;
   if (size < nbytes) nbytes = size;
   memcpy(buf, ibuf->buf.s + ibuf->pos, nbytes);
   ibuf->pos += nbytes;
   return nbytes;
}
```

Implementierung für eine gepufferte Eingabe

inbuf.c

```
/* works like fgetc but from ibuf */
int inbuf_getchar(inbuf* ibuf) {
   char ch;
   ssize_t nbytes = inbuf_read(ibuf, &ch, sizeof ch);
   if (nbytes <= 0) return -1;
   return ch;
}

/* move backward one position */
int inbuf_back(inbuf* ibuf) {
   if (ibuf->pos == 0) return 0;
   ibuf->pos--;
   return 1;
}

/* release storage associated with ibuf */
void inbuf_free(inbuf* ibuf) {
   stralloc_free(&ibuf->buf);
}
```

- Reguläre Ausdrücke spezifizieren Sprachen, die mit endlichen Automaten syntaktisch anaylisiert werden können.
- Mit regulären Ausdrücken bzw. regulären Sprachen lassen sich die Grundsymbole einer Programmiersprache oder eines Eingabeformats spezifizieren wie z.B. das Format für eine Konstante in Gleitkommadarstellung.
- Reguläre Ausdrücke werden seit frühesten Zeiten in UNIX unterstützt: *ed*, *sed*, *grep*.
- Reguläre Ausdrücke sind im Rahmen des POSIX-Standards standardisiert:
  `http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap09.html`
- Reguläre Ausdrücke wurden von Perl deutlich erweitert und diese Erweiterungen werden dank des Erfolgs der PCRE-Bibliothek von vielen Werkzeugen unterstützt:
  `http://www.pcre.org`

Metazeichen in regulären Ausdrücken

Kleine Übersicht der Metazeichen:

„\“ nimmt dem folgenden nicht-alphanumerischen Zeichen die Sonderfunktion.

„^“ Anfang der Zeile.

„$“ Ende einer Zeile.

„.“ Beliebiges Zeichen außer dem Zeilentrenner.

„|“ Alternation. Beispiel: „foo|bar“ trifft sowohl für „foo“ als auch „bar“ zu.

„()“ Gruppierung. Beispiel: „(foo|bar)∗“ trifft für „“, „foo“, „bar“, „foofoo“, „barfoo“, „barbar“ usw. zu.

„[]“ Zeichenbereiche. Beispiel: „[A-Za-z]“ (alle Klein- und Großbuchstaben).

- Dies entspricht dem Umfang von *egrep*.
- Im Gegensatz zu *vim* gelten die Klammern als Metasymbole.
- Es empfiehlt sich, alle nicht-alphanumerischen Zeichen, die sich selbst repräsentieren sollen, durch ein „\“ zu schützen.
- „\“ gefolgt von einem alphanumerischen Zeichen nimmt nicht dem alphanumerischen Zeichen seine (nicht vorhandene) Sonderbedeutung, sondern – ganz im Gegenteil – verleiht eine Sonderbedeutung (siehe noch folgende Übersicht).

Quantifikatoren in regulären Ausdrücken

Quantifikatoren beziehen sich auf das Zeichen, die Gruppierung oder den Zeichenbereich, der dem Quantifikator vorausgeht.

„$*$“	0 bis beliebig oft.
„?“	Optional: 0 oder 1 mal.
„$+$“	Mindestens einmal, ansonsten beliebig oft.
„$\{n\}$“	Genau n mal.
„$\{n,\}$“	Mindestens n mal.
„$\{n,m\}$“	Zwischen n und m mal.

- Normalerweise sind reguläre Ausdrücke „gierig“, das heisst sie versuchen, soviel Text wie möglich zu erfassen.
- Perl bzw. Perl-kompatible reguläre Ausdrücke unterstützen auch noch die Angabe eines „?“ hinter einem Quantifikator, um eine minimale Erfassung zu erreichen.

Sonderzeichen in regulären Ausdrücken

Perl bzw. Werkzeuge mit perl-kompatiblen regulären Ausdrücken unterstützen u.a. folgende Sonderzeichen (siehe *man perlre*):

„\t“	Tabulator.
„\n“	Zeilentrenner.
„\r“	Wagenrücklauf (CR).
„\f“	Papiervorschub (FF).
„\a“	Alarmglocke (BEL).
„\e“	Fluchtzeichen (ESC).
„\033“	Oktaldarstellung eines Zeichens.
„\x1b“	Hexdarstellung eines Zeichens.
„\cx“	Control-X.
„\N{*name*}“	Benanntes Zeichen, siehe *perldoc charnames*.
„\l“	Folgendes Zeichen in einen Kleinbuchstaben verwandeln.
„\u“	Folgendes Zeichen in einen Großbuchstaben verwandeln.
„\L“	Bis zu „\E“ alles in Kleinbuchstaben verwandeln.
„\U“	Bis zu „\E“ alles in Großbuchstaben verwandeln.
„\Q“	Bis zu „\E“ allen Metazeichen die Sonderbedeutung nehmen.

Perl und perl-kompatible Werkzeuge unterstützen diverse Kurzformen. Eine Auswahl:

„\w“	Erfasst alle alphanumerischen Zeichen einschließlich dem Unterstrich „_“.
„\W“	Erfasst alle Zeichen, die „\w“ nicht erfasst.
„\s“	Erfasst alle Leerzeichen.
„\S“	Erfasst alle Zeichen, die nicht zu den Leerzeichen gehören.
„\d“	Alle Ziffern.
„\D“	Alle Zeichen, die nicht zu den Ziffern gehören.

- Mit „\w“ wird noch kein Wort, sondern nur ein Zeichen eines Wortes erfasst. Jedoch lässt sich ersteres leicht mit „\w+“ erreichen.

```
#include <regex.h>

int regcomp(regex_t *restrict preg, const char *restrict pattern,
       int cflags);
size_t regerror(int errcode, const regex_t *restrict preg,
       char *restrict errbuf, size_t errbuf_size);
int regexec(const regex_t *restrict preg, const char *restrict string,
       size_t nmatch, regmatch_t pmatch[restrict], int eflags);
void regfree(regex_t *preg);
```

- Der POSIX-Standard unterstützt diese Funktionen.
- Mit *regcomp* wird ein regulärer Ausdruck übersetzt, mit *regexec* ausgeführt.
- *regexec* erwartet in *string* den vollständigen Text, auf den der reguläre Ausdruck bezogen wird.
- Es gibt hier keine teilweise Erfassung eines regulären Ausdrucks mit möglicher Fortsetzung.

mygrep5.c

```
/* compile pattern */
regex_t regex; /* compiled regular expression */
unsigned int regex_flags = REG_NOSUB;
if (opt_i) {
   regex_flags |= REG_ICASE; /* ignore case */
}
if (opt_e) {
   regex_flags |= REG_EXTENDED; /* supported egrep syntax */
}
unsigned int regex_error = regcomp(&regex, pattern, regex_flags);
if (regex_error) {
   char errbuf[128];
   regerror(regex_error, &regex, errbuf, sizeof errbuf);
   fprintf(stderr, "%s: invalid regular expression: %s\n",
      cmdname, errbuf);
   return 1;
}
```

- Reguläre Ausdrücke müssen zuerst mit *regcomp* in eine interne Datenstruktur (den endlichen Automaten) übersetzt werden.

mygrep5.c

```
if ((regexec(&regex, line, 0, 0, 0) != 0) == opt_v) {
```

- Anschließend kann mit *regexec* der Automat auf eine Vielzahl von Eingaben verwendet werden.

PCRE-Bibliothek

- Da die Erweiterungen von Perl für reguläre Ausdrücke sehr populär sind, wurde die entsprechende Bibliothek als selbständige Lösung herausfaktorisiert und weiter entwickelt, so dass sie auch für andere Anwendungen genutzt werden kann.
- Die PCRE-Bibliothek unterstützt partielle Erfassungen.
- `http://www.pcre.org`

Der prinzipielle Algorithmus:

1. Ein temporärer Puffer wird angelegt (als *stralloc*-Objekt),
2. Der Eingabe-Puffer wird gefüllt, falls er leer sein sollte.
3. Der gesamte Inhalt des Eingabe-Puffers wird an den temporären Puffer angehängt.
4. Der temporäre Puffer wird *pcre_exec* übergeben mit der Option *PCRE_PARTIAL_HARD*.
5. Im Erfolgsfalle lässt sich ermitteln, wieviel von der Eingabe von dem regulären Ausdruck erfasst wurde. Die Einlese-Position wird entsprechend weitergesetzt und das Einlesen ist beendet.
6. Wenn *pcre_exec* mit einem anderen Fehler als *PCRE_ERROR_PARTIAL* endet, wurde der reguläre Ausdruck in der Eingabe nicht erkannt. Entsprechend wird das Einlesen mit Fehler abgebrochen.
7. Beim Fehler *PCRE_ERROR_PARTIAL* wird der gesamte Eingabe-Puffer als gelesen markiert und die Verarbeitung mit Schritt 2 fortgesetzt.

Schnittstelle von *inbuf_scan*


```
#include <afblib/inbuf_scan.h>

int inbuf_scan(inbuf* ibuf, const char* regexp, ...);

typedef struct {
   const char* captured;
   size_t captured_len;
   int callout_number;
} inbuf_scan_callout_block;

typedef int (*inbuf_scan_callout_function)(inbuf_scan_callout_block*,
      void* callout_data);

int inbuf_scan_with_callouts(inbuf* ibuf, const char* regexp,
      inbuf_scan_callout_function callout, void* callout_data);
```

- „. . . “ steht für beliebig viele *stralloc*-Objekte, in denen die Eingabe abgelegt wird, die mit geklammerten Teilausdrücken im regulären Ausdruck erfasst worden sind.
- Zurückgeliefert wird -1 im Fehlerfall und die Zahl der gefüllten *stralloc*-Objekte im Erfolgsfall.

Lies eine Eingabezeile und entferne den Zeilentrenner:

```
stralloc line = {0};
int count = inbuf_scan(&ibuf, "(.*)\n", &line);
```

Analog, aber führende und abschließende Leerzeichen werden zusätzlich entfernt:

```
int count = inbuf_scan(&ibuf, "[␣\t]*(.*?)\\s*?\n", &line);
```

Lies die letzte Zeile der Eingabe, überlies alles andere:

```
int count = inbuf_scan(&ibuf, "(?:.*\n)*(.*)\n", &line);
```

Nachteile

Nachteil bzw. Verbesserungsmöglichkeit:

- Inkrementelle Aufrufe von *pcre_exec* stehen leider noch nicht zur Verfügung. Entsprechend muss im temporären Puffer der gesamte bislang gelesene Text gesammelt werden. Dies sollte unnötig sein.
- Leider besteht *pcre_exec* auch bislang noch darauf, dass der gesamte zu untersuchende Text in einem zusammenhängenden Puffer steht.
- Wenn ein zusammenhängender Puffer nicht mehr Voraussetzung wäre, dann könnte es hilfreich sein, mit Pufferketten zu operieren. Auf Systemaufrufebene besteht durchaus Unterstützung von Pufferketten bei den Systemaufrufen *readv* und *writev*.
- Schließlich wäre auch ein Handler bei *pcre_exec* denkbar, der jedesmal aufgerufen wird, wenn mehr Eingabe benötigt wird.

Dann könnte das Umkopieren in einen sich sukzessive vergrößernden temporären Puffer entfallen und die Verabeitung wäre schneller mit Aufwand $O(n)$.