1. Programmieren II (Fortgeschrittene)

Markus Demleitner *(msdemlei@cl.uni-heidelberg.de)*

Worum geht es?

- Die Programmiersprache C
- Entwicklungswerkzeuge
- Einblick in Rechnerarchitektur
- „Gutes“ Programmieren

Technisches

Schein := (Übungen lösen)+(Kleine mündliche Prüfung)

Übungen lösen := Aus 70% der Aufgaben mindestens 50% der erreichbaren Punktzahl hinkriegen.

abgeben := Lösungen rechtzeitig (Termine stehen auf den Übungsblättern) an die Tutorin schicken.

Tutorium

In den Tutorien soll der Stoff der Vorlesung praktisch erprobt werden. Außerdem werden dort die Übungsaufgaben besprochen.

Teilnahme an einem Tutorium ist Voraussetzung für die Scheinvergabe.

Bildet Banden!

Literatur

Literaturangaben zu Vorlesung sind auf einer Wiki-Seite[1] zu finden. Ihr seid herzlich eingeladen, die Seite zu verbessern.

[1] `http://wiki.cl.uni-heidelberg.de/moin/ProgIILiteratur`

2. Goldene Regeln

Wer größere Sachen schreibt, sollte diese Regeln kennen:

Whenever possible, steal code

Brooke's law: Plan to throw one away, you will anyhow

If you lie to a computer, it will get you

Make it work first before you make it work fast

If you have too many special cases, you are doing it wrong

Get your data structures [classes] right first, and the rest of the program will write itself

When in doubt, use brute force

Wenn du es nicht auf Deutsch sagen kannst, kannst du es auch nicht programmieren

nach: Bentley, J. (1988), More Programming Pearls, Addison-Wesley

Noch ein paar mehr Zitate von Größen, die ich bemerkenswert finde:

If it doesn't solve some fairly immediate need, it's almost certainly over-designed. – Linus Torvalds (2004)

Projects promoting programming in „natural language" are intrinsically doomed to fail. – Edsger Dijkstra (1975)

Computer science still seems to be looking for the magic bullet that will cause people to write correct programs without having to think. Instead, we need to teach people how to think – Larry Wall (1999)

Indeed, pleasure has probably the main goal all along. But I hesitate to admit it, because computer scientists want to maintain their image as hard-working individuals who deserve high salaries. – Don Knuth

Rule: „You shouldn't have to open up a black box and take it apart to find out you've been pushing the wrong buttons!" Corollary: „Every black box should have at least TWO blinking lights: Paper Jam and Service Required (or equivalent)." – Steven Majewski

You need to realize that your system is just going to be a module in some bigger system to come, and so you have to be part of something else, and it's a bit of a way of life. – Tim Berners-Lee

You write a great program, regardless of language, by redoing it over & over & over & over, until your fingers bleed and your soul is drained. But if you tell newbies that, they might decide to go off and do something sensible, like bomb defusing – Tim Peters

Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it. – Brian Kernighan

If you can't do a first version in six months with a team of six people it is a sign that you don't really know what you want. – Jack Diederich (2006)

But it could all be Quatsch, Quatsch mit Soße – Robert Weinberg (2006)

It's hard to make a mistake by having too many short and simple functions. And much too easy to make them when you have too few – Thomas Bartkus (2006)

Schöner Code

Manche dieser Regeln kennen wir schon aus Python, andere sind dort nicht nötig, weil sie sich von selbst ergeben. So oder ähnlich möchte man das aber in jeder Programmiersprache haben, ob nun Maschinensprache oder Prolog.

- „Indentation“ (Syntax-sensitive Editoren, verschiedene Stile, Konsistenz)
- Variablennamen (Aussagekräftig und nicht zu lang, `user_input` vs. `userInput` – jedenfalls kleinschreiben, Makros immer groß) – es gibt eine Philosophie, die vorschreiben möchte, dass aus dem Namen einer Variable ihr Typ hervorgehen soll. Das ist Unfug: Wer Code schreibt, in dem die Deklaration nicht ohnehin sehr schnell gefunden werden kann, hat mit oder ohne eine solche Konvention verloren, alle anderen können den Typ an der Deklaration erkennen
- Kommentare (old-style vs. new-style, am besten an jedem Funktionskopf strukturiert und an kritischen Stellen in der Funktion, Embedded Documentation[2], Synchronisation mit Code)
- Factoring (Funktionen i.d.R. kürzer als 20 Zeilen, sinnvolle Aufspaltung des Quelltexts in Module, Information Hiding)

3. Ein Blick über den Tellerrand

Es gibt eine Unzahl von Programmiersprachen, die nach vielen Kriterien eingeteilt werden können, z.B.

- prozedural, funktional, deklarativ, (objektorientiert)
- kompiliert, interpretiert, bytecode-kompiliert
- low-level, high-level
- ...

Im (unrealistischen) „Idealfall“ läuft ein prozedurales Programm Zeile für Zeile durch den Code, und es kommt in jeder Zeile dem Ergebnis durch Manipulation von Daten ein Stück näher. C arbeitet bevorzugt prozedural.

Funktionale Programmierung verzichtet im Idealfall vollständig auf die Seiteneffekte, auf denen das prozedurale Paradigma basiert, das Programm kommt durch verschachtelte Funktionsaufrufe zu seinem Ergebnis. In Python erlauben Funktionen wie map, filter und reduce, aber auch list comprehensions funktionales Programmieren. In C ist funktionales Programmieren aufgrund des Fehlens einer automatischen Speicherverwaltung (und auch eines vernünftigen eingebauten Sequenztyps) eher schwierig.

Idee der deklarativen Programmierung ist, durch eine Spezifikation des Wissens über das Problem der Rechner in die Lage zu versetzen, es lösen zu können. Normales Python bietet ebensowenig wie C eingebaute Mechanismen, um sich in diesem Paradigma zu bewegen.

Objektorientierung ist eine Art Erweiterung des prozeduralen Paradigmas. Zentraler Gedanke ist, Daten mit den auf ihnen operierenden „Methoden“ in einer Einheit zu verbinden und dabei eine bessere Kapselung von Details der Implementation zu sorgen.

Komplierte Sprachen werden vor der Ausführung in Maschinencode verwandelt, interpretierte Sprachen dagegen werden von einem „Laufzeitsystem“ ausgeführt, das das Programm wie geschrieben liest und die Kommandos sozusagen „on the fly“ der Maschine vermittelt.

Bytecode-kompilierte Sprachen erzeugen Code für eine Maschine, die es in Wirklichkeit gar nicht gibt, die aber mittels eines Laufzeitsystems (Emulator, Bytecode-Interpreter) wiederum „on the fly“ der realen Maschine vermittelt werden können.

Low-level heißt, dass sich die Sprache relativ nah an der eigentlichen Architektur der Maschine orientiert, die Maschinensprache sozusagen nur mit etwas „syntaktischem Zucker“ überstreut.

[2] `http://wiki.cl.uni-heidelberg.de/moin/EmbeddedDocumentation`

High-level heißt, dass das, was die Maschine wirklich kann, erstmal nicht interessiert und sich das Design der Sprache an den Bedürfnissen der Problemlösung orientiert.

Alle diese Klassifikationen hinken – z.B. gibt es Mischformen, und für viele Sprachen existieren sowohl Compiler als auch Interpreter.

Ein paar bekannte oder originelle Sprachen

Basic: Prozedural, viele Dialekte, heute vor allem von Microsoft propagiert.

C++: Populäre objektorientierte Sprache, die etwas unter ihren C-Wurzeln leidet.

C#: Java-C-Chimäre von Microsoft.

COBOL: The teaching of COBOL should be made a criminal offense (Dijkstra).

Java: Verschönerung von C++, ziemlich gelungen, immer noch hip.

Javascript: Skriptsprache für lustige Effekte im HTML-Browser. Hat mit Java ziemlich wenig zu tun.

LISP: Die Mutter aller KI-Sprachen, funktional, meistens interpretiert, lebt als Scheme und Elisp weiter.

FORTRAN: Uralt, prozedural, schlimm. NatWis mögen es.

Ruby: Objektorientierte Skriptsprache, quasi eine Fortschreibung von Python zu mehr Reinheit der Ideen.

Pascal: Lehrsprache, prozedural, meistens kompiliert, „C mit Fesseln“; ein wenig aus der Mode.

Perl: Von einem Sprachwiss. entworfene Skriptsprache; vielseitig, chaotisch, bunt.

Prolog: Die KI-ste aller Sprachen: Deklarativ, meistens interpretiert.

Smalltalk: Die reine Lehre der Objektorientierung.

Problems

(3.1)* Sucht im Netz nach Code in einer der hier diskutierten Programmiersprachen (nach Möglichkeit einer, die ihr nicht kennt...). Seht, was ihr davon versteht.

4. Technik: Zahlensysteme

Wir wollen uns in dieser Veranstaltung auf die Maschine zubewegen. Um zu verstehen, wie Computer wirklich mit Zahlen umgehen, muss man bedenken, dass sie nicht zehn, sondern nur einen Finger haben. Deshalb stellen sie Zahlen anders dar als wir.

„Normale“ Zahlen, nämlich Dezimalzahlen, sind zur *Basis* 10 geschrieben:

$$245 = 2 \cdot 10^2 + 4 \cdot 10^1 + 5 \cdot 10^0$$

– eine Zahl wird dargestellt als Summe von Beiträgen von Zehnerpotenzen.

Computer setzen Zahlen als Summen von Zweierpotenzen zusammen:

$$10010100_2 = 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3$$
$$+ 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = 128 + 16 + 4 = 148.$$

Binärzahlen sind also Folgen von Nullen und Einsen, ganz wie Dezimalzahlen Folgen von Dezimalziffern sind. In diesem Skript werden wir Binärzahlen immer mit einem Index 2 notieren.

Um Binärzahlen in Dezimalzahlen zu wandeln, fängt man also an, von rechts bei Null anfangend die Ziffern durchzuzählen und multipliziert die Ziffer jeweils mit der entsprechenden Zweierpotenz.

Vier binäre Ziffern können Zahlen zwischen 0 und 15 beschreiben, und zwei von diesen Vierergruppen sind ein Byte (nämlich 8 bit). Kompromiss zwischen Länge und Computergerechtigkeit: Hexadezimalziffern (besser: Sedezimalziffern). 0-9 werden erweitert um A-F (A=10$\cdots$F=15), und dann ist

$$0x5F = 5 \cdot 16^1 + 15 \cdot 16^0 = 95.$$

Dies ist offenbar ganz analog zum Vorgehen bei Binärzahlen; nur ist eben die Basis 16 und nicht Zwei, und entsprechend werden Sechzehnerpotenzen und nicht Zweierpotenzen verwendet.

Das Hin- und Herrechnen zwischen Zahlensystemen ist mühsam, einE guteR ProgrammiererIn sollte aber wenigstens für die 16 Hex-Ziffern die Bitmuster von 0 = 0000 bis F = 1111 sowie die ersten paar Zweierpotenzen (vielleicht bis 1024) im Kopf haben – es hilft sehr beim Analysieren von Problemen, die einem die Maschine so bereitet.

Es ist trivial, Konversionsfunktionen von Hex oder Binär nach Dezimal zu schreiben (in Python tuts auch `"%d"%0xbaba`). Für sedezimale Zahlen in Strings könnte folgender Python-Code verwendet werden:

```
def hexToInt(hexStr):
  res = 0
  for digit in hexStr:
    res *= 16
    if digit in string.digits:
      res += int(digit)
    elif digit in string.ascii_lowercase:
      res += ord(digit)-ord('a')+10
    elif digit in string.ascii_uppercase:
      res += ord(digit)-ord('A')+10
  return res
```

– kompliziert ist das eigentlich nur, weil wir Groß- und Kleinbuchstaben verarbeiten wollen (wie könnte das mit einem Dictionary implementiert werden?)

Die umgekehrte Prozedur ist auch nicht schwer, wenn man „von hinten“ anfangen darf (Python macht es uns hier einfach):

```
def intToHex(number):
  res = []
  while number>0:
    curRem = number%16
    res.append(string.hexdigits[curRem])
    number /= 16
```

```
    res.reverse()
    return "".join(res)
```

Präfixe

Es ist:

$$2^{10} = 1024 \approx 1000$$
$$2^{20} = 1\,048\,576 \approx 10^6$$
$$2^{30} = 1\,073\,741\,824 \approx 10^9$$

Wegen dieser bequemen Nähe haben Leute angefangen, 2^{10} Bytes als „Kilobytes“ zu bezeichnen und analog für Mega und Giga. Das Problem ist, dass Kilo *exakt* für 10^3 steht und, wenns drauf ankommt, Konfusion herrscht. In diesem Skript halten wir uns an die IEC-Konvention[3], nach der binäre Vielfache mit einem i gekennzeichnet sind. Es sind also $1\,\text{kB} = 10^3$ Bytes, aber $1\,\text{kiB} = 2^{10}$ Bytes und so fort für MB, MiB, GB und GiB.

Gesprochen werden soll kiB übrigens als „kibibyte“. Ich kenne aber niemanden, der/die das tut.

Problems

(4.1)* Erweitert die Funktionen hexToDez und dezToHex so, dass sie für beliebige Basen kleiner als 36 funktionieren. **(L)**

(4.2)* Die guten alten 3 1/2-Zoll-Disketten waren (sind) in High Density auf zwei Seiten zu je 80 Spuren mit je 18 Sektoren zu je 512 Bytes beschrieben. Wie viele Daten passen auf eine so formatierte Diskette? Drückt das in Bytes, kB, kiB, MB und MiB aus. **(L)**

(4.3)* Wie groß ist der relative Unterschied zwischen 1 kiB und 1 kB? Zwischen 1 MiB und 1 MB? Zwischen 1 GiB und 1 GB? Wenn ihr wollt, probiert euch an der Asymptotik (d.h., wie entwickelt sich das Verhältnis von dezimalen und binären Präfixen für immer größere Zahlen?) **(L)**

[3] http://physics.nist.gov/cuu/Units/binary.html

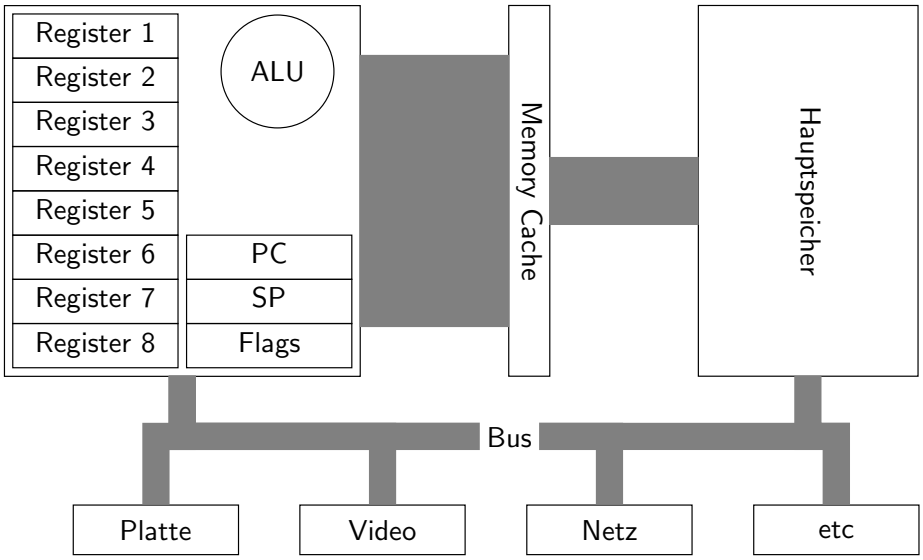


Fig. 1

Von-Neumann-Rechner

Prozessor

CPU

Register

Adresse

Bus

5. Technik: Aufbau eines Rechners

An dieser Stelle wollen wir einen Ausflug in die Niederungen des Rechners machen – das Wissen, wie (jedenfalls ungefähr) ein normaler Rechner aufgebaut ist (man spricht dabei gerne von einem *Von-Neumann-Rechner*, im Gegensatz zu einem Harvard-Rechner, in dem Code und Daten in separaten Speichern liegen) wird helfen, das Design von C zu verstehen.

Stark vereinfachte Skizze eines modernen Rechners:

(cf. Fig. 1)

Er besteht aus dem

- *Prozessor* (der *CPU*, Central Processing Unit), mit allgemeinen und speziellen *Register*n – das sind ein paar Byte Speicher, auf die der Prozessor besonders schnell zugreifen kann und die eventuell auch noch eine spezielle Bedeutung für ihn haben – und der Arithmetic and Logic Unit (ALU), die die eigentliche Rechnung macht (in der Tat haben moderne Prozessoren normalerweise eine Handvoll von denen, und dazu noch eine FPU, die Fließkommarechnungen macht, und dazu noch Gerätschaften, die auf andere Operationen spezialisiert sind, z.B. MMX oder AltiVec),
- dem Hauptspeicher, in dem Daten und Programme liegen (etwa im Normalfall unsere Variablen) – der Speicher ist heute fast immer in Bytes eingeteilt, und jedes Byte hat eine Nummer, seine *Adresse*,
- zwischen CPU und Hauptspeicher einige Caches, deren Funktion ist, der CPU häufig gebrauchte Daten schneller zur Verfügung zu stellen, als es relativ billiger Hauptspeicher kann,
- einem *Bus* zur Kommunikation zwischen CPU und „Peripherie“, also Festplatten, dem Netzwerk, der Tastatur, dem Bildschirm usf. Meist kann die Peripherie auch direkt mit dem Speicher sprechen (DMA, Direct Memory Access), was ProgrammiererInnen im Zweifel das Leben schwer machen kann.

Dieses Bild ist beliebig vereinfacht, ist aber genau genug, um uns später zu helfen, das Verhalten von C-Programmen und auch klassische Kompromisse, die beim Programmieren so anfallen, zu verstehen.

Was sind die Spezialregister?

- PC: Program Counter – die Adresse des augenblicklich ausgeführten Maschinenbefehls.
- SP: Stack Pointer – die Adresse der augenblicklichen „Oberkante“ des Maschinenstacks

- Flags: Eine Sammlung von Bool'schen Informationen. Darin steht z.B., ob bei einer Rechnung ein Überlauf eingetreten ist, ein Übertrag vorliegt, Null herausgekommen ist, welche Rechte der gerade laufende Prozess hat und vieles mehr.

Vieles kommt in diesem Modell nicht vor, etwa Memory Management (die Übersetzung von logische in physikalische Adressen, Speicherschutz) und Interrupts, mit denen z.B. die Peripherie die CPU über Wünsche informieren kann. Fürs erste soll das aber reichen.

Wir werden uns nachher (leider) an der x86-Architektur von Intel orientieren, wenn wir uns CPUs genauer ansehen. Auf dieser haben auch die nicht-speziellen Register lustige Namen, die zum Teil nur noch historische Bedeutung haben.

An für uns relevanten Registern gibt es in solchen CPUs ax (gern auch als Akkumulator bezeichnet, woran noch eine Unzahl von Befehlen erinnert, die nur mit ax gehen), bx, cx (das c sollte mal an Counter erinnern), dx, si (das soll an Source Index errinnern), di (analog Destination Index) und bp (Base Pointer). Diese Register existierten schon in den alten Tagen des 8086, der 16-bit breite Register hatte. Die Namen, wie ich sie jetzt hingeschrieben habe, beziehen sich dann auch nur auf die „unteren" 16 bit dieser Register. Seit dem 80386 haben aber alle x86-Prozessoren (mindestens) 32-bittige Register, die durch Voranstellen eines „e" gekennzeichnet werden, aus ax wird also eax. Im Fließtext werde ich keinen Unterschied zwischen ax und eax machen, im Code natürlich schon.

Tatsächlich gab es etliche dieser Register schon im längst vergessenen 8-bit-Prozessor 8080. Der hat z.B. das ax-Register nochmal zerlegt in ah und al (a high und a low, die oberen und unteren 8 bit). Es ist kaum zu glauben, aber auch der neueste und schickste Pentium 27 kennt noch ah und al von 1974.

Schon das ist alles absurd genug, um zu bedauern, dass die x86-Architektur nicht mit dem 80286 untergegangen sind.

Problems

(5.1) Schraubt euren Rechner auf (ich bin nicht verantwortlich, wenn ihr etwas kaputt macht) und seht (ggf. anhand eines Handbuchs zum Motherboard oder geeigneter Hilfen aus dem Netz, die ihr euch vorher besorgt habt), ob ihr darin die CPU und den Hauptspeicher, vielleicht auch Hinweise, dass es etwas wie einen Bus gibt, vielleicht auch einen Cache findet.

6. Ein erstes C-Programm

```
#include <stdio.h>

int main(void)
{
  printf("Hello World\n");
  return 0;
}
```

In Python:

```
print "Hello world"
```

C wird (im Allgemeinen) kompiliert:

```
examples> ls
hello.c
examples> make hello
cc     hello.c   -o hello
examples> hello
Hello World
```

Ein paar Worte zur Kompilation: C-Compiler sind Programme wie alle anderen. Unter Unix heißen sie in der Regel cc oder gcc, unter Windows gibts häufig IDEs, also Programme, die den Compiler hinter einem Editor und Menüs verbergen. Mit cygwin oder ähnlichem (vgl. auch die

einschlägige Wiki-Seite[4]) lässt sich aber auch ein Windows-System so herrichten, dass die hier beschriebenen Beschwörungen funktionieren – und das ist ziemlich empfehlenswert.

Warnung
Kompilation
Maschinensprache
transparent
Bytecode
Mnemonics

Gesteuert werden die Compiler durch mehr oder minder viele Optionen. Da niemand all die Optionen im Kopf behalten kann und sie sich auch von Programm zu Programm ändern, wird man in der Regel ein Programm namens make benutzen, das ausrechnen kann, wie ein Compiler aufgerufen werden soll. Wir werden später mehr von make hören. Ohne weiteres weiß make jedenfalls, dass es, wenn wir `make hello` schreiben und es eine Datei `hello.c` im aktuellen Verzeichnis gibt, das Kommando `cc hello.c -o hello` ausführen muss. (probiert aus, was passiert, wenn ihr `make hello.o` tippt – die Erklärung folgt). Dieses Kommando sagt so viel wie: „Kompiliere die Datei hello.c und schreibe die Ausgabe (den Output, daher -o) in die Datei hello."

In der Tat ist das nicht ganz das, was wir wollen. C-Compiler können *Warnung*en ausgeben, die meistens darauf hinweisen, dass man irgendwas nicht so gemacht hat, wie man es hätte machen sollen. Der gcc, der unter Unix üblicherweise verwendet wird, muss dazu mit dem Flag `-Wall` aufgerufen werden. Die einfachste Art, dafür zu sorgen und trotzdem die Kompilation immer noch per make machen zu lassen, ist, eine Datei namens `Makefile` mit dem Inhalt `CFLAGS += -Wall` in das aktuelle Verzeichnis zu schreiben. Erklärungen folgen auch hier.

Kompilation ist die Übersetzung eines Programms in eine maschinennähere Sprache. Im Fall von C ist das im Allgemeinen die Maschinensprache selbst.

Die *Maschinensprache* sind dabei die Bits, die der Prozessor des Rechners wirklich verarbeiten kann. Python wird übrigens auch kompiliert, allerdings nicht in die Maschinensprache des Zielprozessors, sondern in eine Zwischensprache, die wiederum zur Laufzeit interpretiert wird. Dieser Prozess ist aber *transparent*, d.h. als ProgrammiererIn merkt man nicht viel davon. Die .pyc- oder .pyo-Dateien, die nach einem `import` in python erzeugt werden, enhalten diesen *Bytecode*.

Das folgende ist ein Ausschnitt aus der Ausgabe von `objdump -S hello` und stellt ganz links eine (recht bedeutungslose) Adresse, dann in Sedezimalnotation die Instruktionen an den Prozessor. Ganz rechts steht eine marginal menschenlesbare Repräsentation dieser Zahlen. Die Wörtchen wie push, mov, sub, and usf. entsprechen (fast) eins zu eins Teilen der Zahlen, die die CPU wirklich ausführt (die restlichen Teile dieser Zahlen werden durch die Operanden ausgemacht, wie das geht, steht auf der nächsten Folie). Weil diese Wörtchen letztlich nur Merkhilfen für die Zahlen sind, heißen sie auch *Mnemonics*.

Dies ist im Wesentlichen die Ausgabe des Compilers, in diesem Fall des gcc 3.3.1 auf einem Athlon XP.

```
08048654 <main>:
#include <stdio.h>

int main(void)
{
 8048654:       55                      push   %ebp
 8048655:       89 e5                   mov    %esp,%ebp
 8048657:       83 ec 08                sub    $0x8,%esp
 804865a:       83 e4 f0                and    $0xfffffff0,%esp
 804865d:       b8 00 00 00 00          mov    $0x0,%eax
 8048662:       29 c4                   sub    %eax,%esp
        printf("Hello World.\n");
 8048664:       c7 04 24 44 87 04 08    movl   $0x8048744,(%esp,1)
 804866b:       e8 08 ff ff ff          call   8048578 <_init+0x38>
        return 0;
 8048670:       b8 00 00 00 00          mov    $0x0,%eax
}
 8048675:       c9                      leave
 8048676:       c3                      ret
```

Wenn ihr dieses Experiment selbst macht, werdet ihr feststellen, dass der Compiler noch weit mehr Kram in das kleine Hello-Programm reingeschrieben hat. Das ist im Groben Code, der

[4] `http://wiki.cl.uni-heidelberg.de/moin/RettetWindows`

verwendet wird, damit die main-Funktion auch in einer Umgebung leben kann, in der sie sich wohl fühlt.

In Python kann man sich so etwas ähnliches durch das Modul `dis` (für „Disassembler“) ausgeben lassen. Das kann z.B. so aussehen:

```
>>> def main():
...     print "Hello World"
...
>>> dis.dis(main)
  2           0 LOAD_CONST         1 ('Hello World')
              3 PRINT_ITEM
              4 PRINT_NEWLINE
              5 LOAD_CONST         0 (None)
              8 RETURN_VALUE
```

Man ahnt schon, dass die einzelnen Instruktionen der durch den Bytecode definierten Python-Maschine mächtiger sind als die der realen Maschine, auch wenn das in diesem Trivialbeispiel noch nicht wirklich gut herauskommt.

Problems

(6.1)* Macht euch mit der Bedienung eures Compilers vertraut. Compiliert das Beispielprogramm. Andert den String im Quelltext und überzeugt euch, dass, wenn ihr euer Compilat laufen lasst, die Änderung natürlich nicht dort angekommen ist. Re-kompiliert den Quelltext und überzeugt euch, dass danach das Programm in der Tat die neue Ausgabe hat.

Tut euch den Gefallen und verwendet make.

7. Technik: Maschinensprache I

Bevor wir uns ansehen, was wir C eigentlich gesagt haben, werfen wir einen Blick auf den erzeugten Maschinencode. Dieser Kurs wird noch öfter einen Blick „nach unten“ werfen – das hilft zu verstehen, warum C so ist wie es ist.

Normale Mikroprozessoren haben sehr primitive Instruktionen. Sie können üblicherweise Dinge wie:

- Werte bewegen (Konstante nach Register, Speicher nach Register und umgekehrt, Register nach Register, manchmal auch Speicher zu Speicher): mov
- Rechnungen mit zwei Operanden: add, sub, mul div
- Bitweise Logik: and, or, not
- Werte vergleichen: cmp
- Springen, auch bedingt auf den Zustand der Flags: jmp, je, jne, jz, jc uvm.
- In „Unterprogramme“ springen und aus diesen zurückkehren: call/ret, int/iret
- Werte auf einen Stack legen und wieder von ihm runterholen: push, pop
- Nichts tun: nop

Beachtet: Hier steht nirgends etwas von Variablen (gut, man kann in Speicher schreiben), von Funktionen (die „Unterprogramme“ qualifizieren vielleicht, aber von Argumenten ist hier noch nicht die Rede) oder von Schleifen (dafür kann man springen).

Dafür haben wir hier schon einen Stack (vgl. Programmieren I, Folie XML II: SAX), d.h. eine Datenstruktur, die Daten so aufnimmt, dass die zuletzt reingeschobenen Daten zuerst wieder rauskommen (Last in, First out, LIFO).

Diese Instruktionen nehmen jeweils Operanden, und aus Instruktion plus Operanden lassen sich die Zahlen ausrechnen, die der Prozessor schließlich ausführen kann.

Für x86-Prozessoren ist das hoch kompliziert, bei anderen Prozessoren ist das viel einfacher. Der Umstand, dass ausgerechnet die idiotischste verfügbare Prozessorarchitektur sich am Markt

durchgesetzt hat, sollte im Hinblick auf den Wahrheitsgehalt diverser ökonomischer Theorien zu denken geben.

Wir verwenden hier die so genannte AT&T-Syntax, um die Maschinenbefehle in lesbare Form zu bringen. Außerhalb der GNU-Welt verbreiteter ist die so genannte Intel-Syntax, die sich in vieler Hinsicht von der AT&T-Syntax unterscheidet – am dramatischsten ist wahrscheinlich, dass Quell- und Zieloperanden bei Intel vertauscht sind, d.h. `mov a, b` bewegt in AT&T-Syntax von `a` nach `b` (wie „bewege a nach b“), in Intel-Syntax von `b` nach `a` (wie bei einer Zuweisung).

Wer sich näher mit der Programmierung mit Assemblern (das sind Programme, die die Mnemonics in den tatsächlichen Maschinencode umrechnen) auseinandersetzen wird, wird damit nicht lange Schwierigkeiten haben, zumal in der manpage des GNU as eine schöne Übersicht über die syntaktischen Differenzen gegeben wird.

Beispiele: `push %epb` (das heißt: Lege den Inhalt des Registers ebp auf den Stack) besteht aus dem Opcode für push (das ist im „alternate encoding“ $01010rrr_2$), worin rrr das Register spezifiziert, was für ebp 101_2 ist. Es ist 01010101_2 =0x55.

`sub $0x8,%esp` (das heißt: Ziehe den Wert 8 – man spricht bei solchen Literalen hier auch gern von „immediate value“, Werte also, die ohne weiteres Nachsehen bekannt sind und im Opcode kodiert werden können – vom Stackpointer ab) besteht aus dem Opcode für sub immediate to register, $1000\,00sw_2$: $1110\,1rrr_2$, wo rrr wieder das Register (100_2), s die sign extension (siehe unten) und w die Größe des Arguments gibt, und dem Argument. s ist hier 1, w ist 1, weil wir ein Byte abziehen (und nicht etwa ein Wort (16 bit) oder ein Doppelwort (32 bit)). Das Argument ist einfach ein Byte, 8. Zusammen haben wir: $1000\,0011_2$: $1110\,1100_2$: $0000\,1000_2$ oder kurz 83 ec 08 hexadezimal.

Was aber tut das Codefragment auf der letzten Folie? Gehen wir es durch:

```
08048654 <main>:
#include <stdio.h>

int main(void)
{
 8048654:       55                      push   %ebp
 8048655:       89 e5                   mov    %esp,%ebp
 8048657:       83 ec 08                sub    $0x8,%esp
 804865a:       83 e4 f0                and    $0xfffffff0,%esp
 804865d:       b8 00 00 00 00          mov    $0x0,%eax
 8048662:       29 c4                   sub    %eax,%esp
```

Wir sehen zunächst, dass die komische include-Zeile keinen Code produziert. Wir werden gleich sehen, warum das so ist.

Der folgende Funktionskopf erzeugt hingegen einen Haufen Code, der ausgesprochen wirr wirkt. In Wirklichkeit wird hier im Wesentlichen dafür gesorgt, dass Platz für lokale Variablen der Funktion da ist (vgl. die Folie „Parameterübergabe in C“). Jede C-Funktion auf x86-Maschinen muss nach Konvention (der so genannten calling convention) als erstes das Register ebp (Base Pointer, das e steht dafür, dass es 32 bit groß sein soll) auf dem Stack retten (push) und danach den Stackpointer in den ebp kopieren (mov). Das, und was danach kommt, sird auf der genannten Folie erklärt.

```
        printf("Hello World.\n");
 8048664:       c7 04 24 44 87 04 08    movl   $0x8048744,(%esp,1)
 804866b:       e8 08 ff ff ff          call   8048578 <_init+0x38>
```

Hier haben wir endlich Maschinencode für das, was wir geschrieben haben. Das movl (move long) legt die Adresse 0x8048744 auf den Stack. Eigentlich sollte hier ein push stehen, und es reicht, sich das so vorzustellen. Der gcc macht auf meiner Maschine die Dinge hier etwas konfuser, aber das Ergebnis seiner Operation unterscheidet sich nich von einem push, gefolgt von einem pop nach der folgenden Instruktion. Die Hoffnung ist, dass der Code so schneller ist.

Wirklich interessant an der movl-Instruktion ist aber, dass hier indirekt adressiert wird: Der Rechner nimmt die Adresse, an die das erste Argument kommen soll, *aus* dem Register esp (im

Gegensatz zu: Es schreibt das erste Argument nach esp) – das ist die Bedeutung der Klammern, und wir werden sehen, dass C mit dem *-Operator ein ziemlich enges Analogon dazu bietet.

Tatsächlich können CPUs (sogar die von Intel) noch viel raffinierter adressieren: Ein Ausdruck wie `16(%esp, 20, 4)` nimmt den Wert von esp, addiert 16 und dann nochmal $20 \cdot 4$ drauf. Warum man sowas haben möchte, werdet ihr sehen, wenn wir arrays und records behandeln.

An der erwähnten Adresse 0x8048744 sollte unser String `Hello World\n` liegen. Das wird er in der Realität natürlich nicht tun, weshalb das Betriebssystem diese Adresse auch anpassen wird, wenn es das Programm lädt und ausführt. Aber das ist noch ein anderes Thema.

Auch die im call erwähnte Adresse wird angepasst, so dass sie auf die Funktion `printf` zeigt, deren Maschinencode anderswo liegt. Dafür ist (nicht ganz genau, aber ungefähr) der Linker zuständig, von dem wir bald hören werden. Jedenfalls sorgt das call dafür, dass die Ausführung in dessen Code weitergeht, bis dort ein `ret` (wie return) auftaucht. Danach läuft das Programm weiter in unserem Code – wohin er zurückspringen muss, weiß der Prozessor, weil call die Adresse der nächsten Instruktion auf dem Stack hinterlassen hat, so dass ret nur noch diesen Wert in den Program Counter poppen muss.

```
       return 0;
 8048670:       b8 00 00 00 00          mov    $0x0,%eax
```

Return macht in C ziemlich das, was es in Python auch tut. Der Compiler macht aus unserem Ansinnen, Null zurückzugeben, einen move nach eax – und in der Tat sehen die calling conventions vor, dass eine Funktion ihren Rückgabewert in eax lässt (das bedeutet insbesondere, dass es nicht so einfach sein wird, aus C-Funktionen mehr als einen Wert zurückzugeben, weil es eben nur ein eax-Register gibt).

```
}
 8048675:       c9                      leave
 8048676:       c3                      ret
```

Die Instruktion `leave` hatte ich oben nicht erwähnt – sie macht das, was der Compiler oben aus unserem Funktionskopf gemacht hat, rückgängig (in der Tat hätte gcc oben auch die Maschineninstruktion `enter` erzeugen können, was er aber wohl deshalb nicht gemacht hat, weil auf Athlon XPs der Code, den er wirklich erzeugt hat, schneller ist als ein äquivalentes enter). Das `ret` hatten wir oben schon disktutiert, es sorgt hier dafür, dass unser Programm wieder in den Untiefen des eigenartigen Compiler-Codes verschwindet.

8. Anatomie unseres Programmes

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

(„Hash include“) Sagt dem Compiler, eine Datei `stdio.h` zu lesen. Im Fall von `stdio.h` („Standard I/O“) wird erklärt, welche Argumente die Funktionen zur Ein- und Ausgabe nehmen (die ist das Äquivalent der Python-Signaturen, die dem Compiler hier explizit bekannt sein müssen) und verschiedene Konstanten definiert. Hier für `printf` nötig.

Es sei hier ausdrücklich vor dem Irrtum gewarnt, dieses `include` sei irgendwie enger mit Pythons `import` verwandt. Include-Dateien enthalten anders als Python-Module in der Regel keinen (oder nur sehr wenig) Code. Sie sind allein dafür da, den Compiler mit verschiedenen Funktionen, Typen und anderen Kleinigkeiten bekannt zu machen. Definiert sind diese Funktionen woanders. Im Fall von stdio.h ist das die Standardbibliothek, die deshalb Standard ist, weil der C-Compiler „von selbst“ in ihr nachsieht, so dass wir hier keine weiteren Beschwörungen brauchen. Im Fall anderer Module (die in C in Bibliotheken organisiert sein können) ist das anders, ein `include` reicht nicht. Mehr dazu später, wenn wir selbst C-Module basteln. Schon jetzt aber der Hinweis, dass, auch wenn der include-Mechanismus dazu verführen mag, einfach kompletten Quelltext (mit Funktionsdefinitionen) einzubinden, so eine Vorgehensweise ausschließlich Ärger bringt und deshalb zu unterlassen ist.

```
int main(void)
```

Definiert eine *Funktion* `main`, die eine ganze Zahl zurückgibt (`int`) und kein Argument nimmt (`void`). C definiert, dass die Funktion main aufgerufen wird, wenn das Programm startet. Hier sehen wir zum ersten Mal, dass anders als in Python in C Variablen (und nicht Werte) Typen haben. Insbesondere gibt eine bestimmte Funktion immer nur Werte eines bestimmten Typs zurück, anders als in Python, wo eine Funktion etwa sowohl None als auch eine ganze Zahl zurückgeben kann.

Funktionskörper

```
{
```

In C werden Blöcke mit geschweiften Klammern abgeteilt, sie ersetzen sozusagen die Einrückung. Man sollte in C dennoch Blöcke einrücken, weil Menschen mehr Schwierigkeiten als Computer haben, aus den Klammern die Struktur des Programms zu rekonstruieren. Die korrekte Einrückung von C-Programmen ist Gegenstand heißer Debatten der Fachwelt.

Wer ein Linux-System mit installierten Kernelquellen hat, kann sich die Datei /usr/src/linux/Documentation/CodingStyle ansehen. Ich stimme Linus' Ansichten recht weitgehend zu.

```
printf("Hello World\n");
```

Die Funktion `printf` gibt ihre Argumente auf den Bildschirm aus, sie entspricht etwa

```
def printf(fmt, *args):
  print fmt%args
```

in Python. Der Strichpunkt schließt die Anweisung ab und hat in etwa in Funktion des Zeilenvorschubs in Python.

```
return 0;
```

Das kennen wir aus Python, neu ist wieder der Strichpunkt. Zeilenenden bedeuten für C fast nichts.

```
}
```

Diese geschweifte Klammer schließt den Block (nämlich den *Funktionskörper*).

9. Warum der Aufwand?

Offenbar sind C-Programme wortreicher als Python-Programme – in nichttrivialen Beispielen wird das deutlicher werden. Tatsächlich hat C noch erheblichere Nachteile gegenüber Python, insbesondere den, dass es sehr einfach ist, in C schlimme Fehler zu machen – die meisten Einbrüche in Rechnersysteme nutzen klassische C-Fehler aus.

Warum dann in C programmieren? Der offensichtlichste Grund: Geschwindigkeit. Das Programm

```
#include <stdio.h>

int main(void)
{
  int i, k;
  double res=0;

  for (i=1; i<10000; i++) {
    for (k=1; k<10000; k++) {
      res += i/(double)k;
    }
  }
  printf("%f\n", res);
  return 0;
}
```

braucht auf meinem Rechner 1.46 Sekunden.

Das äquivalente Python-Programm

```
res = 0.
```

```
for i in range(1, 10000):
  for k in range(1, 10000):
    res += i/float(k)
print res
```

läuft hingegen rund 160 Sekunden, über 100 Mal länger. Natürlich würde das so niemand schreiben. In der Tat verbringt Python lange Zeit mit dem Bauen um Abreißen der Listen von range, und schon der Übergang zu xrange bringt einen Geschwindigkeitsgewinn von etwa 30%. Der Übergang zu

```
  seq = range(1, 10000)
  multipliers = [1/float(k) for k in seq]
  res = 0.
  for i in seq:
    res += sum(
      itertools.starmap(operator.mul,
        itertools.izip(itertools.repeat(i), multipliers)))
  print res
```

– bei dem man sozusagen die Umwandlung in floats samt Division nur einmal am Anfang durchführt – reduziert die Ausführungsdauer auf rund 40 Sekunden. Das ist immer noch ein ganzes Stück von C weg, aber immerhin acht Mal schneller als die Originalversion (dieses Verhältnis würde noch wachsen, wenn man die Sequenzen länger machen würde). C überholen kann man allerdings nur mit Hirnschmalz. Man kann nämlich beobachten, dass wir letztlich $\sum_i \sum_k i/k$ ausrechnen und dass hier die Summationsreihenfolge vertauscht werden kann, $\sum_{k=1}^{n-1} \frac{1}{k} \sum_{i=1}^{n-1} i$. Die innere Summe kann man direkt auswerten und kommt auf $n(n-1)/2$. Das resultierende Programm,

```
  res = 0
  n = 10000
  iSum = n*(n-1)/2.
  for k in range(1, n):
    res += iSum/k
  print res
```

läuft in 0.023 Sekunden, also 50 Mal schneller als das C-Programm. Das ist durchaus typisch: Gute Algorithmen und eine saubere Problemanalyse bringen häufig genug mehr als trickreiche Optimierungen oder Implementationen in „schnellen“ Sprachen.

In der Regel liegt der Geschwindigkeitsunterschied von C zu Python für typische computerlinguistische Probleme bei etwa einem Faktor 10 oder 20 – wenn C und Python identische Algorithmen verwenden.

Gerade bei interaktiven Programmen ist auch das häufig nicht wichtig, da der Rechner ohnehin die meiste Zeit mit Warten auf den Benutzer verbringt. In anderen Fällen ist es egal, ob das Programm 0.02 oder eine Sekunde läuft, der Unterschied zwischen zwei Stunden oder zwei Tagen Entwicklungszeit ist aber durchaus relevant.

Dennoch: Gerade bei rechenintensiven Problemen lohnt der Übergang zu C oder einer anderen „mid-level“-Sprache.

Das gilt noch mehr, wenn man an der Hardware oder an einzelnen Bits herumfummelt.

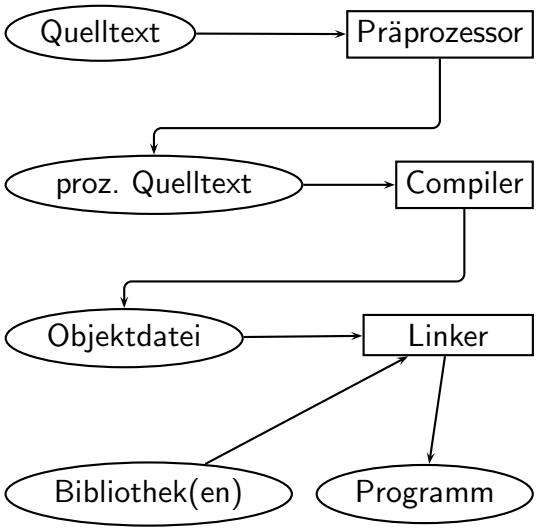


Fig. 2

10. Die Dreifaltigkeit des C-Compilers

Datenflussdiagramm der Kompilation

Bei einem `cc hello.c -o hello` passiert folgendes:

(cf. Fig. 2)

C besteht gewissermaßen aus drei Sprachen:

Präprozessor: z.B. `#include`, `#define`, allgemein alles mit einem Hashmark davor. Textersetzungen, bevor der eigentliche Compiler etwas sieht.

Python hat nichts Entsprechendes, auch wenn prinzipiell nichts dagegen spricht, auch Python-Programme durch einen Präprozessor irgendeiner Art zu schicken – es ist aber weder üblich noch direkt sinnvoll, da Python viele und weitaus ausgefeiltere Möglichkeiten bietet, Probleme zu lösen, für die C den Präprozessor braucht. Wie auch immer: Der C-Präprozessor erhält nicht unbedingt whitespace und ist deshalb für Python ungeeignet.

Sprachkern: Reservierte Wörter wie `int`, `void`, `return` (und etwa 40 weitere), die der Compiler direkt versteht, dazu noch Operatoren (`+`, `*`, `%` usf.) und diverse syntaktische Symbole (Anführungszeichen, Klammern, Strichpunkt usf.).

Die reservierten Wörter von Python sind etwas unterschiedlich von denen in C – die meisten Operatoren funktionieren aber in den beiden Sprachen ganz ähnlich, viele (arithmetische) Python-Ausdrücke lassen sich direkt in ein C-Programm kopieren. Die vordefinierten Typen (int, float, list, dict. . .) sind in Python keine reservierten Wörter (sie sind im builtin-Dictionary), in C schon.

Bibliotheksfunktionen: Im Beispiel wird nur printf benutzt. Für sie baut der Compiler nur eine Referenz in das Kompilat ein, die der Linker nachher auflöst. Zu unterscheiden sind Standardbibliothek und Zusatzbibliotheken.

Generell kommen alle Funktionen, die aufgerufen, aber in der Quelle nicht definiert werden, „von außen“. Vorläufig bedeutet das für uns „aus Bibliotheken“.

Die Bibliotheksfunktionen entsprechen etwa dem, was in Python per `import` geholt wird (plus den Builtin-Funktionen). Es gibt in C kein Äquivalent zu import, eine etwas ähnliche Funktionalität stellt der Linker zur Verfügung. Insbesondere gibt es in C keine Namespaces, das Dazulinken eine Bibliothek entspricht also quasi immer einem `from lib import *`.

Problems

statische Typisierung

dynamische Typisierung

Definition

(10.1)* Ihr könnt beim gcc sehr einfach den Präprozessor alleine laufen lassen, indem ihr die Option -E mit übergebt. Der Compiler gibt dann einfach das, was er eigentlich dem „wirklichen“ Compiler übergeben würde, auf die Standardausgabe aus. Probiert also

```
gcc -E hello.c | more
```

(oder etwas ähnliches) und seht nach, was der Compiler aus eurem Code macht. Verstehen müsst ihr das nicht unbedingt...

(10.2) Schreibt einen naiven Präprozessor für Python. Er soll #include können und sein Ergebnis zunächst auf die Standardausgabe ausgeben. Überlegt euch, was der Unterschied zwischen so einem include und einem import ist, und was der Unterschied zu einem from x import *. **(L)**

11. Variablen I

In C sind Variablen Namen für Speicherplätze, die zusätzlich Information darüber enthalten, wie die Bits in diesem Speicherplatz interpretiert werden sollen (den Typ).

Das ist ein entscheidender Unterschied zu Python. In Python ist die Variable a nur ein Schlüssel in einem Dictionary, das auf irgendwelche Werte verweisen kann – in der Regel wird sich bei jeder Zuweisung zu a die Speicheradresse ändern, auf die a verweist.

In C bedeutet die Zuweisung a=7 so viel wie „Schreibe das Bitmuster, das für den Typ von a die Zahl 7 kodiert, in die Speicherzelle, die du für a reserviert hast“, in Python heißt a=7 „Lasse a auf den Wert 7 vom Typ Integer verweisen“.

Es gibt auch in C einen Mechanismus, der ähnlich wie die Python-Referenzen funktioniert, nämlich die Pointer; zu denen kommen wir aber erst viel später.

Nochmal: In C gehört der Typ zur Variable (*statische Typisierung*), in Python zum Wert (*dynamische Typisierung*).

`int numberOfLines=0;` bedeutet: „Reserviere einen Speicherplatz unter dem Namen numberOfLines und schreibe Null rein. Die Bits stellen eine Ganzzahl dar.“ – *Definition* der Variable.

Gültige Namen in C

Gültige Namen bestehen aus:

- Buchstabe oder Unterstrich
- Buchstabe, Unterstrich oder Zahl

Oder in EBNF-Formulierung:

$$
\begin{aligned}
\mathit{VarName} &::= \mathit{VarOpen}\,\{\,\mathit{VarChar}\,\} \\
\mathit{VarChar} &::= \mathit{VarOpen} \mid \texttt{"0-9"} \\
\mathit{VarOpen} &::= \texttt{"A"} - \texttt{"Z"} \mid \texttt{"a"} - \texttt{"z"} \mid \texttt{"\_"}
\end{aligned}
$$

(cf. Fig. 3)

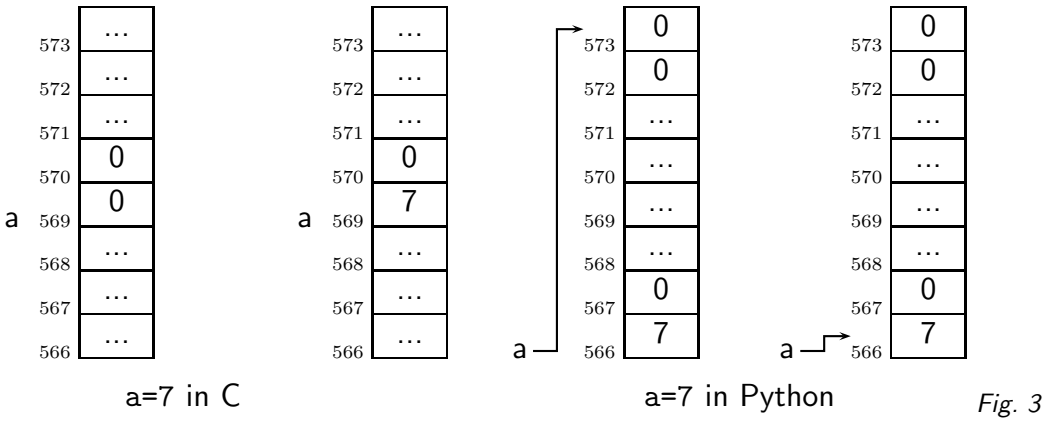


Fig. 3

Nichtterminalsymbol

Terminalsymbol

Exkurs: Kommentieren in C

Ein Kommentar sieht in C in der Regel so aus:

```
/* Some comment,
   may span several lines */
```

Ähnlich wie die Python-Kommentare (# macht den Rest der Zeile zum Kommentar) funktioniert das durch die C++-Sprachdefinition eingeführte `//`:

```
  a++; // C++-Style comment
```

Auch wenn sie mittlerweile wohl von allen noch verwendeten Compilern aktzeptiert werden und auch vom C99-Standard sanktioniert sind, haben C++-Kommentare meiner bescheidenen Meinung nach in C-Programmen nichts verloren; der wichtigste Grund dafür ist ästhetischer Natur: Der Sprachkern von C ignoriert das Codelayout (ihm ist egal, wo welche Sorte von whitespace ist, insbesondere sind Leerzeichen und Zeilenumbrüche äquivalent), und die C++-Kommentare verletzen diese Regel. Im Präprozessor übrigens ist das anders: Der legt erheblichen Wert auf Zeilenumbrüche an den richten Stellen (z.B. vor Präprozessorinstruktionen) und wird sehr irritiert, wenn Zeilenumbrüche an den falschen Stellen kommen (z.B. in einer Makrodefinition).

Will man große Codeteile auskommentieren (ähnlich wie in Python durch Verwandlung in einen langen String durch `"""` oder `'''`), macht man das am Besten mit dem Präprozessor:

```
#if 0
  commented-out stuff
#endif
```

Bei der Wahl der Inhalte der Kommentare taugen die Leitlinien für Python, auch wenn C natürlich keine Docstrings hat.

12. Exkurs: EBNF

EBNF (Extended Backus-Naur Form) ist Sprache zur Darstellung von kontextfreien Grammatiken. Jede EBNF-Klausel definiert eine Ersetzungsregel, die vorschreibt, wie ein *Nichtterminalsymbol* (ein Symbol, das nicht in Wörtern der erzeugten Sprache vorkommen darf) in andere Nichtterminalsymbole oder *Terminalsymbol*e (Symbole aus dem Alphabet der erzeugten Sprache) umgesetzt wird. Eine Regel sieht so aus:

$$\textit{ganzeZahl} \ ::= [\textit{vorzeichen}]\textit{ziffer}\,\{\textit{ziffer}\}$$

Eine ganze Zahl besteht danach aus einem optionalen Vorzeichen (die eckigen Klammern), einer Ziffer und null oder mehr weiteren Ziffern (die geschweiften Klammern). Die (kursiv geschriebenen) Nichtterminalsymbole *vorzeichen* und *ziffer* müssen noch erklärt werden. Das geht mit

$$\begin{aligned} \textit{ziffer} &::= \texttt{"0"} \mid \texttt{"1"} \mid \texttt{"2"} \mid \texttt{"3"} \mid \texttt{"4"} \mid \\ &\quad\ \texttt{"5"} \mid \texttt{"6"} \mid \texttt{"7"} \mid \texttt{"8"} \mid \texttt{"9"} \mid \\ \textit{vorzeichen} &::= \texttt{"+"} \mid \texttt{"-"} \end{aligned}$$

In `typewriter` gedruckt sind Terminalsymbole , der vertikale Strich steht für „oder“.

Oft sieht man auch (schlampig, aber bequem):

$$\textit{ziffer} ::= \texttt{"0"} - \texttt{"9"}$$

Letztlich ist EBNF eine kompakte Darstellung der bei formalen Sprachen gewohnten Regelmengen. Die Regel für *ganzeZahl* z.B. ließe sich auch schreiben als:

$$\begin{aligned} G &\to Z \\ G &\to VZ \\ Z &\to NZ \\ Z &\to N, \end{aligned}$$

wo G die Ganze Zahl, Z eine Ziffernfolge, V ein Vorzeichen und N eine Ziffer ist.

Vorsicht: Es gibt so viele Varianten von BNF und EBNF wie es AutorInnen gibt, die etwas damit Ausdrücken. Häufige Varianten:

- Variation von „::=“
- Nichtterminale in spitzen Klammern
- Terminale ohne Anführungszeichen

13. Variablen II

Definitionen von Variablen

Mögliche Typen von Variablen sind

- `int` – eine Ganzzahl: -1, 1, 3, 32890
- `float` und `double` – eine „reelle“ Zahl: 3.1425, 3e10
- `char` – ein Zeichen: a, +

C hat nicht wirklich einen eingebauten String-Typ – ein char ist genau ein Zeichen. Wir werden später sehen, wie so etwas ähnliches wie Stings in C manipuliert werden können.

Die Typen können (teilweise) modifiziert werden durch

- `unsigned` – Vorzeichenlose `int` und `char` können keine negativen, aber dafür größere positive Zahlen darstellen
- `long` – für `int` (und `double`) wird der Wertebereich erweitert
- `short` – für `int` wird der Wertebereich eingeschränkt und dafür weniger Speicherplatz benötigt

Übrigens hat auch Python longs, diese sind dort aber Ganzzahlen beliebiger Länge (und auch entsprechend langsam). Also: C-longs sind Python-ints, Python-longs gibts in C so erstmal nicht.

Definitionen und Literale

```
int zahl;
int eins=1;
int dez32=0x20; /* Hex-Zahlen haben 0x vorne */
int dez16=020;  /* Oktal-Zahlen haben Null vorne --
  das ist heute nur noch als Falle zu verstehen */
long l=3204918L;/* Bei großen Zahlen kommt eine
  Warnung ohne das L */
unsigned long l2=0xdeadbeefUL;
double fl1=3.45;
double fl2=3.45e-10;
char initial='M';
char buchstabe=68;
char str[]="Ein String\n";
char a, b, c, d;
```

Man kann in C also bei der Definition von Variablen gleich einen Wert mitgeben – oder es auch lassen. In der letzten Zeile sieht man, dass man mehrere Variablen gleichen Typs in einem Statement definieren kann.

Oktalzahlen sind Zahlen, die zur Basis 8 geschrieben sind. Die Schreibweise für Literale ist in C und Python weitgehend identisch. Die Hauptfalle sind die Anführungszeichen. In C sind die einfachen Anführungszeichen für char-Literale reserviert (also einzelne Zeichen), die doppelten für Strings. Die eckigen Klammern hinter `str` werden ebenfalls erst später behandelt.

Man beachte, dass es kein Problem ist, einem char eine Zahl zuzuweisen. C wandelt viele Werte quasi von selbst in die Typen, die die Variablen haben möchten. Leider kann das auch ins Auge gehen, was Grund etlicher klassischer C-Fehler ist. Gute Compiler geben Warnungen aus, wenn mal Sachen wie `char c=332;` macht – und char nicht genügend bits zur Darstellung von 332 hat, was heute aber der Normalfall ist und sich aus vielen Gründen auch nicht ändern dürfte.

Üblich ist, dass ein char 8 bit hat (256 Werte), ein short int 16 bit (Wertebereich von -32768 bis 32767) und ein (long) int 32 bit (ca. -2 Milliarden bis +2 Milliarden – das sind die normalen python-ints). doubles gehen von etwa -10^{308} bis 10^{308} (das sind die python-floats), floats von etwa -10^{34} bis 10^{34}. Der ANSI-Standard schreibt das aber nicht genau vor, und insbesondere ist mit der Verbreitung von 64-bit-Arechitekturen damit zu rechnen, dass ints bald 64 bit breit sein werden (also Zahlen zwischen -2 und 2 Trillionen darstellen können).

Die unsigned-Varianten der ganzen Zahlen gehen von 0 bis $2^{\text{Länge in bits}}$. Üblich ist also, dass ein unsigned int Werte von 0 bis gut 4 Milliarden speichern kann.

Character-Literale der Art `'c'` zählen als ganze Zahlen. Auf üblichen Maschinen gibt es keinen Unterschied zwischen `'A'` und 65. Leider überlässt der C-Standard es dem Compiler, ein unspezifiziertes `char` signed oder unsigned zu sehen.

Auf üblichen Maschinen kann `'Ä'` demnach -60 oder 196 sein.

14. Technik: Zahlendarstellung im Computer

MSB
Signed Magnitude
Einerkomplement
Zweierkomplement
Carry

Vorzeichen

Ganze Zahlen können wir einfach in Bits speichern. Was machen wir mit dem Vorzeichen? Drei Methoden wurden versucht, alle „reservieren“ das *MSB* (most significant bit, das höchste bit) – sie unterscheiden sich in der Interpretation der anderen Bits.

Signed Magnitude: Negation einer Zahl durch Flippen des MSB. Damit sind $1000\,0000_2$ und $0000\,0000_2$ Null. Arithmetik schwer zu bauen.

Einerkomplement: Negation einer Zahl durch Flippen *aller Bits*, -51 wäre dann $1100\,0011_2$, weil $0011\,1100_2$ 51 ist. Auch hier ist Arithmetik schwierig und sowohl 0x00 als auch 0xff sind 0.

Zweierkomplement: Negation einer Zahl durch Berechnung des Einerkomplements und addieren von 1. Jetzt gibt es nur eine Null, deren Negation $1111\,1111_2 + 1 = 1\,0000\,0000$ ist. Das neunte Bit speichern wir nicht, wir sind also wieder bei der Null. Implementation in Hardware ist einfach.

Die Einfachheit der Implementation ergibt sich vor allem aus dem Umstand, dass etwa bei einer Addition keine Rücksicht auf das Vorzeichenbit genommen werden muss – im schlimmsten Fall ergibt sich ein „Überlauf“, weil das Ergebnis nicht mehr mit der gegebenen Zahl von Bits darstellbar ist. In diesem Fall funktioniert die Zweierkomplementdarstellung aber immerhin so, dass durch Addition oder Subtraktion der größten bzw. kleinsten darstellbaren Zahl das richtige Ergebnis herauskommt.

Ein Beispiel: $33 - 97 = -64$. Ihr könnt leicht ausrechnen, dass

$$33 = 0010\,0001_2$$
$$97 = 0110\,0001_2$$

ist. Die Subtraktion entspricht der Addition der negativen Zahl, man muss also das Zweierkomplement von 97 ausrechnen. Erst flippt man alle Bits – $1001\,1110_2$ – und addiert dann $0000\,0001_2$, was auf $-97 = 1001\,1111_2$ führt.

Um nun die Zahlen zu addieren, geht man wie in der Grundschule vor: Man schreibt die beiden Zahlen untereinander und addiert die Stellen von links nach rechts. Wenn man $1 + 1$ rechnen muss, kommt 0 heraus und man hat einen Übertrag von 1 auf die nächste Stelle, muss man $1+1$ plus einen Übertrag berechnen, bekommt man 1 und einen Übertrag.

Das Ganze kann dann so aussehen:

$$\begin{array}{r} 0010\,0001_2 \\ \underline{1\underline{001}\,\underline{111}1_2} \\ 1100\,0000_2 \end{array}$$

Dabei habe ich Stellen mit Übertrag durch einen Unterstrich markiert. Das Ergebnis ist negativ (das MSB ist gesetzt). Um herauszubekommen, was es ist, ziehen wir eins ab. Das entspricht der Addition von $-1 = 1111\,1111_2$, was auf $1011\,1111_2$ führt, wobei wir den hier eigentlich anfallenden Übertrag auf der höchsten Stelle – das ist der schon erwähnte *Carry* – ignorieren. Jetzt flippen wir die Bits und kommen auf $0100\,0000_2$ oder einfach 64. Das Ergebnis unserer Rechnung war also wirklich -64.

Natürlich funktioniert das auch mit 16- oder 32-bittigen Zahlen ganz analog. Das Zweierkomplement hat leider die Falle, dass es nicht gleich viele positive wie negative Zahlen gibt. Die größte darstellbare positive Zahl (wieder für ein Byte) ist $0111\,1111_2$, nämlich 127. Das Zweierkomplement davon ist $1000\,0001_2$, nämlich -127 in Zweierkomplementdarstellung. Davon kann man offenbar noch eins abziehen, um auf $1000\,0000_2$ zu kommen, -128 in Zweierkomplementdarstellung. Das Zweierkomplement davon ist wiederum $0111\,1111_2 + 1$, mithin $1000\,0000_2$, also unser bekanntes -128. So etwas kann einen beißen (es bedeutet, dass etwas so harmloses wie der Ausdruck -a für bestimmte a „nicht funktionieren“ kann). Beim Umgang mit negativen Zahlen am Rande des vom jeweiligen Typ darstellbaren Bereichs ist das zu bedenken.

Anmerkung: Python hat solche Probleme natürlich nicht, weil Bitmuster und Interpretation *im Wert* zusammenhängen. Früher konnte man zu große Integer einfach nicht konstruieren, mittlerweile schaltet Python bei einem Überlauf der Integer automatisch auf Long um. Das heißt nicht, dass es hier keine Probleme mit dem Zweierkomplement gäbe – sie tauchen z.B. im Zusammenhang mit großen Integer-Literalen auf. Wer will, kann sich das in PEP 237[5] ansehen, sucht nach oct literals.

Mantisse

Exponent

Sign Extension

Toll an der Zweierkomplementdarstellung ist auch, dass, wenn die Zahlen klein genug sind, man einfach „oben“ Bits wegstreichen kann und die Zahlen gleich bleiben, egal, ob die Bits 0 oder 1 sind. So ist -1 in 16 bit 0xffff, in 8 bit 0xff.

Das scheitert natürlich dann spektakulär, wenn die Zahl betragsmäßig zu groß ist – -180 kann natürlich nicht in 8 bit-Zweierkomplement dargestellt werden. Es ist nicht schwer, zu sehen, wann das nicht mehr klappt: Dann nämlich, wenn nicht alle bits über dem MSB der kleineren Darstellung den Wert des MSB der größeren haben.

Das macht den umgekehrten Weg etwas schwierig: Will man mehr Bits haben, muss man „oben“ Kopien des Sign Bits anhängen. Aus 0xa0 (das ist -0x60, also -96 – rechnet es nach) muss 0xffa0 werden, aus 0x60 hingegen 0x0060.

Der C-Compiler macht das richtig; so genannte Sign Extension Bugs schleichen sich allerdings bei unüberlegter Kombination von signeds und unsigneds ein.

Fließkommazahlen

Fließkommazahlen bestehen aus *Mantisse* und *Exponent*. In einer Zahl wie 1.23456×10^4 ist die Mantisse dabei 123456 und der Exponent 4. Dabei stellt der Rechner die Zahlen natürlich wieder binär dar.

Die üblichen IEEE-floats machen Signed Magnitude. Von 32 bits ist einer das Vorzeichen, 8 der Exponent plus 127 und 23 die Mantisse, die nach dem ersten gesetzten Bit gespeichert wird.

Das Ganze gibt es auch noch für doubles, die in 64 bits mit 11 bits für den Exponenten (plus 1023) und 52 bits für die Mantisse gespeichert werden. Wenn ihr das abschätzt, ergeben sich in etwa die Wertebereiche und Genauigkeiten, die auf der letzten Folie angegeben sind – etwas mehr als 3 Binärstellen entsprechen einer Dezimalstelle (weil 3 bits gerade 8 Ziffern darstellen können), d.h. 22 bits sind weniger als $22/3 \approx 7$ Stellen, die 51 bits entsprechend weniger als 17 Stellen.

Was den Exponenten angeht, kann man entweder 256 oder 2048 verschiedene Werte speichern. Allerdings sind das natürlich Exponenten für *binäre* Mantissen. Weil eine Binärstelle wieder knapp einer Drittel Dezimalstelle entspricht, haben wir dezimal einen Wertebereich von ungefähr 85 oder 680 – wenn man das halbiert, ist man ungefähr bei den -38 bis 38 und -308 bis 308 (dass es nicht genau stimmt, liegt wieder daran, dass man etwas mehr als drei Bits für ein Dezimalzeichen braucht).

Wer will, kann sich das mit Hilfe von Quanfei Wens IEEE-754 Calculator[6] genauer ansehen.

Übrigens werden in Anwendungen, in denen man auf Genauigkeit verzichten kann, aber dafür eine exakte Repräsentation von Dezimalzahlen braucht (das ist im Groben vor allem der Finanzsektor) auch gerne andere Darstellungen reeller Zahlen verwendet. Am bekanntesten sind wohl die Binary Coded Decimals (BCDs), in der jeweils vier Bit verwendet werden, um eine Dezimalziffer zu speichern.

[5] `http://www.python.org/peps/pep-0237.html`

[6] `http://babbage.cs.qc.edu/courses/cs341/IEEE-754.html`

Problems

(14.1)* C prüft (in der Regel) nicht, ob eine Zahl „überläuft“, d.h. der Wert einer Variable größer wird als der (betragsmäßig) größte in ihr speicherbare Wert. Probiert das mit folgendem Programm aus:

```
#include <stdio.h>

int main(void)
{
  signed char i=0;
  int ct=0;

  while (1) {
    printf("%d -- ", i++);
    if (ct++>300) {
      break;
    }
  }
  return 0;
}
```

Erklärt die Ausgabe des Programms. Ersetzt den signed char durch einen unsigned char, beobachtet, was passiert und erklärt auch das. **(L)**

(14.2)* S. F. Xaver möchte will 2*128 ausrechnen, ist aber ein Freund undurchschaubaren Codes und hat deshalb für diese Rechnung folgenden Code geschrieben:

```
#include <stdio.h>

int main(void)
{
  signed char i=-128;
  int whatever=128;

  i = -i;
  whatever += i;
  printf("%d\n", whatever);
  return 0;
}
```

Zu seiner Überraschung wird aber mitnichten 256 ausgegeben, sondern 0 (probiert es). Warum? **(L)**

(14.3)* Rechnet im 8-bit-Zweierkomplement $127 + 1$ aus. Rechnet umgekehrt $-128 + (-1)$ aus.

Rechnet die binäre Repräsentation von 99 und 95 aus. Addiert die Zahlen dann als 8 bit breite Zahlen und interpretiert das Ergebnis im Zweierkomplement. Könnt ihr die Systematik erkennen? **(L)**

(14.4) Beobachtet die Ausgabe des folgenden Programms und erklärt, was passiert (der u-Formatcode in printf sagt, dass das dafür übergebene Bitmuster als unsigned int zu interpretieren ist).

```
#include <stdio.h>

int main(void)
{
  unsigned int u;
  int i;
  signed char c=-10;

  u = c;
  i = c;
  printf("%u %d\n", u, i);
  u = (unsigned char)c;
  i = (unsigned char)c;
  printf("%u %d\n", u, i);
```

```
  return 0;
}
```
(L)

15. Operatoren in C

Wie in Python können Variablen und Literale mit Operatoren zu Ausdrücken kombiniert werden. Die Grundrechenarten laufen exakt analog zu Python, bis hin zur Integer-Division: 1/2 oder 4/5 ist Null, aber 1/2. ist 0.5. Grundsätzlich wird bei Operationen zwischen verschiedenen Typen immer auf den „größten“ Typen interpoliert.

Das heißt, dass der Typ des Ergebnisses eines Ausdrucks dem größten Typen seiner Operanden entspricht und die anderen Operanden bei Bedarf ebenfalls auf diesen Typen gebracht werden. Dabei ist short größer als char, int größer als short, long größer als int, float größer als long, double größer als float. Ähnliches gilt übrigens für Python, wo die Sequenz int, long, float, complex ist. Diese Interpolierei ist häufig ein wenig magisch, weshalb strenger typisierte Sprachen als C solche Dingen nicht machen – sie ist allerdings so bequem, dass sie so bald nicht verschwinden wird.

Beispiel: Sei `int i=1, j=2; double r=2.;`.
Dann gilt `i+j==3`, `i-j==-1`, `j*r==4.`, `i/j==0`, `i/r==0.5`.

Technik: Bitweise Operatoren

C bietet bitweise Operatoren, die auf die binären Darstellungen ihrer Argumente operieren (das soll heißen, dass man die binären Darstellungen der beiden Operanden hinschreibt und dann bit für bit eine logische Operation ausführt):

- `^` – bitweises exklusiv-oder
- `&` – bitweises und
- `|` – bitweises oder
- `~` – bitweises nicht
- `<<, >>` – bitweises shift left und right

Dabei heißt exklusiv-oder, dass im Ergebnis eine 1 steht, wenn *entweder* im einen *oder* im anderen Operanden eine 1 steht, sonst 0.

Bei „und“ steht im Ergebnis 1, wenn in beiden Operanden eine 1 ist, bei „oder“, wenn in mindestens einem eine 1 ist.

„nicht“ macht aus jeder 1 eine 0 und umgekehrt – „nicht“ ist natürlich kein zweistelliger Operator mehr, es hat nur einen Operanden.

Die Schiebeoperationen schieben einfach alle bits nach rechts oder links. Das Schieben von vorzeichenbehafteten Zahlen sollte man in der Regel unterlassen, es sei denn, man versteht den Abschnitt 6.5.7 des ISO-C-Standards und weiß, warum die dort erwähnten undefinierten Werte undefiniert geblieben sind – Details gibt es auf Anfrage.

Beispiele:

Sei `a=0xF2`$=1111\,0010_2$ und `b=0x2E`$=0010\,1110_2$. Dann ist

$$\texttt{a\&b} = 0010\,0010_2 = \texttt{0x22} \quad \texttt{a|b} = 1111\,1110_2 = \texttt{0xFE}$$
$$\texttt{a>>1} = 0111\,1001_2 = \texttt{0x79} \quad \texttt{\~{}a} = 0000\,1101_2 = \texttt{0x0D}$$
$$\texttt{a<<1} = 1\,1110\,0100_2 = \texttt{0x1E4} \quad \texttt{a\^{}b} = 1101\,1100_2 = \texttt{0xDC}$$

Diese bitweisen Operatoren existieren genau so in Python, werden dort aber nicht ganz so häufig verwendet.

Wozu ist das alles überhaupt gut? Grundsätzlich immer, wenn Bits manipuliert werden müssen, etwa bei der Grafikverarbeitung.

Ein anderer Anwendungsfall sind Mengen. In Python lassen sich Mengen recht einfach durch Dictionaries darstellen, C hat aber keine Dictionaries. Wenn die Mengen klein sind (sagen wir, nur 32 verschiedene Elemente haben können, was etwa bei nicht-ausschließenden Optionen von Programmen der Fall sein mag), lassen sie sich durch die bits in einem long darstellen.

In Python geschrieben könnte das etwa so aussehen, wenn wir zunächst damit zufrieden sind, wirklich nur die Zahlen zwischen 0 und 31 in der Menge zu haben:

```
class Set:

  def __init__(self, initMask=0):
    self.bitMask = initMask

  def __str__(self):
    return "{%s}"%(", ".join([str(i)
      for i in range(32) if (1<<i)&self.bitMask]))

  def addElement(self, elInd):
    self.bitMask |= 1<<elInd

  def delElement(self, elInd):
    self.bitMask &= ~(1<<elInd)

  def union(self, other):
    return Set(self.bitMask|other.bitMask)

  def section(self, other):
    return Set(self.bitMask&other.bitMask)
```

Wer will, kann die Klasse ja verbessern, z.B. um Fehlerprüfungen (die Elementnummern sollen ja beschränkt sein) oder auf eine einstellbare Anzahl von Elementen durch Verwendung einer geeigneten Liste von ganzen Zahlen. Mit Python-longs gehts natürlich auch ohne solche Tricks. Wer aber richtige Mengen in Python braucht, sollte wie gesagt entweder mit Dictionaries oder, ab Python 2.3, mit dem sets-Modul arbeiten.

Logische Ausdrücke

Logische Ausdrücke funktionieren ebenfalls wie in Python: Wahr ist alles, was verschieden von Null ist, falsch, was gleich Null ist.

Vergleiche sehen wie in Python aus: `<`, `>`, `<=`, `>=`, `==`, `!=`

Die logischen Operatoren `and`, `or` und `not` von Python sind in C symbolisch geschrieben: `&&`, `||` und `!`.

Vorsicht: Die logischen und die bitweisen Operatoren machen *ganz* verschiedene Dinge, auch wenn sie zufällig mal übereinstimmen.

Vergleicht *jetzt* bitweise und logische Operatoren und spart euch später viel Pein und Verdruss.

Übrigens werden in C wie in Python logische Ausdrücke mit short circuit evalutation bewertet, d.h. nur so lange berechnet, bis das Ergebnis des Ausdrucks feststeht.

Vorsicht: `if (a=0)` ist in C legal, aber etwas ganz anderes als `if (a==0)`.

Diese Konsequenz aus dem Umstand, dass die Zuweisung in C ein Operator ist (siehe unten – gute Compiler sagen allerdings etwas wie „Suggest parenthesis...“ oder so, wenn die Warnings eingeschaltet sind), ist eine endlose Quelle von ärgerlichen Programmierfehlern. Eine Möglichkeit, solche Fehler etwas zu reduzieren ist, sich anzugewöhnen, beim Vergleich gegen eine Konstante die Konstante immer zuerst zu schreiben: `if (0=a)` gibt einen Syntaxfehler, während `if (0==a)` natürlich völlig legal ist.

Die Warnung aus dem Python-Kurs bezüglich des Vergleichs von Fließkommazahlen gilt natürlich auch in C (tatsächlich übernimmt Python die Fließkommaimplementation vom C-System, mit dem es kompiliert wurde). In C sollte der Vergleich so aussehen: `fabs(a-b)<1e-10` – die Schranke muss dabei natürlich auf das Problem angepasst werden. Die Funktion `fabs` ist im Headerfile `math.h` definiert. cast

Problems

(15.1)* Sei `a=0x3a` und `b=0x86`, beide unsigned. Rechnet `~a`, `!a`, `a& b`, `a&&b`, `a|b`, `a||b` und `a<<1` sowie `a>>1` aus. **(L)**

(15.2) Konstruiert einen Fall, in dem `a&b` logisch falsch, `a&&b` aber logisch wahr ergibt. Wie typisch ist das? Wie ist die Situation, wenn ihr „oder“ statt „und“ betrachtet? **(L)**

16. Operatoren und Ausdrücke

Ausdrücke sind in C weitgehend wie in Python definiert. Eine wichtige Abweichung ist die Zuweisung – sie ist in Python ein Statement, in C ein Operator. Damit kann man Sachen wie

```
b = (c=4)+3;
c += (b*=2);
```

schreiben, wonach b 14 ist und c 18.

`b*=2` ist dabei eine Kurzschreibweise für `b = 2*b`. Das geht auch für die anderen Operatoren. Auch Python hat diese inkrementierten Zuweisungen.

Casts

C weigert sich in der Regel, Variablen verschiedenen Typs einander zuzuweisen. Wenn das doch sinnvoll ist, muss man in der Regel den *cast*-Operator anwenden. Ein Cast ist wörtlich übersetzt eine Gussform oder ein Gips, „zwingt“ also einen Typ in einen anderen. Dazu wird einfach der erwünschte Typ in Klammern vor den Ausdruck geschrieben: `(int)(1/2.)` oder `1/(double)2`.

Im Augenblick ist für uns das casten zwischen `int` und `double` noch das Interessanteste: Das Programm

```
#include <stdio.h>

int main(void)
{
  float a=2.7182818;

  printf("%f %d\n", a, (int)a);
  return 0;
}
```

gibt `2.718282 2` aus – es werden also einfach die Nachkommastellen abgeschnitten (die übliche Rundung kann für positive Zahlen etwa mit `(int)(a+0.5)` erreicht werden).

Bei einer Zuweisung macht C das Abschneiden der Nachkommastelle übrigens von selbst. Trotzdem ist es gut, auch bei Umwandlungen, die C automatisch macht, Casts zu schreiben, insbesondere, um deutlich zu machen, dass der/die ProgrammiererIn sich bewusst ist, dass hier Typumwandlungen stattfinden.

Wirklich wichtig wird der Cast erst im Zusammenhang mit Pointern.

Casts scheinen etwas Ähnliches wie die „Umwandlungsfunktionen“ in Python (`int`, `str`, `list`, ...) zu tun.

Diese Ähnlichkeit ist aber nur oberflächlich. Der wichtigste Unterschied ist, dass die Konstruktoren von Pythons Datentypen (nichts anderes sind ja diese Umkwandlungsfunktionen) tatsächlich immer neue Werte erzeugen, während Casts eigentlich nur Vorschriften zur nicht dem Typ der Variable entsprechenden Interpretation ihrer Bits sind.

Prä- und Postinkrement

Den ++-Operator (analog für --) erhöht seinen Operanden um eins, und zwar, bevor (++a) oder nachdem (a++) der Ausdruck bewertet wird. Beispiel:

```
a=4;
printf("%d\n", ++a);
printf("%d\n", a++);
printf("%d\n", a);
```

gibt 5, 5 und 6 aus.

Hintergrund dieser Operatoren ist nicht nur, dass man damit toll kompakten Code schreiben kann, sondern auch, dass die meisten Prozessoren Instruktionen wie inc und dec haben, die sehr schnell inkrementieren und dekrementieren können – Compiler haben es leichter, diese Instruktionen zu verwenden, wenn sie von vorneherein wissen, dass das Programm sie haben möchte.

Heute ist das relativ irrelevant – einerseits ist es für praktisch alle heute üblichen CPUs egal, ob man eins oder eine andere Zahl addiert, sie brauchen immer die gleiche Zeit dafür, andererseits sind die Compiler auch schlau genug, ein add durch ein inc zu ersetzen, wenn das vorteilhaft ist.

Präzedenz

Wie schon in Python, so lassen sich auch die C-Operatoren der Präzedenz nach ordnen. In der folgenden Tabelle „binden“ die Operatoren, die weiter oben stehen, stärker als die weiter unten stehenden. + steht beispielsweise unter dem binären *, also ist `4+5*6` äquivalent zu `4+(5*6)`, während >> noch unter + steht, `1+1>>4` also `(1+1)>>4` entspricht. Letzteres ist bereits eine der Fallen in dieser Präzedenztabelle, da die meisten Menschen die Shiftoperatoren eher als eine Art Multiplikation im Kopf haben und die „Punkt vor Strich“-Regel hier gerade zum falschen Ergebnis führt.

Assoziativität heißt, ob C Ausdrücke mit den entsprechenden Operatoren von links nach rechts oder umgekehrt liest. `a+b-c` wird beispielsweise linksassoziativ, also als `(a+b)-c` gelesen, während `a=b=c` rechtsassoziativ ist, also `a=(b=c)` entspricht. Wer von solchen Feinheiten abhängt, möge in der Hölle schmoren. Eine goldene Regel des C-Programmierens ist, nicht allzu oft in diese Tabelle zu gucken und lieber rechtzeitig Klammen zu setzen.

Nicht alle hier erwähnten Operatoren haben wir schon gesehen.

(cf. Fig. 4)

Problems

(16.1)* Betrachtet das folgende Programm:

```
#include <stdio.h>

int main(void)
{
  int a=8;

  printf("%d\n", a&8==(1<<3));
  return 0;
}
```

Versucht, vorherzusagen, was das Programm tut, probiert es dann aus. Kompiliert es mit dem -Wall-Flag (oder dem Äquivalent eures Compilers). Könnt ihr die (hoffentlich) erscheinende Warnung erklären? Was gibt das Programm wirklich aus und warum? **(L)**

(16.2)* Klammert die folgenden C-Ausdrücke nach den Präzedenzregeln vollständig (d.h. alle Zugehörigkeiten sind explizit durch Klammern markiert).

Beispiel: `5+b%8` wird zu `(5+(b%8))`

1. `1<<3+1`
2. `a = b = c`

Operatoren	**Assoziativität**
`() [] . ->`	links nach rechts
`! ~ - + ++ -- * & (type-cast) sizeof`	rechts nach links
(in der Vorzeile sind unäre +, -, & und * gemeint)	
`* / %`	links nach rechts
`+ -`	links nach rechts
`<< >>`	links nach rechts
`< <= > >=`	links nach rechts
`== !=`	links nach rechts
`&`	links nach rechts
`^`	links nach rechts
`|`	links nach rechts
`&&`	links nach rechts
`||`	links nach rechts
`?:`	links nach rechts
`= += -= *= /= %= &= |= <<= >>=`	rechts nach links
`,`	links nach rechts

Fig. 4

3. `a < b < c`
4. `a = b==a*b/5`
5. `a=15 || a&8==1<<3 && b<=5`

(L)

(16.3)* Ich habe oben behauptet, dass sich C in der Regel weigert, Variblen verschiedenen Typs zuzuweisen. Wenn wir also einen Typ string und einen Typ int haben, dann wäre das folgende Programm inkorrekt:

```
string a="bla";
int b=5;
b = a;
```

(nehmt vorläufig einfach an, es gebe einen Typ string). Demgegenüber hindert einen natürlich niemand daran, in Python

```
a = "bla"
b = 5
b = a
```

zu schreiben.

Erklärt, was diese Differenz mit den unterschiedlichen Variablenkonzepten von C und Python zu tun hat. **(L)**

17. Einfache Ein-/Ausgabe

`printf(format, arg1, ...)` gibt die Argumente gemäß dem Formatstring auf die Standardausgabe (das ist ohne weitere Vorkehrungen der Bildschirm) aus. Die Formatstrings sehen weitgehend wie in die ersten Operanden für den String-Prozentoperator in Python aus. Für die zusätzlichen Typen von C gibt es einige Formatcodes mehr:

- `%u` Unsigned integer wird dezimal ausgegeben
- `%c` Ein `char` (oder sonstiger integer) wird als Zeichen ausgegeben
- `%l`. Bei Ausgabe von `long ints` muss ein l-Modifier vor dem Formatcode stehen

Erschöpfend Auskunft über weitere Möglichkeiten gibt `man 3 printf` oder das glibc-Manual.

Ein paar Beispiele:
```
char user[]="msdemlei";
char format[]="%d-%d=%d";
int pagecount=5;
printf("%d --- ", pagecount);
printf("%10s hat %4d Seiten gedruckt\n", user, pagecount);
printf("Wahr ist: 129=%d%d", 1, 29);
printf(format, 5, 2, 5-2);
```

`printf` weiß nichts über die Typen seiner Argumente – ob das floats, chars, oder longs sind, muss allein aus dem Formatstring hervorgehen (gcc -Wall warnt allerdings, wenn man da groben Unfug versucht). Die Regeln zur Promotion von Argumenten (zu denen wir noch kommen werden), verhindern zwar häufig echte Katastrophen, dennoch werden Werte je nach Formatcode ganz verschieden ausgegeben. Das Programm
```
#include <stdio.h>

int main(void)
{
  int i=0x400+'d';
  unsigned long l=(1<<31)+'c';

  printf("%d, %c, %ld, %lu\n", i, i, l, l);
  return 0;
}
```

gibt z.B.
```
1124, d, -2147483549, 2147483747
```

aus.

Wilde Dinge passieren, wenn mehr Formatcodes als Argumente vorhanden sind – das kann bis zum Programmabsturz reichen.
```
  printf("%s %s %s %s\n");
```

kann z.B. einfach sinnlosen Quatsch ausgeben, aber auch einen Segmentation Fault auslösen.

Nochmal hinweisen möchte ich auf die Möglichkeit der I/O-Umleitung in allen ernstzunehmenden shells:
```
examples> meinprogramm > ausgabe.txt < eingabe.txt
```

schreibt alles, was meinprogramm ausgibt, in die Datei ausgabe.txt und nimmt die Eingabe aus eingabe.txt.

Nebenbei: Wer hexadezimal drucken möchte, kann das mit dem Formatcode `%x` tun – in C wie in Python.

Die Formatstrings gelten im Wesentlichen auch fürs Einlesen mit `scanf`. Aber: Skalare Argumente für scanf müssen ein `&` haben (warum, sehen wir später). `scanf("%d", &zahl);` liest einen Integer, `scanf("%s", str);` einen String (aber: Dafür nicht scanf nehmen, sondern `fgets`).

Bei scanf ist es sehr wichtig, die Längencodes korrekt zu verwenden, weil die oben erwähnte Promotion der Argumente hier nicht greift.

Das Programm

```
#include <stdio.h>

int main(void)
{
  int i;
  short s;
  double d;
  float f;

  scanf("%d %hd %lf %f", &i, &s, &d, &f);
  printf("%d %d %f %f\n", i, s, d, f);
  scanf("%hd %d %f %lf", &i, &s, &d, &f); /* Falsch! */
  printf("%d %d %f %f\n", i, s, d, f);
  return 0;
}
```

gibt mit der Eingabe

```
500000 50000 3.1415 2.71
500000 5000 3.1415 2.71
```

heute und auf einer i386-Architektur

```
500000 -15536 3.141500 2.710000
458752 5000 3.141499 584860314976236483507101602781593600.000000
```

aus. Abgesehen davon, dass nicht das rauskommt, was wir wollen, kann die zweite Zeile auch zum Absturz führen und wird auf verschiedenen Maschinen verschiedene Dinge tun. Unter MacOS X/PPC beispielsweise gibt das gleiche Programm

```
500000 -15536 3.141500 2.710000
-1591697120 0 50.112022 2.088750
```

aus.

Das Einlesen nichttrivialer Geschichten mit scanf ist etwas kryptisch, weil es allerlei Regeln gibt, wie mit blanks umgegangen wird und auch Schablonen angegeben werden können.

Ein Beispiel:

```
#include <stdio.h>

int main(void)
{
  int zahl, res;

  res = scanf("Zahl: %d", &zahl);
  printf("%d, %d\n", zahl, res);
  return 0;
}
```

```
Eingabe            Ausgabe
Quatsch            -1073743480, 0
Zahl:3             3, 1
Zahl:      8       8, 1
```

Leerzeichen werden also überlesen, der Rückgabewert von scanf ist die Zahl der tatsächlich gelesenen Dinge. Variablen, die nicht gelesen werden, werden nicht verändert – in diesem Fall bleibt `zahl` uninitialisiert.

Unformatierte E/A: `fgets(str, n, stdin);` liest eine bis zu `n` Zeichen lange Zeile nach `str`, `fputs(str, stdout);` gibt einen String aus.

`fgetc(stdin);` liest ein Zeichen und gibt `EOF` zurück, wenn es keine mehr gibt (Falle: `EOF` ist nicht als `char` zu repräsentieren), `fputc(c, stdout)` gibt ein Zeichen aus.

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

int main(void)
{ int c; /* int ist wichtig! */

  c = fgetc(stdin);
  if (c==EOF) {
    printf("Ende der Eingabe.");
  }
  return 0;
}
```

Problems

(17.1)* Probiert folgendes Programm, macht Eingaben und beobachtet die Ausgabe. Macht das so lange, bis ihr versteht, was der Rechner tut.

```
#include <stdio.h>

int main(void)
{
  char str[80];
  int res;

  res = scanf(" %s", str);
  printf("%d %s\n", res, str);
  res = scanf("%s ", str);
  printf("%d %s\n", res, str);
  res = scanf(" %s ", str);
  printf("%d %s\n", res, str);
  return 0;
}
```

(L)

(17.2) Erklärt das -15536 und die zweite Zeile in der i386-Ausgabe des Beispiels mit dem misslungenen Formatstring.

18. Kontrollstrukturen

In Python hatten wir die Sequenz, Iteration (`for`, `while`) und Selektion (`if`). In C finden sich dafür natürlich Entsprechungen.

Die Sequenz ist wieder durch Hintereinanderschreiben zu erreichen; neu sind lediglich die Semikola:
```
msg = "Der Zähler ist jetzt %d\n";
count = 3
printf(msg, count);
```

Selektion in C

Gegenüber Python wichtigster Unterschied: Die Bedingung muss immer in Klammern stehen.

```
if (str[0]=='a') {
  printf("%s fängt mit a an\n", str);
} else {
  printf("%s fängt nicht mit a an\n", str);
}
```

Allgemein:

$$\begin{aligned} \textit{ifClause} ::= &\texttt{"if"}\ \texttt{"("}\ \textit{logicExpr}\ \texttt{")"}\ \textit{CmpdStmt} \\ &[\,\texttt{"else"}\ \textit{CmpdStmt}\,] \\ \textit{CmpdStmt} ::= &\textit{statement} \mid \texttt{"\{"}\,\{\textit{statement}\}\,\texttt{"\}"} \end{aligned}$$

Vorsicht Falle

Verbundanweisungen werden in C durch Klammern und nicht durch Einrückung definiert:

```
if (a%2!=0)
  b = a/2+1;
  printf("a ist ungerade\n");
else
  printf("a ist gerade\n");
```

– im if-Zweig fehlt die Klammer. Schlimmer noch das *dangling else problem*:

```
if (a%4!=0)
  if (a%2!=0)
    printf("a ist ungerade\n");
else
  printf("a ist durch 4 teilbar\n");
```

Das else gehört zum zweiten if! Deshalb: Immer geschweifte Klammern setzen.

19. Iteration

while

`while`-Schleifen funktionieren in C genau wie in Python:

```
while (a>5) {
  printf("Größer als 5: %d\n", a--);
}
```

break und continue

Analog zu Python gibt es `break`, das die engste umschließende Schleife verlässt. In längeren Schleifen liberal benutzt führt es zu Chaos.

Gleiches gilt für `continue`, das aus dem Schleifenkörper unter Auslassung seines Rests direkt zur Schleifenbedingung springt.

Beispiel dazu:

```
while (1) {
  if (!fgets(str, LINE_LEN, inputFile)) {
    break;
  }
  if (str[0]==0) {
    continue;
  }
  printf("%s", str);
}
```

In diesem Beispiel liefert `str[0]` das erste Zeichen von `str`. Wenn dieses Zeichen `0` (nicht `'0'`!) ist, wird die Zeile nicht gedruckt. Im wirklichen Leben bedeutet das, dass das Programmfragment alle Zeilen außer den ganz leeren wieder ausgibt.

do-while

Abhänging von einer Bedingung wird ein Block mindestens einmal ausgeführt, der Test auf die Schleifenbedingung findet am Ende statt.

```
do {
  fgets(str, LINE_LEN, stdin);
  printf("%s", str);
} while (!feof(stdin));
```

Dieses Fragment illustriert eine der häufigsten Fallen bei akzeptierenden Schleifen, nämlich, dass die Logik der meisten Probleme so ist, dass die Schleife eventuell eben gar nicht durchlaufen werden sollte (oder auch beim ersten Mal nur teilweise). Hier prüft das `feof` in der `while`-Bedingung am Ende letztlich, ob das `fgets` fehlerfrei arbeiten kann (außerdem würde das Programm angesetzt auf eine leere Datei möglicherweise abstürzen). Generell gibt es wenig wirklich gute Anwendungen für eine Schleife mit Endbedingung, fast immer ist es verständlicher, eine Schleife mit `break` zu konstruieren.

Der tiefere Grund liegt in der goldenen Regel, Fehlerbedingungen (in dem Fall: „Es gibt nichts mehr zu lesen“) dort abzufragen, wo sie entstehen, hier also beim `fgets` – freundlicherweise gibt `fgets` den speziellen Wert `NULL` zurück, wenn es nichts mehr lesen konnte – deshalb stürzt das Programm oben auch ab, wenn die Datei leer ist, denn das `printf` versucht dann, `NULL` zu drucken, was zum Absturz führt. Das Programm oben sollte jedenfalls besser so aussehen:

```
while (1) { /* Endlosschleife, wir breaken nachher raus */
  if (!fgets(str, LINE_LEN, stdin) {
    break;
  }
  printf("%s", str);
}
```

Python hat keine akzeptierende Schleife, auch wenn do-while natürlich trivial zu simulieren ist:

```
while 1:
  ... loop body ...
  if condition:
    break
```

for

Weil C keine eingebauten Sequenzen hat, muss die for-Schleife etwas anders aussehen, genauer:

forLoop ::= `"for" "("` [*initExpression*] `";"` [*logicalExpression*] `";"` [*updateExpression*] `")"` *CmpdStmt*

Dabei werden im initStatement Schleifenvariablen gesetzt, die Schleife wird abgebrochen, wenn die logicalExpression false wird, und bei jeder Wiederholung der Schleife wird die updateExpression ausgeführt.

Jede der drei Ausdrücke ist optional, kann also weggelassen werden – `for (;a<10;)` ist z.B. äquivalent zu `while(a<10)`.

Die Standardschleife
```
for i in range(n):
  foo(i)
```

sieht also in C so aus:
```
for (i=0; i<n; i++) {
  foo(i);
}
```

In gewisser Weise ist die for-Schleife in C ein allgemeineres Konstrukt als die for-Schleife in Python – es lassen sich wie gesagt alle while-Schleife auch als for-Schleifen ausdrücken.

Das hat auch Nachteile, weil die for-Schleife auf diese Weise gerne missbraucht wird. Generell sollte eine for-Schleife auch in C nur dann eingesetzt werden, wenn man über eine vorher bekannte Zahl von „Objekten“ iterieren will, also in etwa das hat, was in Python eine Sequenz ist.

Noch eine Warnung: Aus Gründen, die bei der Behandlung von Arrays klar werden sollten, sind Schleifenbedingungen der Art `i<=n` immer verdächtig, in der Regel sollten sie die Form `i<n` haben.

Ein weiteres Beispiel zu for:
```
#include <stdio.h>

int main(void)
{
  char c;

  for (c='a'; c<='z'; c++) {
    fputc(c, stdout);
  }
  fputc('\n', stdout);
  return 0;
}
```

Anmerkung: Es gibt keine Garantie, dass das Programm wirklich alle und nur die Kleinbuchstaben des Alphabets ausgibt. Auf IBM-Großrechnern beispielsweise wäre das unter Umständen nicht der Fall, und natürlich auch nicht auf Maschinen mit kyrillischem Zeichensatz. Offenbar fehlen auch die Umlaute. All diese Probleme sind potentiell lösbar – ihr ahnt schon, dass hier wieder mal Locales und Encodings lauern. Der Umgang mit ihnen ist in C deutlich komplizierter als in Python, und meine Empfehlung ist, Aufgaben, in denen man ASCII signifikant verlässt, in Python oder mithilfe geeigneter Bibliotheken (z.B. der noch zu besprechenden glib) zu erledigen.

(19.1)* Schreibt mit `fgetc` und `fputc` – wie in der einfachen Ein- und Ausgabe dargestellt – ein Programm, das seine Standardeingabe auf seine Standardausgabe kopiert, bis das Dateiende kommt. Wenn ihr eure Quelldatei copyi2o genannt habt, könnt ihr es folgendermaßen ausprobieren:

```
copyi2o < copyi2o.c
```

Euer Quelltext sollte ausgegeben werden, das Programm dann anhalten. **(L)**

20. Selektion II: switch

Das `switch`-Statement erlaubt eine Fallunterscheidung nach konstanten Werten. Allgemein hat es die Form:

switchStatement ::= `"switch" "("` *integerExpression* `")" "{"` {*caseExpression*} `"}"`
caseExpression ::= `"case"` *constantLabel* `":"` {*statement*} | `"default" ":"` {*statement*}

Dabei wird die integerExpression ausgewertet und dann gegen jedes constantLabel verglichen, bis eine Übereinstimmung gefunden wird; dann wird die Ausführung dort fortgesetzt, bis ein `break` gefunden wird. Hat keiner der der constantLabels der Wert der integerExpression, wird zum `default`-Label gesprungen.

Beispiel:

```
switch(ch) {
  case 'o':
    openDoor();
    break;
  case 'c':
    closeDoor();
    break;
  default:
    fprintf(stderr, "Unknown command\n");
}
```

Python hat kein `switch`-Statement. Wenn man so etwas braucht, ist es nicht schwer, es durch eine Folge von `if...elif`-Statements zu ersetzen.

Das Beispiel oben könnte etwa so aussehen:

```
if ch=='o':   # case 'o':
  openDoor()
elif ch=='c': # case 'c':
  closeDoor()
else:         # default:
  sys.stderr.write("Unknown command\n")
```

In der Tat ist dieses Konstrukt (das ganz ähnlich auch in C verwendet werden kann) flexibler als `switch` (lädt damit aber natürlich auch zu mehr Missbrauch ein), das mit `switch` keine Bedingungen wie „`ch` zwischen ‘`a`‘ und ‘`z`‘“ oder gar „Funktion `isascii(ch)` ist wahr“ geprüft werden können.

Für das erste Problem gibt es übrigens eine Erweiterung von GNU C. Der GNU C-Compiler versteht Dinge wie

```
case 'a' ... 'z':
  foo();
```

Ob ihr sowas verwenden wollt, ist fraglich. Erstens reduziert ihr damit die *Portabilität* eures Programms, d.h., es wird schwieriger, es ohne den gcc zum Laufen zu bekommen. Angesichts des Umstands, dass es den gcc eigentlich auf allen Rechnern gibt, die man überhaupt ansehen

will, ist das vielleicht kein so großes Problem – vor allem aber will man Zeichenklassifikation gewiss nicht so machen.

Vorsicht Falle: Die Ausführung wird *beim* ersten passenden Label fortgesetzt und geht *bis* zum ersten `break`. Steht am Ende einer caseExpression kein `break`, „fällt“ die Ausführung zum nächsten Zweig durch. Klassischer Fehler:

```
switch(ch) {
  case 'a': printf("a");break;
  case 'b': printf("b");
  default: printf("Fehler");
}
```

Hier wird auch bei jedem b „Fehler“ ausgegeben.

21. Präprozessor

Der Präprozessor von C hat, wie gesagt, mit der eigentlichen Sprache wenig zu tun – er ist aber mittlerweile Teil des C-Standards. Seine Funktion ist, etwas veredelte Suchen-und-Ersetzen-Operationen auf den Quelltext des Programms laufen zu lassen, *bevor* der Compiler überhaupt etwas sieht. Wenn ihr verstanden habt, dass der Präprozessor wirklich kaum mehr als Suchen-und-Ersetzen macht, seid ihr gegen 80% der klassischen Fehler mit dem C-Präprozessor gefeit.

Symbolische Konstanten

Konstanten außer 0, 1 und (manchmal) 2 sollten *nie* im Quelltext vorkommen. In Ausnahmefällen kommen globale Variablen in Frage, normalerweise aber Präprozessor-Konstanten mit aussagekräftigen Namen in Großbuchstaben:

```
#define MAX_LINE_LENGTH 80
#define CONFIG_FILE_NAME ".progrc"
```

Anweisungen an den Präprozessor – Namen werden „als Text“ ersetzt, deshalb *kein* Strichpunkt.

Eine semantische Einheit – ein Name! Nur weil irgendetwas für euch gerade den gleichen Wert hat, sollte es noch lange nicht von einer gemeinsamen Präprozessor-Konstante kontrolliert werden.

Damit ist gemeint, dass, wenn etwa zusätzlich noch
`#define MAX_FNAME_LEN 80`

wäre, die maximale Länge einer Zeile also mit der eines Dateinamens übereinstimmen würde, es trotzdem schlecht wäre, ein Symbol für beide Limits zu verwenden. Was, wenn `MAX_FNAME_LEN` aus irgendwelchen Gründen plötzlich kleiner werden müsste?

Schlecht außerdem:
`#define FIVE 5`

Der Makroname `FIVE` trägt keine Information, die über das Literal 5 hinausgeht. Wirklich schlimm wird das dann, wenn irgendwer auf die schlaue Idee kommt, dass dort, wo bisher die Fünf stand, jetzt besser eine Neun sein sollte – eine Definition
`#define FIVE 9`

ist weit schlimmer als gar keine Makros.

Makros

Der Präprozessor kann auch Makros mit Parametern, was ähnlich wie bei Funktionen aussieht:
```
#define SQR(a) ((a)*(a))
SQR(8);SQR(x+x);SQR(i++);
```

Dabei ist wichtig, dass zwischen dem Makronamen und der öffnenden Klammer kein Blank ist, denn der Präprozessor erkennt dadurch den Unterschied zwischen
```
#define PAR_OPEN (
```

und
```
#define NO_SQR(a) a*a
```

Der Präprozessor macht nur Textersetzungen. Das bedeutet, dass, wann immer irgendwo `NO_SQR(text)` steht, der Präprozessor daraus `text*text` macht, unabhängig davon, was `text` wohl sei, und zwar, noch bevor der Compiler überhaupt zum Code kommt. Das ist beispielsweise beim Ausdruck `NO_SQR(a+a)` ziemlich sicher nicht das, was man haben wollte.

Der Compiler sieht davon:
```
((8)*(8));((x+x)*(x+x));((i++)*(i++));
```

Also: Lieber mehr als weniger Klammern, Namen von Makros in Großbuchstaben als Warnung, dass es sich um Makros handelt, Makros nicht übertreiben und im Zweifelsfall lieber Funktionen nehmen.

Warum will man Makros dann eigentlich haben? Früher war das Hauptargument, dass Makros „schneller“ sind als Funktionen (durch die Textersetzung erübrigt sich ein Funktionsaufruf); heute ist das aus einer Reihe von Gründen nicht mehr so interessant. Nach wie vor kann der gezielte Einsatz von Makros die Les- und Portierbarkeit von C-Programmen aber erheblich verbessern. Es gibt auch einige eher arkane Probleme, die den Compiler vor unlösbare Aufgaben stellen würden, die mit dem Präprozessor aber einfach zu lösen sind.

Bedingte Compilierung

Wir haben schon `#if 0` zum Auskommentieren gesehen. Allgemeiner kann man prüfen, ob bestimmte Konstanten gesetzt sind oder nicht (`#ifdef FOO` bzw. `#ifndef BAR`) oder ob sie einen bestimmten Wert haben (`#if BAZ==0`). Wiederum ist das vor allem wichtig, um Programme portabel zu machen.

So definiert etwa GNU C von selbst das Makro `__GNUC__`; Code, der nur unter GNU C compiliert, kann so vor anderen Compilern versteckt werden. Compiler, die ANSI C versehen, definieren `__STDC__` usf.
```
#include <stdio.h>

int main(void)
{
#ifdef __STDC__
  printf("ANSI-C\n");
#endif
#ifdef __GNUC__
  printf("GNU C\n");
#endif
  return 0;
}
```

Man kann ähnliches auch mit eigenen Konstanten machen, etwa
```
#define SPECTACULAR_SHOW 1  /* undefine to skip flashy intro */

....

#ifdef SPECTACULAR_SHOW
  playIntro()
#endif
```

Problems Seiteneffekt

(21.1)* Ihr erinnert euch vielleicht, dass ihr euch mit `gcc -E` die Ausgabe des C-Präprozessors ansehen könnt. Seht euch an, was der Präprozessor aus folgendem Text macht (der natürlich kein C-Programm darstellt und auch keine korrekte C-Syntax ausspuckt):

```
#define NO_SQR(a) a*a
#define SQR(a) ((a)*(a))
#define ASSIGN(a,b) a = b

Hier ein bisschen Murks:
NO_SQR(8)  Tjaja...  NO_SQR(x+x)    Wird nicht besser...   SQR(i++);

#define Murks besserer Kram

Hier ein bisschen Murks:
SQR(8)  Oho.  NO_SQR(x+x)    Passt scho.   SQR(i++);

#if 0
Das hier könnt ihr gar nicht lesen
#endif

ASSIGN(1,2)
```

Erklärt die Ausgabe des Präprozessors. Fällt euch noch etwas auf? **(L)**

22. Funktionen I

Auch wenn Funktionen in C etwas anders aussehen als in Python, gilt doch weiterhin:

Programme sollten zerlegt werden in überschaubare Funktionen, von denen jede „eine Sache macht, die aber gut“.

In C geben Funktionen immer nur Werte eines bestimmten Typs zurück, und auch alle Elemente der formalen Parameterliste müssen jeweils einen Typ haben. Syntaktisch wird das so gemacht:

funcDefinition ::= *returnType functionName*
"`(`" *parameterList* "`)`" *block*
parameterList ::= [*parSpecifier* { "`,`" *parSpecifier* }] | "`void`"
parSpecifier ::= *typeSpecifier argumentName*

Eine Funktion hat also einen Rückgabetyp (das kann `void` sein für eine Funktion, die nichts zurückgibt), einen Namen, eine Liste von (formalen) Parametern, und einen Funktionsrumpf, in dem lokale Variablen definiert werden können und dann die eigentlichen Anweisungen folgen.

Sehr häufig gibt eine Funktion nicht direkt ein Ergebnis zurück, sondern hat im Wesentlichen nur *Seiteneffekte*. Solche Funktionen sollten nach C-Konvention 0 zurückgeben, wenn sie „funktioniert“ haben, der Seiteneffekt also erreicht wurde (z.B. Daten gespeichert, eine Manipulation an den Argumenten, ein Fenster geöffnet), eine Zahl ungleich 0 sonst (evtl. kann in dieser Zahl kodiert werden, was schief gegangen ist, wirklich üblich ist das aber nicht).

Variablen, die innerhalb von Blöcken definiert sind, sind auch in C lokal (dürfen aber nur am Anfang jedes Blocks definiert werden), Variablen, die außerhalb aller Blöcke definiert sind, sind auch in C global.

Das folgende Programm demonstriert das Verhalten lokaler Variablen:

```
#include <stdio.h>

double power(double base, int exp)
```

```
{
  int i; double res=1;

  printf("base, exp in power: "
    "%f, %d\n", base, exp);
  for (i=0; i<exp; i++) {
    res *= base;
  }
  return res;
}

int main(void)
{
  int exp=2; double base=2.3;

  printf("%f^%d = %f\n",
    base, exp, power(base, exp));
  printf("base, exp in main: "
    "%f, %d\n", base, exp);
  printf("4.2^5=%f\n", power(4.2, 5));
  return 0;
}
```

Seine Ausgabe ist

```
base, exp in power: 2.300000, 2
2.300000^2 = 5.290000
base, exp in main: 2.300000, 2
base, exp in power: 4.200000, 5
4.2^5=1306.912320
```

23. Arrays

Arrays (Felder) sind Sammlungen von Daten *gleicher* Art, z.B. 20 ganze Zahlen. Auf die einzelnen Elemente wird über den *Index*, eine ganze Zahl, zugegriffen, analog zu Python-Sequenzen steht der Index in eckigen Klammern.

In C ist der Index des ersten Elements 0. Das bedeutet, dass der letzte Index eines N-elementigen Arrays $N-1$ ist. Daher auch das Idiom `for (i=0; i<N; i++)`

Auch wenn das alles etwas an Python-Sequenzen, sagen wir Listen, erinnert, sind Arrays doch etwas fundamental anderes. Aus NutzerInnensicht fällt zunächst auf, dass man sich vor der Benutzung entscheiden muss, im Augenblick sogar zur Compilezeit, wie groß das Array ist – C-Arrays können nicht wachsen. Weiter wissen sie nicht notwendig etwas über ihre Länge, es gibt also (in der Regel) keine `len`-Funktion für Arrays. Schließlich stehen in einem Array immer nur Daten gleicher Art, während wir in Listen und Tupeln recht beliebig Daten kombinieren konnten.

Daneben ist das Verhältnis von Arrays zu Listen wiederum ein wenig so wie das von C- zu Python-Variablen. Arrays stellen direkt den Speicherplatz für die Daten, die in ihnen stehen, zur Verfügung, in Python-Listen stehen immer nur Verweise auf die Werte, die in der Liste stehen.

Definition

buffer overflow

Eine Definition eines Array sieht so aus:

arrayDeclaration ::=*typeSpecifier variableName*
"[" [*integerLiteral*] "]" ";"

Beispiele:
```
unsigned int nums[14000];
char string[20];
char powers[]={1,2,4,8,16};
```

Im letzten Beispiel haben wir das Array initialisiert. Wenn wir das tun, können wir die Größe des Arrays weglassen, C macht es automatisch groß genug. Die Initializer sind immer mit Kommata getrennt und stehen in geschweiften Klammern.

Neuere C-Implementationen können auch nur einzelne Elemente initialisieren – der Rest wird dann automatisch auf Null gesetzt:
```
int flags[N_FLAGS]={[3]=23, [N_FLAGS-2]=4, 3};
```

setzt `flags[3]` auf 23 und die letzten beiden Elemente (also `N_FLAGS-2` und `N_FLAGS-1`) auf 4 und 3, alles andere auf 0.

Nicht alle Maschinen vertragen beliebig große Arrays als lokale Variable. Wenn euer Programm abstürzt, bevor es überhaupt losläuft, seht mal nach, ob in main ein riesiges Array definiert wurde. Eine mögliche Abhilfe ist, das Array global zu definieren.

Array-Fallen

In C kann einfach über die Arraygrenzen rausgeschrieben werden, was fast sicher zum Absturz des Programms führt. Viele Cracks basieren auf diesen *buffer overflow*s.
```
void crash1(void)
{
  int i, arr[20];

  for (i=0; i<=20; i++) {
    arr[i] = 0;
  }
}
```

Manche C-Implementationen erlauben per Compiler-Option, zur Laufzeit zu überprüfen, ob über die Arraygrenzen rausgeschrieben wurde, meistens unter irgendwas wie range-checking. Sowas ist nützlich zur Programmentwicklung, kostet in den fertigen Programmen aber natürlich massiv Zeit (die man aber häufig hat).

Zum Teil hilft auch die Hardware bei der Suche nach buffer overflows (das geht dann ohne Laufzeitstrafen). Eine Bibliothek, die so etwas tut, heißt electric fence (hilft aber prinzipbedingt leider nur bei dynamisch allozierten arrays, die wir erst später kennenlernen).

Problems

(23.1) Wenn ihr mal einen Buffer Overflow in Aktion sehen wollt, probiert folgendes Programm (funktioniert so mit gcc 3.3 auf x86, für andere Compiler und Architekturen muss möglicherweise der Index in der Zuweisung geändert werden:
```
#include <stdio.h>

int main(void)
{
  int a[5];

  a[11] = (int)main;
  printf("Hallo Welt.\n");
```

```
  return 0;
}
```

Wenn ihr das Programm laufen lasst, sollte ganz oft „Hallo Welt“ ausgegeben werden, bis das Programm endlich einen Segmentation Fault verursacht. Wenn das nicht so ist, spielt an der 11 rum, bis es passiert. Könnt ihr euch einen Reim darauf machen? **(L)**

24. Arrays II

Arrays und Funktionen

Die Längenangabe von Arrays kann unterbleiben, wenn sie an eine Funktion übergeben werden. Funktionen, die Arrays nehmen, sollten in aller Regel auch ein Argument haben, in dem die Zahl der Einträge im Array steht.

```
void printHistogram(int vals[], int numVals)
{
  int i, j;

  for (i=0; i<numVals; i++) {
    printf("%3d ", vals[i]);
    for (j=0; j<vals[i]; j++) {
      printf("#");
    }
    printf("\n");
  }
}
```

Diese Funktion könnte beispielsweise so verwendet werden:

```
#include <stdio.h>

#define MAX_VALS 20

int main(void)
{
  int vals[MAX_VALS];
  int i;

  for (i=0; i<MAX_VALS; i++) {
    if (scanf("%d", &(vals[i]))!=1) {
      break;
    }
  }
  if (i==MAX_VALS) {
    fprintf(stderr, "Can't handle more than %d values.  Raise MAX_VALS"
      " if you need to.\n", MAX_VALS);
  }
  printHistogram(vals, i);
  return 0;
}
```

Der Effekt einer Deklaration ohne Längenangabe ist jedenfalls, dass die Funktion Arrays „beliebiger“ Länge verarbeiten kann. Davon haben wir vorderhand noch nicht viel, da wir uns immer noch zur Compilezeit festlegen müssen, wie groß die Arrays sind – Abhilfe schafft da erst die dynamische Speicherverwaltung.

Tatsächlich definiert der jüngste C-Standard, meist C99 genannt, *variable length array*s, deren Größe erst zur Laufzeit bestimmt wird. So etwas kann manchmal praktisch sein, ich rate aber vorläufig von ihrer Verwendung ab.

variable length array

Mehrdimensionale Arrays

Häufig hat man zweidimensionale Daten, z.B. die Übergangshäufigkeit von einem Buchstaben 1 auf einen Buchstaben 2. Dann können in C zweidimensionale Arrays verwendet werden. Deklaration etwa durch

```
int langmodel[NUM_ALPHA][NUM_ALPHA];
```

Verwendung wie

```
int loadLangmodel(int langmodel[][NUM_ALPHA])
{
  int i,j;

  for (i=0; i<NUM_ALPHA; i++) {
    for (j=0; j<NUM_ALPHA; j++) {
      if (1!=scanf("%d", &(langmodel[i][j]))) {
        return -1;
      }
    }
  }
  return 0;
}

int getTransFreq(int langmodel[][NUM_ALPHA],
  char c1, char c2)
{
  return langmodel[c1-'A'][c2-'A'];
}
```

Diese Funktionen ließen sich in ein Programm wie

```
#include <stdio.h>

#define NUM_ALPHA 26

int main(void)
{
  int langmodel[NUM_ALPHA][NUM_ALPHA];

  if (loadLangmodel(langmodel)) {
    printf("Oops...\n");
    return 1;
  }
  printf("Übergang von A auf R: %d\n",
    getTransFreq(langmodel, 'A', 'R'));
  return 0;
}
```

einbauen.

Wenn die Dimensionen nicht von vorneherein so fixiert sind wie in diesem Beispiel, ist es meistens besser, eindimensionale Arrays zu verwenden und die zweidimensionale Struktur selbst zu machen. Will man beispielsweise ein Array mit `N` Zeilen und `M` Spalten, geht das so:

```
int arr[M*N];

arr[N*j+i] = 5; /* Setze Zeile j, Spalte i auf 5 */
```

Auf die Weise kann unter anderem das Problem umgangen werden, dass alle Dimensionen bis auf die letzte in jedem Prototypen angegeben werden müssen. Eine weitere Alternative, die je nach Zugriffsmuster bequemer sein kann, ist die „Python-Methode“ – in Python würde man mehrdimensionale Daten durch verschachtelte Sequenzen repräsentieren. Eine Liste, in der andere Listen „stehen“, ist ja in Wirklichkeit nur eine Sammlung von Referenzen auf andere Objekte. Analog kann man in C Arrays von Pointern auf Arrays der eigentlichen Nutzdaten definieren. Wir werden dieser Methode begegnen, wenn wir Arrays von Strings verwalten wollen.

Bibliothek

Problems

(24.1)* Die oben gezeigte Funktion `getTransFreq` enthält eine böse Falle: Sie prüft nicht, ob ihre Argumente im Definitionsbereich der Funktion liegen (d.h. in diesem Fall zwischen `'A'` und `'A'+NUM_ALPHA` – unterlässt man diese Prüfung, gibt die Funktion für unzulässige Argumente kompletten Unsinn zurück, was zwar aus einer theoretischen Perspektive („partielle Funktion“) nicht schlimm sein mag, aber in realistischen Programmen zu den wüstesten Fehlern führen wird.

Seht nach, was mit dieser „naiven“ Funktion bei unzulässigen Argumenten passiert und ergänzt eine Prüfung der Argumente – die Funktion muss dann einen Fehlercode zurückgeben, der hier naheliegenderweise `-1` sein könnte, denn natürlich kann eine „richtige“ Übergangsfrequenz nie `-1` sein.

25. Strings I

Strings (Zeichenketten) sind in C Arrays von `chars`, wobei das Ende des Strings durch eine Null (nicht `'0'` sondern `0`) gekennzeichnet ist.

Das ist ein ungleich primitiverer String, als wir ihn aus Python kennen. Das Problem, dass Strings keine 0 enthalten dürfen, ist noch zu verschmerzen, schlimmer ist schon, dass C-Strings alle Beschränkungen von Arrays haben, also etwa eine zur Compilezeit festgelegte Maximallänge. JedeR C-ProgrammiererIn wird früher oder später über das Ende eines Strings rausschreiben und damit wirre Abstürze und Sicherheitslücken provozieren. Vorsicht gleich mal: Da die abschließende Null natürlich im Array steht, kann ein mit `char str[80]` deklarierter String maximal 79 Zeichen enthalten, und der Index des letzten möglichen Nutz-Zeichens ist 78.

Es gibt *Bibliothek*en, also Funktionssammlungen, die „bessere“ Strings implementieren – nicht zuletzt die Bibliothek des Python-Interpreters. Insbesondere muss man sich bei diesen Strings keine Sorgen mehr darum machen, ob das Array groß genug ist für das, was man gerade vorhat. Wir bleiben hier zunächst bei dem, was uns die Standardbibliothek gibt.

```
#include <stdio.h>

void stringdemo(char str[])
{
  int i=0;

  printf("%s\n", str);
  while (str[i]) printf("%03d ", str[i++]);
  printf("\n");
}

int main(void)
{
  char str1[80]="Ein String";

  stringdemo(str1);
  stringdemo("01\n@gemein");
  return 0;
}
```

gibt (heute) aus:

```
Ein String
069 105 110 032 083 116 114 105 110 103
01
@gemein
048 049 010 064 103 101 109 101 105 110
```

Standard-Stringfunktionen

Die C-Bibliothek bietet einige Funktionen, um den Umgang mit Zeichenketten zu erleichtern. Deklariert werden sie in `string.h`. Übersicht über Stringfunktionen mit `man 3 string`, eine Auswahl:

- `strlen` berechnet die Länge eines Strings ohne das Nullbyte (nicht den Platz im Array!).
- `strcmp` vergleicht zwei Strings und gibt 0 zurück, wenn sie gleich sind.
- `strncpy` kopiert eine Anzahl von Zeichen von einem String in einen anderen (`strcpy` kopiert den ganzen String und kann deshalb zu hässlichen buffer overflows führen).
- `strncat` hängt eine Anzahl von Zeichen aus einem String in einen anderen an.
- `strchr` findet ein Zeichen in einem string (braucht Pointer – später)
- `strstr` findet einen string in einem anderen (ditto)

Ein Beispiel für die Verwendung dieser Funktionen:

```
#include <stdio.h>  /* Wie immer */
#include <string.h> /* Wir wollen Stringfunktionen verwenden */

#define MAX_LEN 80

int mangleStrings(char str1[], char str2[])
{
  char tmp[MAX_LEN*2];
  int res = strcmp(str1, str2);

  printf("len(str1)=%zu, len(str2)=%zu\n", strlen(str1), strlen(str2));
  if (res<0) { /* str1<str2 */
    strncpy(tmp, str1, 2*MAX_LEN); /* erstmal str1 nach tmp schreiben */
    strncat(tmp, str2, 2*MAX_LEN-strlen(tmp));
      /* dann str2 anhängen und dabei aufpassen, dass wir nicht
         überlaufen */
  } else if (res==0) { /* str1==str2 */
    printf("Die Strings sind ja gleich.  Wie langweilig.\n");
    return -1;
  } else { /* str1>str2 */
    /* The same procedure as above, nur kommt jetzt str2 zuerst rein */
    strncpy(tmp, str2, 2*MAX_LEN);
    strncat(tmp, str1, 2*MAX_LEN-strlen(tmp));
  }
  printf("Zusammen: '%s', %zu Zeichen lang\n", tmp, strlen(tmp));
    /* n.b.: strlen zählt die Null am Schluss nicht mit */
  if (strstr(tmp, "Rotz")) {
    /* das können wir auch ohne pointer: Wenn strstr "wahr" zurückgibt,
       ist das "Rotz" in tmp */
    printf("Pfui.  Gleich wäscht du dir den Mund aus.\n");
  }
  return 0;
}

int main(void)
```

```
{
  mangleStrings("Welt", "Hallo ");
  mangleStrings("rot", "rot");
  mangleStrings("rot ", "rot");
  mangleStrings("Sehr Rot", "zigtausend mal");
  return 0;
}
```

Das Programm gibt aus:

```
len(str1)=6, len(str2)=4
Zusammen: 'Hallo Welt', 10 Zeichen lang
len(str1)=3, len(str2)=3
Die Strings sind ja gleich.  Wie langweilig.
len(str1)=4, len(str2)=3
Zusammen: 'rotrot ', 7 Zeichen lang
len(str1)=8, len(str2)=14
Zusammen: 'Sehr Rotzigtausend mal', 22 Zeichen lang
Pfui.  Gleich wäscht du dir den Mund aus.
```

Merke: `strcmp` und Freunde wissen nichts vom Telefonbuch. Sie sortieren nach ASCII-Wert (was vor allem mit Umlauten richtig mistig ist).

Zum %zu-Formatcode, mit dem wir die Ergebnisse von strlen ausgeben, siehe unten bei `size_t`.

26. Strings II

sscanf und snprintf

`scanf` und `printf` gibt es auch in Varianten, bei denen das erste Argument ein String ist – sie operieren dann nicht auf Standardein- und Ausgabe, sondern eben auf dem übergebenen String.

Auch diese Varianten sind in stdio.h deklariert, was, da sie mit I/O kaum etwas zu tun haben, vielleicht etwas seltsam anmuten mag.

`sscanf` bietet *die* korrekte Art, Zahlen aus Strings zu lesen. Die alten Funktionen `atoi`, `atof` und Freunde, die in manchen C-Einführungen noch empfohlen werden, sind demgegenüber Mist, in allererster Linie, weil sie keine brauchbare Möglichkeit bieten, falsche Argumente zu erkennen (also das zu tun, was in Python durch Auslösen eines `ValueError`s realisiert wird).

Um also aus einem String `str` einen int `i` zu lesen, könnt ihr etwas tun wie:

```
if (1!=sscanf(str, "%d", &i)) {
  /* Fehlerbehandlung */
}
```

Analog geht das natürlich für Fließkommazahlen.

Dann und wann bekommt man Strings in bestimmten Formaten geliefert. Wenn z.B. definiert ist, dass drei ints durch Kommata getrennt ankommen, lassen sie sich per

```
if (3!=sscanf(str, "%d,%d,%d", &i1, &i2, &i3)) {
  /* Fehlerbehandlung */
}
```

parsen – da diese Strings häufig bereits durch Rechner generiert werden und bei sscanf eine vernünfigte Fehlerbehandlung möglich ist, ist hier die Verwendung fixer Bestandteile (wie den Kommata im Beispiel) in Formatstrings nicht selten sinnvoll. Das steht im Gegensatz zur Empfehlung, beim normalen scanf keine großen Tricks zu schmeißen. Hintergrund ist, dass beim normalen scanf meist nicht klar ist, wie viel Eingabe verbraucht wurde, bevor ein Fehler auftrat, was also das nächste Zeichen ist, das ein erneutes scanf bekommen würde – man kann sich also schlecht von einem Fehler erholen. Mit sscanf hingegen kann man einfach den kompletten String wegwerfen und einen neuen bestellen.

Will man per `sscanf` Strings in Teilstrings zerlegen, muss man wie immer aufpassen, dass die Zielstrings nicht überlaufen. Deshalb *muss* bei der Verwendung von %s bei allen `scanf`-Varianten immer die Länge angegeben werden, und zwar eins weniger als die Länge des Strings, in den eingelesen wird (die `'\0'` braucht ja auch Platz). Bekommt man etwa Eingabe der Art `key = value`, wobei key keinen Whitespace enhält, vor dem = mindestens ein Whitespace kommt und value, sagen wir, eine Dezimalzahl ist, so ließe sich das so parsen:

```
#define MAXKEYLEN 20
...
char key[MAXKEYLEN];
int value;

/* Warning: Magic 19 here is MAXKEYLEN-1 ! */
sscanf(str, "%19s = %d", key, &value);
```

Das ist natürlich Bockmist – wer `MAXKEYLEN` ändert, müsste auch dran denken, den Formatstring zu ändern, und das wird fast sicher nicht passieren. Wir brauchen also eine bessere Art, den Formatstring zu erzeugen. Sowas würde im Prinzip mit Präprozessorhacks gehen (vgl. unten), einfacher und allgemeiner geht das aber, wenn wir den Formatstring im Programm berechnen. Das wiederum ist ein Klassiker für `snprintf`.

`snprintf` unterscheidet sich von `printf` dadurch, dass es als erstes Argument den String nimmt, in den ausgegeben werden soll, und als zweites Argument dessen Größe. Danach geht es wie gewohnt mit dem Formatstring weiter. Die Funktion gibt die Zahl der hinterlassenen Zeichen (ausgenommen der abschließenden Null) zurück. Die Funktion schreibt aber nie wirklich mehr Zeichen, als ihr zweites Argument angibt, so dass man einfach prüfen kann, ob die Ausgabe abgeschnitten wurde.

Um einen Formatstring des oben benötigten Typs zu erzeugen, können wir etwas wie

```
  char format[FORMAT_LENGTH];

  if (FORMAT_LENGTH<=snprintf(format, FORMAT_LENGTH,
      "%%%ds = %%d", MAXKEYLEN)) {
    printf("Format string too short.\n");
    return 1;
  }
```

machen. Besonders interessant ist `snprintf` auch, wenn Programme formatierte Ausgabe nicht direkt in Dateien schreiben wollen.

Warnung: Es gibt auch `sprintf`, das keine Längenkontrolle eingebaut hat, d.h. so viele Zeichen schreibt, wie es eben schreiben möchte. Die Verwendung von `sprintf` ist praktisch immer ein Bug bzw. eine Sicherheitslücke, weil es fast immer möglich ist, eine Eingabe so zu drechseln, dass *sehr* viele Zeichen geschrieben werden. Also: `sprintf` nicht verwenden.

In Summe: Die Kiste mit dem einlesen von Schlüssel-/Wertpaaren könnte etwa entlang der in folgendem Programm vorgezeichneten Linien gelöst werden:

```
/* A little hack to demonstrate either snprintf or the preprocessor's
 * stringify operation */
#include <stdio.h>
#include <string.h>

#define MAXKEYLEN 4
#define KEYBUFSIZE MAXKEYLEN+1

#ifdef USE_STRINGIFY
#    define STRINGIFY(x) #x
#    define EXPAND_AND_STRINGIFY(x) STRINGIFY(x)
#else
#    define FORMAT_LENGTH 20
#endif
```

```
int main(void)
{
  char key[KEYBUFSIZE];
  int val, itemsRead;

#ifdef USE_STRINGIFY
  itemsRead = scanf("%" EXPAND_AND_STRINGIFY(MAXKEYLEN) "s = %d", key, &val);
#else
  char format[FORMAT_LENGTH];

  if (FORMAT_LENGTH<=snprintf(format, FORMAT_LENGTH,
      "%%%ds = %%d", MAXKEYLEN)) {
    printf("Format string too short.\n");
    return 1;
  }
  itemsRead = scanf(format, key, &val);
#endif

  switch(itemsRead) {
    case 0:
      printf("All botched.\n");
      break;
    case 1:
      printf("Key overflowed or bad format: %s, buffer size: %d\n",
        key, KEYBUFSIZE);
      break;
    case 2:
      printf("%s = %d\n", key, val);
      break;
    default:
      printf("You're kidding me, no?\n");
  }
  return 0;
}
```

Ich habe dabei gleich noch die alternative Lösung mit dem so genannten Stringify-Operator des Präprozessors eingeflochten. Welche Fassung verwendet wird, entscheidet der Compiler anhand des Präprozessorsymbols `USE_STRINGIFY`. Ist es nicht definiert (wie abgedruckt), wird unsere snprintf-basierte Lösung einkompiliert, ist es definiert, die andere, die ihr mit dem, was ihr in diesem Kurs lernt, nicht verstehen könnt. Makrosprachen haben allerdings ihren eigenen Reiz, es ist also bestimmt kein Fehler, sich mal mit dem ISO-C-Standard oder auch einem ausreichend tiefschürfenden Lehrbuch hinzusetzen und nachzuvollziehen, was ich da eigentlich treibe.

Wenn ihr den Kram mit dem Stringify-Code bauen wollt, könnt ihr entweder irgendwas wie
`#define USE_STRINGIFY`

in das Programm schreiben oder, wenn ihr make und eine sh-abgeleitete Shell verwendet, zum Compilieren
`CFLAGS=-DUSE_STRINGIFY make stringifyhack`

sagen (wenn ihr das Programm unter stringifyhack.c abgespeichert habt). Was dabei vorgeht, wird im Kapitel zu make etwas genauer erklärt. Entscheidend ist jedenfalls, dass der Präprozessor beim Aufruf durch make (bzw. den C-Compiler) die Option `-DUSE_STRINGIFY` übergeben bekommt. Das hat die gleiche Wirkung wie das define oben.

Probiert das Programm mit allen möglichen Eingabevariationen und seht nach, wann das klappt und wann nicht.

```
> setenv LC_ALL C
> echo "Ää7a" | localetest
l:Ä u:Ä ia: 0
l:ä u:ä ia: 0
l:7 u:7 ia: 0
l:a u:A ia: 1
> setenv LC_ALL de_DE
> echo "Ää7a" | localetest
l:ä u:Ä ia: 1
l:ä u:Ä ia: 1
l:7 u:7 ia: 0
l:a u:A ia: 1
```

Fig. 5

Das Fazit von all dem sollte sein: Die Verarbeitung und das Parsen von Eingaben sollte man, wann immer möglich, von Python oder einer ähnlichen Sprache aus machen oder einer spezialisierten Bibliothek überlassen. Solche Sachen in C wasserdincht hinzukriegen ist eine hohe Kunst.

Klassifikation von Zeichen

Die C-Bibliothek kann Zeichen klassifizieren, die Funktionen dazu sind in `ctype.h` deklariert. Beispiele:

- `isalpha` Wahr, wenn Argument ein Buchstabe ist.
- `islower` Wahr, wenn Argument ein Kleinbuchstabe ist.
- `isspace` Wahr, wenn Argument ein Leerzeichen ist (z.B. blank, cr, lf oder tab).
- `isdigit` Wahr, wenn Argument eine Dezimalziffer ist

Ebenfalls in `ctype.h` sind Funktionen `toupper` und `tolower` zur Umwandlung von Groß- und Kleinbuchstaben.

locales

Natürlich sind die Klassifikations-Funktionen wieder sprachabhängig. C kennt, ebenso wie Python, *locale*s, um damit zurecht zu kommen. Die Benutzung der Locales ist in C nicht viel anders als in Python:

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

int main(void)
{ int c;

  setlocale(LC_ALL, "");
  while ((c=fgetc(stdin))!=EOF)
    if (!isspace(c))
      printf("l:%c u:%c ia: %d\n", tolower(c),
        toupper(c), !!isalpha(c));
  return 0;
}
```

Dieser Programmierstil kann nicht empfohlen werden – Ausdrücke wie der in der while-Schleife sparen zwar Platz, laden aber zu Schüssen in den Fuß ein, das Fehlen der Klammern wird bei späteren Änderungen zur Falle, das !! ist ein Hack (es ist allerdings garantiert, dass er wahr und falsch auf 1 und 0 normiert). Wer Code von anderen Leuten wartet, kann sich aber fast sicher auf Schlimmeres einstellen.

(cf. Fig. 5)

Das setenv in diesem Beispiel dient zum Setzen einer Umgebungsvariablen (Environment Variable) – quasi eine Variable, die das Betriebssystem verwaltet. Mit sh-ähnlichen Shells (heute die meisten) würde das so gehen:

`export LC_ALL=de_DE`

Unter der DOS-Shell von Windows *könnte* das so aussehen: `set LC_ALL=de_DE`

Allerdings haben die Locales unter Windows ganz andere Namen, und überhaupt ist es für locales die bessere Idee, die einschlägige GUI zu verwenden. Vgl. auch die einschlägige Wiki-Seite[7].

Problems

(26.1) In den guten, alten Zeiten, als das Netz noch das Netz war und Mä... also, damals fanden Leute es lustig, Texte rot13[8] zu verschlüsseln – die Idee war, einfach alle Buchstaben um 13 Plätze zu rotieren, so dass also aus einem A ein N (N ist der 13. Buchstabe des Alphabets, wenn man bei Null mit dem Zählen anfängt), aus einem O aber ein B wird (weil O der 14. Buchstabe ist, 14+13 aber 27 ist und wir nur sechsundzwanzig Buchstaben haben – wie bei unsigned ints fangen wir bei 26 einfach wieder bei Null an. 27%26 ist aber 1 und entspricht damit einem B).

Schreibt eine Funktion, die nachsieht, ob ein Zeichen zwischen a und z oder zwischen A und Z ist und die dann die nötige Transformation macht (Hinweis: ihr tut euch leichter, wenn ihr ein zu transformierendes Zeichen zunächst auf seinen Index (a=A=0, b=B=1 usf) bringt).

rot13 ist selbstinvers: Eine doppelte Anwendung von rot13 bringt den Ursprungstext hervor. Testet euer Programm auch mit einer Konstruktion wie

```
rot13 < rot13.c | rot13
```

Und: Hier ist die rot13-„verschlüsselte“ Lösung:

```
#vapyhqr <fgqvb.u>

vag ebgngr13(vag p)
{
        vs ('n'<=p && p<='m') {
                p = ((p-'n')+13)%26+'n';
        } ryfr vs ('N'<=p && p<='M') {
                p = ((p-'N')+13)%26+'N';
        }
        erghea p;
}

vag znva(ibvq)
{
        vag p;

        juvyr ((p=strgp(fgqva))!=RBS) {
                schgp(ebgngr13(p), fgqbhg);
        }
        erghea 0;
}
```

(L)

[7] http://wiki.cl.uni-heidelberg.de/moin/PythonTipps

[8] http://www.jargon.net/jargonfile/r/rot13.html

27. Pointer I

Nullpointer

Pointer sind die effektivste Art, sich in C in den Fuß zu schießen. Ein Pointer ist eine Variable, die auf einen Speicherplatz zeigt und weiß, was für eine Sorte Daten drinsteht.

Leider lassen sich ohne Pointer kaum nichttriviale Programme in C schreiben, weshalb man gut daran tut, sich mit ihnen anzufreunden.

Man sollte über Pointer ähnlich nachdenken wie über Referenzen in Python: In erster Linie verweist ein Pointer auf etwas. Um an das zu kommen, auf was er verweist, muss man in C eben einen Stern vor den Namen des Pointers malen (in Python kommt man gar nicht an nicht-dereferenzierte Referenzen ran, in C ist das einfach die Adresse, an der der quasi „eigentliche Wert“ steht). Wenn man sich damit vorstellen kann, dass der eigentliche Wert wiederum eine Adresse sein kann, mithin also ein weiterer Pointer und eine weitere Referenz, hat man die Pointer schon fast in der Tasche.

Definition und Verwendung

Ein Pointer wird definiert, indem man einen Stern vor den Variablennamen setzt:

```
char *cp;
int *ptrToInt;
int *(*ptrToIntPtr);
```

Im letzten Beispiel haben wir gleich einen Pointer auf einen Pointer definiert.

Man sieht übrigens häufig Schreibweisen wie `int* a;`, wodurch der Autor wohl sagen will „Es gibt einen Typ `int*`, und `a` ist von diesem Typ“. Diese Idee ist nicht schlecht, verwandelt sich aber in eine böse Falle, wenn man `int* a, b` definiert – C parst das weiterhin als `int *a, b`, also ist `b` vom Typ `int`. Wir werden später `typedef` kennen lernen – damit geht so etwas richtig.

Der Wert eines Pointers ist die Adresse der Speicherzelle, auf die er zeigt. Den Wert der Speicherzelle bezüglich des Typs des Pointers erhält man durch *Dereferenzieren*, wozu der `*`-Operator dient:

```
printf("%c %d\n", *cp, **ptrToIntPtr);
*cp = 'a';
```

Adressoperator

Nach der Definition zeigt der Pointer *irgendwohin*, und ein Dereferenzieren führt fast sicher zum Absturz. Pointer müssen also auf einen les- und schreibbaren Speicher gerichtet werden. Dazu kann der *Adressoperator* `&` dienen, der einen Zeiger auf die dahinterstehende Variable erzeugt:

```
int a, *ip;
ip = &a;
```

Häufiger ist die Verwendung mit Arrays. Ein Array ist in etwa ein konstanter Zeiger auf einen Speicherbereich:

```
int a[20] = {3,2,1}, *ip;
ip = a;
```

Der Name eines Arrays evaluiert zur Adresse seines ersten Elements. Das wird in der zweiten Zeile des Beispiels benützt. Allgemein gilt für ein beliebiges Array: `a = &(a[0])`.

Häufig gibt man Pointer aus Funktionen zurück. Da legale Pointer garantiert verschieden vom *Nullpointer* `NULL` sind, ist die Konvention hier, dass bei Fehlern in der Funktion *NULL* zurückgegeben wird, ansonsten der Pointer. Nützlicher Nebeneffekt der Konvention ist, dass Konstrukte wie

```
if (!(ptr = getAPointer(bla))) {
  /* Fehlerbehandlung */
}
```

eine relativ kompakte Fehlerbehandlung auch ohne Exceptions ermöglichen.

Funktionspointer

Nicht immer ist es ganz einfach, Pointer zu deklarieren. Funktionspointer etwa gehen eigentlich wie in Python (ein Funktionsname ohne Klammern ist ein Pointer auf die Funktion, der Ampersand ist also unnötig), ihr Prototyp ist aber wegen der Operatorpräzedenz etwas komisch, denn

```
char *fun(int);
```

deklariert eine Funktion (die Klammern binden stärker als der Stern), die einen Pointer auf einen char zurückgibt und einen int nimmt. Wenn wir einen Pointer auf eine Funktion, die einen char zurückgibt definieren wollen, müssen wir den Stern enger an den Namen binden, die Deklaration heißt also

```
char (*fun)(int);
```

Neben der aus unseren Python-Erfahrungen zu erwartenden Verwendung als Callbacks kann man mit Funktionspointern auch das Verhalten von Funktionen parametrisieren. Denkbar wäre z.B. eine Funktion, die nur Zeichen eines bestimmten Typs (Zahl, Buchstabe...) ausgibt:

```
void printCharsOfType(char *str, int (*isOfType)(int))
{
  while (*str) {
    if (isOfType(*str)) {
      fputc(*str, stdout);
    }
    str++;
  }
  fputc('\n', stdout);
}
```

Das könnte zusammen mit den Funktionen aus `ctype.h` verwendet werden:

```
int main(void)
{
  printCharsOfType("H4ll0, W31t", isdigit);
  printCharsOfType("Hallo, \n\tWelt", isalnum);
  printCharsOfType("Der, welcher . macht, ist %", ispunct);
  return 0;
}
```

Ausgabe:

```
4031
HalloWelt
,.,%
```

Der Umstand, dass ein Funktionsname ohne Klammern ein legaler Ausdruck und damit ein legales Statement ist, ist eine recht fiese Falle. Will man eine Funktion aufrufen und vergisst die Klammern, dann passiert einfach nichts – und der Fehler ist durchaus nicht immer sofort zu sehen. Compiliert man mit `-Wall` (o.ä.), kommt aber immerhin eine Warnung wie „Statement has no effect“.

Um zu sehen, was Pointer tun, ist es manchmal nützlich, sich vorzustellen, was im Speicher vorgeht. Nach den Definitionen

```
char *sp;
char s[]="str";
int a=513;
int *ap=&a;
int **app=&ap;
sp = s;
```

könnte es im Speicher wie folgt aussehen:

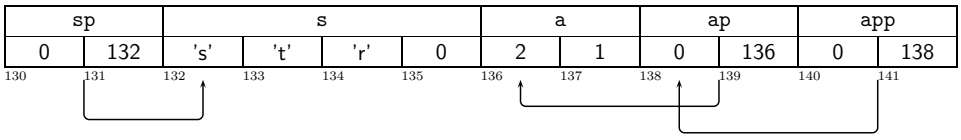


Fig. 6

(cf. Fig. 6)

Tatsächlich sind die Dinge komplizierter – erstens gibt es kaum noch Maschinen, auf denen Integers oder gar Pointer nur zwei Byte lang sind, zweitens würde der Speicher für str vermutlich ganz woanders liegen, und schließlich gibt es noch Fragen der Endianness.

Technik: Endianess

Auf manchen Maschinen wird 513 zur Bytefolge 2,1, auf anderen zur Bytefolge 1,2, je nach dem, ob das höherwertige Byte am Anfang oder am Ende gespeichert wird. Glücklicherweise muss man sich um diese Dinge meistens keine Sorgen machen, weil C einen davon in der Regel isoliert.

Wenn man allerdings binäre Information zwischen zwei verschiedenen Maschinen austauschen will, wird so etwas relevant (und es hilft auch, die Ausgabe von Programmen wie od zu verstehen).

Eine Maschine heißt big-endian, wenn das Byte mit der größten Potenz (most significant byte) im Speicher am Anfang der Zahl steht („big end first“); die Zahl 0x1234abcd würde also als 0x12, 0x34, 0xab, 0xcd gespeichert, in einem Speicherabzug kann man also direkt die Zahlen lesen. Motorola- und Sun-Prozessoren speichern üblicherweise so, auch die diversen Internet-Standards mögen diese Darstellung, weshalb sie auch „network byte order“ heißt.

Intel- und Alpha-Prozessoren hingegen speichern normalerweise little-endian, so dass das Byte mit der kleinsten Potenz hinten steht: 0xcd, 0xab, 0x34, 0x12. Vorteil dieser Darstellung ist, dass Ausdrücke wie `*(char*)l` eher das tun, was man wollen könnte (also: eine als long gespeicherte 12 bleibt auch über einen char pointer gelesen eine 12).

Wer damit spielen will, kann folgendes Programm auf verschiedenen Maschinen probieren (Intel, die RS/6000-Maschinen im URZ, VAXen, die ihr noch irgendwo auftreiben könnt...):

```
#include <stdio.h>

int main(void)
{
  long l=0x1234abcdL;
  int i;
  unsigned char *cp=(unsigned char*)&l;

  for (i=0; i<4; i++) {
    printf("%x ", *cp++);
  }
  printf("\n");
  return 0;
}
```

Wie gesagt: Wenn man „brav“ programmiert, sieht man nichts von der Endianness. Man muss sich aber mit ihr beschäftigen, wenn man z.B. Pointer verschiedener Größe aufeinander castet oder binär gespeicherte Ganzzahlen zwischen verschiedenen Maschinen austauscht, und in diese Kategorie fallen unter anderem auch UTF-16-kodierte Texte in Unicode (das mit der so genannten BOM auch eine Lösung des Problems anbietet – das führt hier allerdings zu weit). Ignoriert man Probleme dieser Art, erhält man unportable Programme (d.h. sie laufen nur auf Prozessoren einer bestimmten Endianess, und das will man in der Regel nicht).

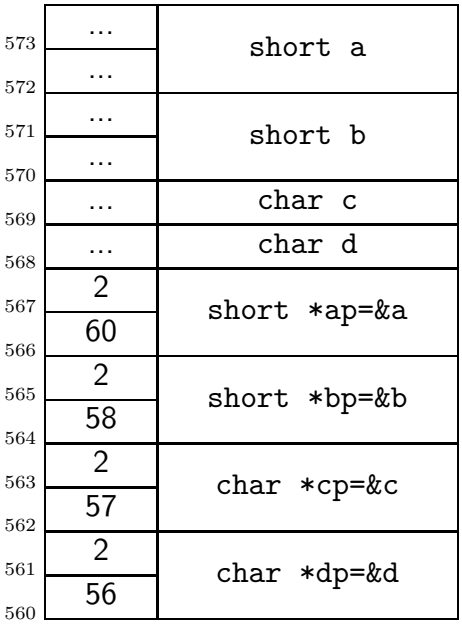


Fig. 7

28. Pointer II

(cf. Fig. 7)

Pointerarithmetik

Einige wenige mathematische Operationen sind auch mit Pointern sinnvoll. Dazu gehören vor allem die Addition einer ganzen Zahl (das Ergebnis ist ein Pointer gleichen Typs) und die Subtraktion zweier Pointer (das Ergebnis ist eine ganze Zahl). Damit kann z.B. folgende Funktion zur Ermittlung der Länge eines Strings geschrieben werden:

```
int strlen(char *str)
{
  char *cp=str;
  while (*cp)
    cp++;
  return cp-str;
}
```

Pointer-Arithmetik findet immer in Einheiten des Typs, auf den der Pointer zeigt statt. In der Situation der Figur ist `ap-bp==1`, und ebenso `cp-dp==1`. Die Differenz in wirklichen Speicherzellen ist durch casten auf `char*` zu erhalten: `(char*)ap-(char*)bp==2`.

Sinn dieser Regelung ist, dass man mit Pointern ähnlich wie mit Arrays umgehen kann. In der Tat *definiert* der C-Standard, dass der Ausdruck `a[i]` für ein Array `a` nichts anderes als `*(a+i)` ist. Das wurde damals unter anderem so gemacht, weil letzteres sich einfach auf die Adressierungsarten üblicher Prozessoren abbilden lässt, und so Arrayzugriffe schnell werden.

Vielleicht fanden Kernighan and Ritichie es auch nur lustig, dass auf diese Weise ein Statement wie `printf("%c\n", 3["abcd"]);` nicht nur legal ist, sondern auch noch ein d ausgibt. Hacker tendieren zu etwas verschrobenem Humor.

Beispiel für die Verwendung von Pointern:

```
void printBigrams(char *str)
{
  char *lead=str+1, *trail=str;

  while (*trail && *lead) {
```

```
    printf("%c%c/", *trail++, *lead++);
  }
  printf("\n");
}
```

Aufruf z.B.:

```
int main(void)
{
  printBigrams("Gallia est omnis divisa in"
    "partes tres");
  return 0;
}
```

Ausgabe:

```
Ga/al/ll/li/ia/a / e/es/st/t / o/om/mn/ne/es/
s / d/di/iv/vi/is/sa/a / i/in/n / p/pa/ar/rt/
te/es/ s / t/tr/re/es/
```

Das ist so natürlich nicht sehr sinnvoll, könnte aber z.B. beim Erzeugen von Sprachmodellen (vgl. Folie „Arrays II“) nützlich sein.

29. Technik: C Calling Convention I

Argumente werden in C *by value* übergeben – Funktionen arbeiten immer mit Kopien ihrer Argumente, können sie also ändern, ohne dass das den Aufrufenden stört.

In Python sieht es so aus, als sei das auch so, der dahinterstehende Mechanismus ist aber ein ganz anderer.

Das Programm

```
def foo(bar):
  bar = bar+1

bar = 7
foo(bar)
print bar
```

gibt natürlich 7 aus, und zwar, weil `bar = 7` im globalen Namespace den Namen `bar` mit dem Objekt 7 verbindet, während der Formalparameter `bar` in der Funktion `foo` im lokalen Namespace von `foo` lebt. Da Python immer zuerst im lokalen Namespace sucht, wird im `bar = bar+1` rechts das lokale `bar` (das nach der Übergabe auch auf 7 verweist) gefragt. Die Zuweisung landet wieder im lokalen Namespace, d.h. das Ergebnis des Ausdrucks auf der rechten Seite, die 8, wird an den Namen `bar` im lokalen Namespace gebunden. Dieser lokale Namespace wird beim Verlassen der Funktion vergessen, so dass zum Zeitpunkt des `print`-Statements `bar` nur noch im globalen Namespace zu finden ist, und dort war und ist es an 7 gebunden. Ähnliches gilt natürlich, für die lokalen Namespaces verschiedener Funktionen. Vgl. auch das `global`-Statement, das in dieses Verhalten eingreifen kann.

Aber: Wenn man Referenzen auf veränderbare Objekte übergibt, können diese Objekte verändert werden – man operiert in diesem Moment nicht mehr mit dem Namen, sondern mit dem Wert, und der hängt natürlich nicht vom Namespace ab. Werden Funktionen so aufgerufen, dass sie ihre Argumente ändern können, heißt das *call by reference*. Wir werden sehen, dass so etwas in C ganz vergleichbar gemacht wird – nur sind in C *alle* Werte prinzipiell veränderbar.

C hat keine Namespaces. Wie also sorgt der Compiler dafür, dass

```
#include <stdio.h>

int baz(int zot) { int bla;
  bla = zot+9; return bla; }
```

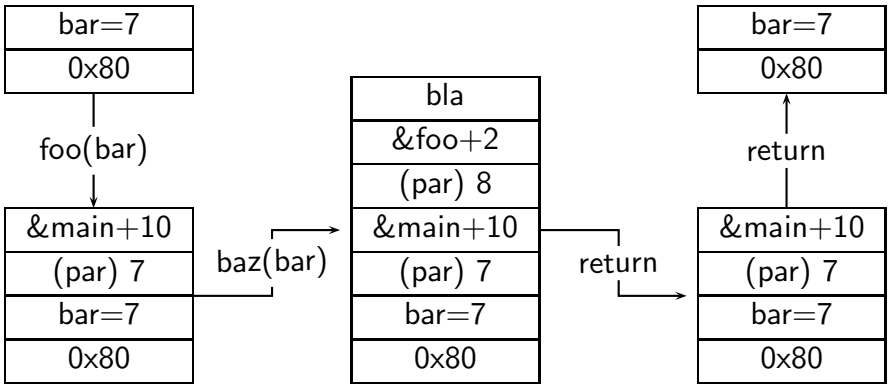


Fig. 8

```
void foo(int bar) { bar = baz(bar+1); }

int main(void) { int bar=7;
  foo(bar); printf("%d", bar);
}
```

die 7 ausgibt?

Lösung: Argumente und lokale Variablen werden auf einem Stack gelagert.

Durch direkte Veränderung des Stackpointers kann man gleich einen ganzen Haufen Zeug von „oben“ wegnehmen, etwa alle lokalen Variablen einer Funktion auf einmal – so kann ein recht effizientes Vergessen überflüssig gewordener Werte realisiert werden.

(cf. Fig. 8)

Diese Grafik zeigt, wie sich der Stack während der Laufzeit des obigen Programms ändern könnte (im Gegensatz zur Darstellung hier „wachsen“ Stacks auf heutigen Maschinen normalerweise nach unten, also zu kleineren Adressen hin). Anfangs (links oben) liegen nur die Rückkehradresse der Funktion main (also die Adresse des Codes, der ausgeführt werden soll, nachdem wir unser main verlassen haben; der Wert 0x80 ist natürlich frei erfunden) sowie der Speicher für die lokale Variable `bar` auf dem Stack. Beim Aufruf `foo(bar)` wird zunächst 7, nämlich eine Kopie von `bar` und danach die Rückkehradresse für die Funktion `foo` auf den Stack geschoben. Das `&main+10` ist wieder frei erfunden; ganz unplausibel ist es aber nicht, dass die CPU 10 Bytes hinter dem Einsprung vom main weiterarbeiten soll, wenn foo fertig ist.

Beim Aufruf von `baz(bar)` passiert wieder ähnliches, wir sehen den Argument 8, eine Rückkehradresse und diesmal Speicher für die lokale Variable `bla`. Bei der Rückkehr aus `baz` muss jetzt nur der Stackpointer (der sagt, was das oberste Element ist) runtergesetzt werden, und alles, was `baz` (am Stack) gemacht hat, ist vergessen; genauso wird bei der Rückkehr aus `foo` verfahren, und wenn wir wieder in main sind (rechts oben), ist der Stack wieder wie zu Beginn.

In der Realität ist das heute nicht so ganz einfach, weil Compiler allerlei Tricks zur Optimierung verwenden, zur Erzeugung von geschicktem Code nämlich. Im hier dargestellten System sind Funktionsaufrufe nämlich relativ teuer (bezüglich der Währung Rechenzeit), und so werden Argumente häufig einfach in Registern übergeben. Zumindest bei nichtoptimiertem Code zumal auf Intel-Prozessoren (die relativ wenige benutzbare Register haben) allerdings sieht das auch heute noch ziemlich wie dargestellt aus.

Wie wir schon am Anfang gesehen haben, wird das Ergebnis von Funktionen üblicherweise in einem Register zurückgegeben.

In diesem Bild sind auch die *Seiteneffekt*e etwas besser fassbar: Wenn die Funktion irgendwelchen Speicher außerhalb des Stacks manipuliert (also nicht ausschließlich auf lokale Variablen zugreift), so hat sie Seiteneffekte. In C muss man relativ viel mit Seiteneffekten operieren – was durchaus als ein weiterer Schwachpunkt zu sehen ist, denn Seiteneffekte sind nicht nur theoretisch schwer

in den Griff zu bekommen, sondern können Programme auch sehr konfus machen, da sie immer mehr oder weniger die Kapselung der Programmlogik in kleine Bereiche verletzen.

Continuations

Call by reference

Wenn wir jetzt übergebene Argumente selbst ändern wollen, geht das offenbar nicht ohne weiteres. Pointer schaffen Abhilfe: Statt der Variablen selbst übergeben wir einen Pointer auf sie. Der Pointer kommt auf den Stack und wird vergessen, die Daten selbst jedoch liegen nicht in dem Bereich des Stacks, der nach dem Ende der Funktion freigegeben wird.

Beispiel:
```
void inc(int *a)
{
  (*a)++;
}
...
int a; inc(&a);
```

In Python gibt es keinen äquivalenten Mechanismus. Das liegt vor allem daran, dass Python ganz grob *immer* call by reference macht. Der Aufrufer übergibt Referenzen, die dann an die lokalen Namen gebunden werden. Unveränderliche Werte können aber natürlich nicht geändert werden – eine Funktion inc wie im Beispiel ist in Python nicht zu machen. Veränderbare Werte kann eine Funktion natürlich verändern, und weil der Aufrufer ja auch eine Referenz auf den Wert hält, bekommt er die Änderungen auch mit, etwa so:
```
def inc(zp):
  zp[0] = zp[0]+1
```

– aber natürlich will man so wirklich nicht programmieren. Der Haupt-Anwendungsfall für „explizites“ (im Gegensatz zum Herumreichen von Referenzen auf allerlei komplizierte Datenstrukturen) call by reference in C ist, mehrere Werte aus einer Funktion zurückzugeben. Python hat dafür Tupel, C nicht (ohne weiteres).

Eine Funktion dieser Art ist `scanf`, das ja dem Aufrufer zusätzlich zu seinem Rückkehrstatus (nämlich der Zahl der gelesenen Elemente) auch noch die gelesenen Werte mitteilen muss. Daraus erklären sich nun auch die vorher mysteriösen &-Zeichen vor skalaren Variablen. Strings dürfen das & natürlich nicht haben, denn ein String ist ja bereits ein Array von chars, also unter Geschwistern ein `char*`, also eine Referenz.

Und noch etwas zu Python: Da Pythons Variablenkonzept ganz anders ist als das von C ist es erstaunlich unproblematisch, Python komplett ohne Stack laufen zu lassen – die Variablen liegen ohnehin nicht auf dem Stack, und statt Rückkehradressen samt Kontexten (z.B. den jeweils aktuellen Namespaces) lassen sich *Continuations* genannte Datenstrukturen verwenden. Wenn man diese Continuations auch Programmen zugänglich macht, kann man ganz erstaunliche Spielchen machen, und ganz grundsätzlich hat ein solches System Vorteile etwa wenn man Programmteile parallel laufen lassen möchte. Mehr zu diesen Fragen findet ihr, wenn ihr das Netz nach stackless python durchsucht.

30. Technik: Maschinensprache II

Die konkrete Umsetzung dieses Rezepts ist im Maschinencode, den gcc erzeugt, zu erkennen. Hier die relevanten Passagen, wie objdump -S sie sieht:

```
#include <stdio.h>

int baz(int zot)
{ int bla;
 8048684:       55              push   %ebp
 8048685:       89 e5           mov    %esp,%ebp
 8048687:       83 ec 04        sub    $0x4,%esp
        bla = zot+9;
 804868a:       8b 45 08        mov    0x8(%ebp),%eax
 804868d:       83 c0 09        add    $0x9,%eax
 8048690:       89 45 fc        mov    %eax,0xfffffffc(%ebp)
        return bla;
 8048693:       8b 45 fc        mov    0xfffffffc(%ebp),%eax
}
 8048696:       c9              leave
 8048697:       c3              ret

void foo(int bar)
{
 8048698:       55              push   %ebp
 8048699:       89 e5           mov    %esp,%ebp
 804869b:       83 ec 08        sub    $0x8,%esp
        bar = baz(bar+1);
 804869e:       8b 45 08        mov    0x8(%ebp),%eax
 80486a1:       40              inc    %eax
 80486a2:       89 04 24        mov    %eax,(%esp,1)
 80486a5:       e8 da ff ff ff  call   8048684 <baz>
 80486aa:       89 45 08        mov    %eax,0x8(%ebp)
}
 80486ad:       c9              leave
 80486ae:       c3              ret

int main(void)
{
        int bar=7;
 80486af:       55              push   %ebp
 80486b0:       89 e5           mov    %esp,%ebp

 80486b2:       83 ec 18        sub    $0x18,%esp
 80486b5:       83 e4 f0        and    $0xfffffff0,%esp
 80486b8:       b8 00 00 00 00  mov    $0x0,%eax
 80486bd:       29 c4           sub    %eax,%esp
 80486bf:       c7 45 fc 07 ... movl   $0x7,0xfffffffc(%ebp)
        foo(bar);
 80486c6:       8b 45 fc        mov    0xfffffffc(%ebp),%eax
 80486c9:       89 04 24        mov    %eax,(%esp,1)
 80486cc:       e8 c7 ff ff ff  call   8048698 <foo>
        printf("%d", bar);
 80486d1:       8b 45 fc        mov    0xfffffffc(%ebp),%eax
 80486d4:       89 44 24 04     mov    %eax,0x4(%esp,1)
 80486d8:       c7 04 24 b4 ... movl   $0x80487b4,(%esp,1)
 80486df:       e8 c8 fe ff ff  call   80485ac <_init+0x38>
        return 0;
 80486e4:       b8 00 00 00 00  mov    $0x0,%eax
}
 80486e9:       c9              leave
 80486ea:       c3              ret
 80486eb:       90              nop
```

Nochmal, um Missverständnissen vorzubeugen: Die Zahlen am Anfang sind zwar Adressen, aber nicht notwendig die, an denen das Programm später laufen wird – das Betriebssystem ist schlau genug, die Adressen zu korrigieren, bevor es das Programm ausführt.

Wir sehen, dass der Compiler wie vor einiger Zeit behauptet jeweils am Anfang den Base Pointer speichert und dann den Wert des Stack Pointers in den Base Pointer schreibt – das gehört zur C Calling Convention auf x86-Maschinen. Sinn der Operation ist, dass (1) Argumente und lokale Variablen jetzt relativ zum Base Pointer adressiert werden können, während wir munter Sachen auf den Stack schieben können und (2) wir am Ende der Funktion den Base Pointer vom Stack holen können und er damit für die aufrufende Funktion wieder stimmt.

Danach holt sich der gcc durch eine Subtraktion vom Stack Pointer Platz auf dem Stack, in den er lokale Variablen und ggf. Argumente von Funktionen schreiben kann (der gcc „simuliert“ den push wohl aus Laufzeitgründen). In `baz` wird eine vier subtrahiert, genug Platz für `bla`, das also bei sp anfängt. Sp liegt zu diesem Zeitpunkt vier Adressen unter bp, und so wird bla später bei bp-4 angesprochen.

Um `zot+9` auszurechnen, holt der Compiler zunächst das Argument nach eax. Das geht über indirekte Adressierung: 0x8(%ebp) bedeutet: Nimm den Wert, der an der Adresse 8 Bytes über der steht, auf die bp zeigt. Über bp steht, so wie das gemacht ist, bp selbst (vom push am Anfang) und dann noch die Rückkehradresse (vom call). Danach steht dann das Argument, das der Aufrufer vor dem call gepusht hat – das geht also auf.

Die Addition mit einer Konstanten wird mit einem immediate-Argument gemacht, und das Ergebnis der Rechnung wird dann nach bp-4 geschrieben – eben dorthin, wo wir uns auf dem Stack Speicher für bla besorgt hatten. Das ist wieder indirekte Adressierung, und objdump stellt die -4 in ihrem Zweierkomplement dar (prüft es nach).

Um den dadurch berechneten Wert zurückzugeben, holt die Maschine den gerade bewegten Wert nach eax zurück. Dann muss sie nur noch aufräumen. Leave kopiert bp nach sp und poppt dann bp (d.h., es schreibt den obersten Wert des Stacks nach bp), was gerade die Beschwörung vom Anfang der Prozedur rückgängig macht. Leave ist in dem Sinn ein Service der CPU-Designer an die AutorInnen von C-Compilern auf x86-Maschinen. Das abschließende ret nimmt den obersten Wert vom Stack und springt dorthin – das call, das uns zu baz geführt hat, hat ja eben dort die Adresse des nächsten auszuführenden Statements hinterlassen.

Ganz offenbar ist dieser Code alles andere als optimal – hier wurde viel zu viel Kram durch die Gegend geschoben. Compiler können merken, wenn sie so einen Unsinn machen und Codesequenzen vereinfachen. Dieser Prozess heißt *Optimierung*. In der Regel muss sie per Hand „angeschaltet“ werden, weil sie die Kompilierung verlangsamt und das Debuggen erschwert (ganz abgesehen davon, dass man glauben muss, dass der Compiler weiß, was er tut). Dennoch: Die Funktion `baz` wird, mit dem Flag `-O2` kompiliert, zu

```
int baz(int zot)
{ int bla;
 8048690:       55                      push   %ebp
 8048691:       89 e5                   mov    %esp,%ebp

        bla = zot+9;
 8048693:       8b 45 08                mov    0x8(%ebp),%eax
        return bla;
}
 8048696:       5d                      pop    %ebp
 8048697:       83 c0 09                add    $0x9,%eax
 804869a:       c3                      ret
```

– was deutlich kürzer ist und wohl auch deutlich schneller läuft (auch wenn das bei modernen CPUs nicht immer leicht zu überblicken ist). Tatsächlich bringt die Optimierung aber nur bei wenigen Programmen auch nur annähernd so viel Nutzen wie man nach diesem einfachen Beispiel erwarten würde.

Wir tun uns jetzt schon etwas leichter mit `foo`. Etwas komisch wirkt hier nur, dass gleich 8 Bytes für lokale Variablen reserviert werden, obwohl keine einzige definiert ist. Vier Bytes davon braucht gcc für das Argument an baz, die anderen vier Bytes werden in der Funktion nicht benutzt – vermutlich hat der Compiler hier beschlossen, sich zur Sicherheit Speicher für das Zwischenergebnis von `bar+1` zu reservieren, falls er ihn brauchen würde. Im optimierten Code gibts sowas natürlich nicht mehr.

alignment

Die nächsten Statements verstehen wir aus dem Stand: Die Maschine holt das Argument aus `8(%ebp)` nach eax und inkrementiert es (das ist das `bar+1`). Dieses Ergebnis kommt auf den Stack. Die Schreibweise `(%esp,1)` bedeutet dabei „die Adresse esp um eine Einheit erhöht“, wobei die Einheit hier vier Bytes sind. Dies entspricht fast einem push auf den Stack, nur dass hier der Stackpointer nicht bewegt werden muss (was klappt, weil der Speicher über dem Stackpointer extra reserviert war). Der Code läuft so etwas schneller als mit einem regulären push, und er ist ja zunächst auch nicht für menschliche Konsumption geschrieben (für uns wäre ein push ganz offenbar klarer).

Dann wird `baz` aufgerufen (was insbesondere impliziert, dass die Adresse des folgenden mov-Statements auf den Stack gepusht wird). Nach der Rückkehr wird das Ergebnis (das in eax zurückkommt) nach `bar` kopiert. Danach ist auch diese Funktion fertig.

In `main` werden diesmal gleich 24 Bytes lokaler Speicher reserviert. Die darauf folgende and-Instruktion sorgt dafür, dass sp auf einer durch 16 teilbaren Adresse steht (die letzten vier bit einer Zahl geben den Rest bei der Division durch 16). So etwas heißt *alignment* (Ausrichtung). Je nach Architektur und Compiler wird auf durch 4, 8 oder sogar 16 teilbare Adressen alignet, weil Zugriffe auf Wörter, Doppelwörter (in sowas werden in der Regel doubles gespeichert) oder sonstige Daten furchtbar langsam werden, wenn sie anders liegen (beim 68000 waren sie sogar ganz verboten). Warum hier auf 16 Bytes alignet wird, entzieht sich meiner Kenntnis. Ebenso weiß ich nicht, was sich der Compiler bei den nächsten beiden (offenbar wirkungslosen) Statements gedacht hat – die Mühe, das im Quellcode des Compilers nachzuvollziehen, lohnt wohl nicht. Die Maintainer des gcc würden wohl argumentieren, dass der Optimierer mit diesen Dingen fertig wird.

Dann wird die Vorbelegung von `bar` erledigt, wie immer indirekt über den bp. Die nächsten beiden Zeilen bringen `bar` auf den Stack – hier kann viel optimiert werden, da eigentlich klar ist, dass das Argument sowieso immer sieben ist (vgl. unten). Nach dem Aufruf von `foo` werden dann die Argumente fürs printf auf den Stack geschoben (beachtet das displacement 0x04 bei bar – das simuliert, dass `bar` zuerst gepusht wird, dann erst die Adresse des Strings `"%d"`).

Der Rest ist mittlerweile wohlbekannt. Am Schluss stehen im Code noch ein paar nops – die nichts machen und, weil vor ihnen ein ret steht, auch nie ausgeführt werden. Sie sind letztlich irgendein Compiler-Voodoo, der vermutlich beim passenden Alignment von Instruktionen oder Spielereien mit dem Instruktionscache der CPU helfen soll. Bei richtig modernen CPUs kann es, das nur nebenbei, tatsächlich sein, dass Programme durch gezieltes Einstreuen von nops schneller werden.

Die optimierte Fassung von main ist

```
int main(void)
{ int bar=7;
 804869c:       55                      push   %ebp
 804869d:       89 e5                   mov    %esp,%ebp
 804869f:       83 ec 08                sub    $0x8,%esp
 80486a2:       83 e4 f0                and    $0xfffffff0,%esp

  foo(bar);
 80486a5:       83 ec 0c                sub    $0xc,%esp
 80486a8:       6a 07                   push   $0x7
 80486aa:       e8 e1 ff ff ff          call   8048690 <foo>
 80486af:       58                      pop    %eax
 80486b0:       5a                      pop    %edx
       printf("%d", bar);
 80486b1:       6a 07                   push   $0x7
 80486b3:       68 94 87 04 08          push   $0x8048794
 80486b8:       e8 ef fe ff ff          call   80485ac <_init+0x38>
       return 0;
}
 80486bd:       31 c0                   xor    %eax,%eax
 80486bf:       c9                      leave
 80486c0:       c3                      ret
```

– das Auseinanderklamüsern davon sei euch überlassen. Das xor von ax mit sich selbst, so viel sei verraten, ist nur eine schnelle Art, die Null nach ax zu bekommen.

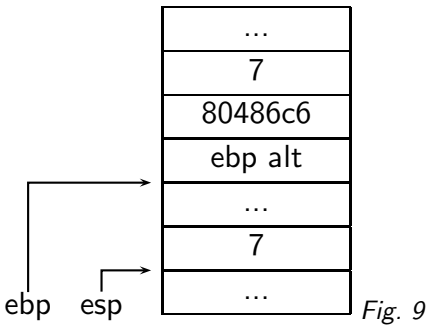


Fig. 9

Der Stack in foo, bevor baz aufgerufen wird (80486a5):

(cf. Fig. 9)

31. Pointer III

In Strings operieren:

```
int readText(char *buf, int len)
{
  char *cp=buf; int c;

  while (((c=fgetc(stdin))!=EOF)
     && cp-buf<len-1) {
    *cp++ = c;
  }
  *cp = 0;
  return cp-buf;
}
```

Arrays von Strings macht man in C fast immer als Arrays von Pointern, `char *strarray[15];` (und nicht als zweidimensionale Arrays von chars `char strarray[15][STR_LEN];`)

Dies ist einer der Fälle, in denen die Deklaration `char *a` nicht äquivalent zu `char a[]` ist. Im zweiten Fall wird tatächlich Speicher angefordert, und zwar `STR_LEN` pro String, und die Strings müssen auch tatsächlich dort liegen. Im ersten Fall haben wir keinen Speicher für die tatsächlichen Strings und können die Strings hinlegen, wo wir wollen (natürlich brauchen wir dafür irgendwo Platz – wie wir ihn bekommen, werden wir in Kürze sehen). Der Vorteil ist natürlich die Flexibilität. Wenn wir Wörter in Natursprache speichern wollen (es gibt sehr lange, die meisten sind aber ziemlich kurz), ist Methode (2) furchtbare Speicherverschwendung – außerdem sind Operationen (z.B. Sortieren) auf solchen String-arrays schwierig und langsam.

Die Kommandozeilenargumente kommen als solche Arrays von Strings:

```
int main(int argc, char *argv[])
{
  int i;

  for (i=0; i<argc; i++) {
    printf("%s\n", argv[i]);
  }
}
```

gibt die Kommandozeilenargumente aus.

Anwendung: Einen Text in Einzelwörter zerlegen. Idee: Den Text in den Speicher lesen, alles, was kein Buchstabe ist, durch Null ersetzen und Pointer auf die ersten Buchstaben der übrig bleibenden Worte sammeln. Ein Programm, das das tut, befindet sich im Anhang.

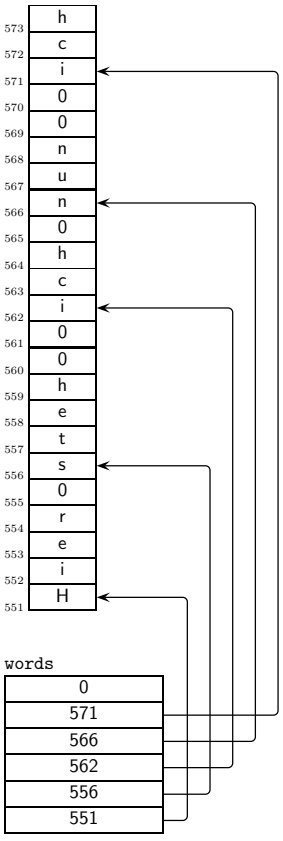


Fig. 10

Rechts das Ergebnis, wenn der String „Hier steh' ich nun, ich“ (armer Tor) ist.

Auf Pointer kann auch der []-Operator angewandt werden, auf Arrays wirkt + wie auf Pointer.

Technik: Adressierungsarten

Wer sich ein wenig mit den Adressierungsarten auseinandergesetzt hat, wird wohl schon ahnen, dass + und [] auf Pointer schlicht der indirekten Adressierung mit Displacement entspricht. Wenn wir unseren (char-) Pointer `ptr` im Register ax haben, lassen sich sowohl `ptr[2]` oder `*(ptr+2)` einfach in `2(%eax)` übersetzen.

Wenn wir jetzt etwas wir `ptr[i]` haben, müsste das Displacement variabel sein, in der Realität wohl aus einem Register kommen. Da aber ohnehin noch die Größe des Basistyps (bei int-Pointern auf 32-bit-Systemen beispielsweise typischerweise 4) eingerechnet werden muss, gibt es eine unabhängige Art des Verschiebens, und zwar über index (das darf ein Register sein) und scale (1, 2, 4 oder 8), so dass Adressen wie `4(%ebp,%eax,4)` möglich sind. Das entspricht der Adresse, die herauskommt, wenn man 4+ebp+4×eax ausrechnet.

Hat der Compiler z.B. dafür gesorgt, dass `i` in eax ist und `int *arr` in edi, so kann `arr[i]` schlicht nach `(%edi, %eax, 4)` kompiliert werden. Das Displacement ist nützlich, wenn Arrays auf dem Stack liegen oder wir Arrays von Structs haben. Dinge, die die CPU schon selbst kann, werden in der Regel auch schnell ausgeführt werden.

Auch andere CPU-Architekturen unterstützen vergleichbare Adressierungen. C-Programmierer-Innen ist das Studium der 68000-Architektur anempfohlen – sie ist auf C zugeschnitten wie kaum eine zweite.

(cf. Fig. 10)

— Dateien zu diesem Abschnitt in der HTML-Version—

Problems

(31.1) Erweitert das auf der Webseite verlinkte Programm lexWords.c so, dass es ein Wort von der Kommandozeile nimmt und dann zählt, wie oft das betreffende Wort in seiner Eingabe vorkommt. Das soll etwa so aussehen:

```
examples> lexWords < lexWords.c
Usage: lexWords <WordToCount>
examples> lexWords char < lexWords.c
char kam 18 mal vor
```

Die Taktik dafür ist recht schlicht: Das Wort, das ihr zählen sollt, ist in `argv[1]` – das werden wir an die Zählfunktion übergeben, es ist ja praktischerweise schon ein `char*`. Die Zählfunktion muss jetzt nur noch durch das `words`-Array gehen, bis `*words` irgendwann `NULL` wird und derweil ein `*words` nach dem anderen zusammen mit dem gesuchten Wort `strcmp` vorlegen. Wenn `strcmp` die beiden Wörter für gleich hält, wird ein Zähler inkrementiert. **(L)**

32. structs

Arrays speichern nur Objekte gleichen Typs. Verschiedene Typen kann man in Strukturen sammeln:

```
struct token {
  char content[MAX_WORD_LEN];
  int type;
  int seen;
};
```

Das erinnert ein bisschen an Pythons Klassen. In der Tat verwenden die objektorientierten Nachfolger von C, allen voran C++, Erweiterungen des struct-Konzepts für ihre Klassen. Mit Funktionspointern können wir structs in der Tat auch mit Methoden ausstatten.

Eine struct-Variable wird so definiert:

```
struct token tok;
```

Zugriff auf die Elemente eines structs (auch: *member*) über den Punktoperator:

```
strcpy(tok.content, "Struktur");
tok.type = TOK_NOUN;
tok.seen = 0;
printf("%s %d %d\n", tok.content,
  tok.type, tok.seen);
```

Die Elemente zweier verschiedener structs haben nichts miteinander zu tun:

```
struct token tok1, tok2;
tok1.type = 1;
tok2.type = 2;
printf("%d %d\n", tok1.type, tok2.type);
```

structs und Pointer

Meistens hat man Pointer auf structs. Der Punkt bindet stärker als der Stern, also `(*tok).contents` schreiben. Abkürzung: `tok->content` – ist auch schöner.

structs und Funktionen

In der Regel nehmen Funktionen keine structs und geben sie auch nicht zurück. Beides ist zwar im Prinzip möglich, aber fast immer unnötig und langsam – structs werden per value übergeben, es muss also immer der komplette struct kopiert werden. Stattdessen sollte mit Pointern auf structs gearbeitet werden:

```
int tokEqual(struct tok *tok1, struct tok *tok2)
{
  return (tok1->type==tok2->type) &&
    !strcmp(tok1->content, tok2->content)
}
...
  struct tok myTok1, myTok2;
  ...
  if (tokEqual(&myTok1, &myTok2))
    ...
```

Für die Rückgabe von Structs über Pointer brauchen wir dynamische Speicherverwaltung, das muss also warten.

Technik: Der Compiler implementiert structs letztlich so, dass er Offsets für alle Felder eines structs berechnet (er sagt euch diese Offsets sogar, wenn ihr ihn mit dem offsetof-Operator danach fragt) – danach kann er einfach mit Displacement adressieren: Aus `r->field` könnte einfach ein `8(%esi)` werden wenn `r` in esi steht und `field` einen Offset von 8 hat. Mit anderen Worten ist auch der Pfeiloperator in sehr schnellen Code zu kompilieren.

33. structs II/typedef

structs initialisieren

Structs können ähnlich Arrays schon bei der Definition Werte bekommen:

```
struct Note_s {
  char *pitch;
  int duration;
};
struct Note_s shortG = {"g", 3};
```

Die Anfangswerte werden in der Reihenfolge angegeben, in der die Elemente in der Strukturdeklaration stehen. Bei größeren structs wird das ziemlich unübersichtlich, weshalb gcc und C99 folgendes erlauben:

```
struct Note_s shortG = {.pitch="g", .duration=3};
```

typedef

Tipparbeit bei structs sparen und Intention klarer machen:

```
typedef int boolean;
typedef char line[LN_LEN];
typedef struct Note_s {
  char *pitch;
  int duration;
} Note;
```

Im Wesentlichen: `typedef` funktioniert wie Variablendefinition, nur wird nicht eine Variable, sondern ein Typ definiert. Nach den Zeilen oben:

```
boolean b;
line l;
Note stillenacht[] = {
{"g", 3},
{"a", 1},
{"g", 2},
};
void playNote(Note *note)
{
  int freq=(int)(computeFreq(note->pitch)+0.5);
  beep(freq, note->duration*BEAT);
}
```

34. Dateien I

Dateien definieren, öffnen und schließen

Die Schnittstelle zu Dateien ist in `stdio.h` deklariert. Dateien können über Variablen vom Typ `FILE` verwaltet werden: `FILE *in;`.

In Python konnten wir Dateiobjekte nur erzeugen, indem wir sie gleich mit einer Datei auf der Platte verbanden (allerdings konnten quasi ungebundene Datei-Objekte entstehen, indem wir die `close`-Methode eines Datei-Objekts riefen). In C ist das ganz ähnlich (nur werden Datei-Objekte durch die Funktion `fopen` erzeugt); da wir aber Referenzen deklarieren müssen, bevor wir sie verwenden, haben wir nach der Definition zunächst eine ungültige Referenz, die eben nicht auf ein `FILE`-Objekt verweist. Alle Versuche, vor der Zuweisung mit `fopen` etwas damit zu machen, werden mit Segmentation Faults und ähnlichem bestraft.

Ein `FILE` ist übrigens in der Regel als struct realisiert.

Die Verbindung einer `FILE*`-Variablen mit einer Datei auf der Platte geht durch
`FILE *fopen(char *path, char *mode);`

`path` ist dabei der Name der Datei, ggf. mit einem Pfad dorthin (ansonsten wird die Datei im aktuellen Verzeichnis gesucht), `mode` ist ein string, in dem z.B. w (Schreibzugiff, Datei wird ggf. neu angelegt, sonst überschrieben), r (Lesezugriff, wenn die Datei nicht existiert, kommt `NULL` zurück) oder a (Daten an bestehende Datei anhängen) stehen kann.

Unter CP/M und seinen Folgesystemen (MS-DOS, Windows) gibt es dazu noch den Modus b („binary“). Wird kein b im Modestring angegeben, so übersetzt die C-Bibliothek die Bytefolge 13 10 (das entspricht CR LF, also etwas wie „Wagenrücklauf, Zeilenvorschub“) beim Lesen durch eine einfache 10, während beim Schreiben aus einer 10 wieder ein 13 10 wird. Grund dafür ist, dass unter diesen Betriebssystemen die Zeilen in Textdateien üblicherweise eben durch CR LF getrennt sind (unter Unix ist nur LF üblich), was einem das Leben bei der Stringverarbeitung etwas schwer macht.

Diese Ersetzung ist aber natürlich tödlich bei Binärdateien (z.B. Bilder oder Audiodateien), in denen gerne auch mal rein zufällig die Bytefolge 13 10 vorkommen kann. Will man Dateien dieser Art verarbeiten, so ist sowohl beim Lesen als auch beim Schreiben das b mit in den Modestring aufzunehmen. Auf anderen Systemen stört das b nicht weiter – unter Python sind die Verhältnisse übrigens analog.

Eine offene Datei sollte mit `fclose(FILE *f)` geschlossen werden, sobald sie nicht mehr gebraucht wird.

Das ist aus ein paar Gründen empfehlenswert: C hält intern Puffer, so dass Material, das in offene Dateien geschrieben wurde, eventuell noch gar nicht auf der Platte angekommen ist, bevor man `fclose` aufruft. Außerdem können Betriebssysteme üblicherweise nicht beliebig viele Dateien offen halten. Sollte auf der Platte ankommen, was ihr geschrieben habt, ohne dass ihr ein fclose gemacht habt, liegt das am netten Service der C-Laufzeitumgebung, alle offenen Dateien zu schließen, wenn main beendet wird. Sollte euer Programm abstürzen, wird das natürlich unterbleiben und eure Dateien sind dann vielleicht leer.

Textdateien

`fprintf` und `fscanf` gehen für Dateien wie von `printf` und `scanf` bekannt, nur ist das erste Argument dann ein `FILE*`. Tatsächlich öffnet die C-Bibliothek immer schon drei Dateien und exportiert sie unter den Namen `stdin`, `stdout` und `stderr` – `printf` ist nur eine Abkürzung für
`fprintf(stdout, ...)`. Außerdem:

- `fgets(string, maxchars, file)` liest eine Zeile, maximal aber `maxchars`-1 Zeichen in einen String
- `fputc(ch, file)` schreibt ein Zeichen in eine Datei
- `fgetc(file)` liest ein Zeichen aus einer Datei
- `fputs(str, file)` schreibt eine Zeile in eine Datei

Ein ganz schlichtes Beispiel: Eine Funktion, die eine Datei öffnet und ausgibt

```
#include <stdio.h>

int catFileChar(char *fName)
{
  FILE *inF=fopen(fName, "r");
  int c;

  if (!inF) {
    return -1;
  }
  while ((c=fgetc(inF))!=EOF) {
    fputc(c, stdout);
  }
  return 0;
}

int main(void)
{
  if (catFileChar("schlicht.c")) {
    fprintf(stderr, "schlicht.c gibt es nicht\n");
  }
  return 0;
}
```

Zur Erinnerung: c ist als int definiert, weil sonst die while-Schleife je nach Maschine entweder nie (unsigned char) oder z.B. für das schöne niederländische Zeichen ÿ (signed char, iso-8859-1, EOF ist -1 – macht euch anhand der Zweierkomplementdarstellung klar, warum das so ist) terminiert.

Ganz ähnlich geht sowas mit den eher zeilenorientierten Funktionen. Dabei besteht natürlich wieder das Problem mit den Arrays fester Größe, bei denen man höllisch aufpassen muss, das man nicht „hinten raus“ schreibt. Eine mögliche Anwendung könnte das Parsen einer einfachen Konfigurationsdatei sein. Zeilen in dieser Konfigurationsdatei können entweder das Format `# Kommentar` oder das Format `key=value` haben – dabei wollen wir uns hier um whitespace nicht kümmern, es geht uns ja vor allem um die Dateien.

```
#include <stdio.h>

#define MAX_LN_LEN 80

void doSomething(char *buf, char *cp)
{
  printf("Key: %s, Value: %s\n", buf, cp);
}

char *skipToEqual(char *cp)
{
  while (*cp && *cp!='=') {
    cp++;
  }
  return cp;
}

void removeLastChar(char *cp)
{
  char *start=cp;

  while (*cp) {
    cp++;
  }
  if (start!=cp) {
    *--cp = 0;
  }
}

void parseConfig(FILE *cfgFile)
{
  char buf[MAX_LN_LEN], *cp, *val;

  while (fgets(buf, MAX_LN_LEN, cfgFile)) {
    if (buf[0]=='#') {
      continue;         /* Ignore Comments */
    }
    cp = skipToEqual(buf);
    if (*cp!='=') {
      fprintf(stderr, "Syntax Error: %s\n", buf);
      continue;
    }
    *cp++ = 0;
    val = cp;
    removeLastChar(cp);
    doSomething(buf, val);
  }
}

int main(int argc, char **argv)
```

```
{
  FILE *cfgFile=fopen(argv[1], "r");

  if (!cfgFile) {
    fprintf(stderr, "Config file %s not found.\n", argv[1]);
    return 1;
  }
  parseConfig(cfgFile);
  return 0;
}
```

(Man sollte natürlich besser prüfen, ob man tatsächlich ein Kommandozeilenargument bekommen hat)

Das Programm könnte etwa so laufen:

```
examples> cat example.cfg
# Ein albernes Beispiel
Kurs=Programmieren II
Dateien=vorl.tex fig_pointer0.tex fig_pointers.tex
Syntaxfehler;
# Ein Kommentar
komisch=dies ist != syntaxfehler (sollte es einer sein?)
tucana:home/msdemlei/coli/lehre/2prog/examples> parsecfg example.cfg
Key: Kurs, Value: Programmieren II
Key: Dateien, Value: vorl.tex fig_pointer0.tex fig_pointers.tex
Syntax Error: Syntaxfehler;

Key: komisch, Value: dies ist != syntaxfehler (sollte es einer sein?)
```

Dies ist übrigens ein ganz einfaches Beispiel für einen (handgestrickten) *Parser*, ein Programm also, das eine Zeichenfolge in eine strukturierte Repräsentation überführt.

Für komplexere Grammatiken wird man sich Parser generieren lassen – etwa von Programmen wie yacc/bison – oder gut untersuchte Parsingalgorithmen mit extern spezifizierten Grammatiken verwenden.

35. Dateien II

Binärdateien

Textdateien enthalten Text, d.h. „friedliche“ Zeichen, die durch einen Zeilentrenner in relativ kurze Zeilen strukturiert wird (das ist LF unter Unix, CRLF unter DOS, CR unter MacOS). Zahlen können so nur ineffizient gespeichert werden. Abhilfe: Binärdateien.

Anzumerken ist, dass binär geschriebene Daten im Allgemeinen nicht „portabel“ sind, d.h. auf Intel-Maschinen geschriebene Daten können auf SPARC-Maschinen nicht ohne weiteres gelesen werden. Mit etwas Pech (und wenn man nicht aufpasst) können selbst verschiedene Compiler auf ein und derselben Maschine unverträgliche Binärdateien schreiben. Aber natürlich gibt es dafür Standards, und man kann Programme so schreiben, dass sie überall lesbare Binärdateien erzeugen – Dateien wie Bilder, Kompressate, Musik- oder Filmdateien sind aus Gründen der Platzersparnis in aller Regel solche Binärdateien, die, wenn sie ordentlich gemacht sind, auch nicht fragen, ob sie auf einem Atari ST oder einem PC unter BeOS geschrieben oder gelesen werden.

Lesen und Schreiben von Binärdateien:

```
size_t fread(void *ptr, size_t size,
  size_t nmemb, FILE *stream);
size_t fwrite(void *ptr, size_t size,
```

```
  size_t nmemb, FILE *stream);
```

Beide Funktionen geben zurück, wie viele Objekte sie wirklich geschrieben oder gelesen haben und nehmen eine Zeiger auf das erste Objekt, die Größe eines Objekts, die Zahl der Objekte die verarbeitet werden sollen, und die Datei, in der die Daten landen sollen. Beispiel: Eine Funktion, die ein Array von longs schreibt:

```
int save_longs(char *fname, long *data,
  size_t n_items)
{
  FILE *targ=fopen(fname, "wb");
  int rtval;

  if (!targ)
    return -1;
  rtval = fwrite(data, sizeof(long), n_items,
    targ)!=n_items;
  fclose(targ);
  return rtval;
}
```

Wir sehen hier zum ersten Mal den `sizeof`-Operator. Er berechnet die Größe seines Arguments in `chars` – das sind in der Regel Bytes. Als Argument kommen sowohl Typen als auch Variablen in Betracht. Außer bei chars sollte man niemals Annahmen über die Größe von Typen machen und diese Entscheidung immer dem Compiler (also sizeof) überlassen.

Der `sizeof`-Operator liefert ein Ergebnis vom Typ `size_t`. Das ist immer eine vorzeichenlose ganze Zahl – ihr Größe hängt allerdings von Compiler und Maschine ab. Dieser Typ wird von der Standardbibliothek immer verwendet, wenn irgendwelche Differenzen von Adressen (oder: Größen im Speicher) gebraucht werden, und deshalb gibt z.B. auch `strlen` einen `size_t` zurück. Ihr solltet `size_t`s *wie* unsigned ints verwenden, aber die Typen nicht mischen, weil ihr nicht wissen könnt, welcher int wohl groß genug ist, um die Daten zu halten.

Das ist bei `printf` ein gewisses Problem – wir müssen, wenn wir `size_t`s ausgeben wollen, ja einen Formatcode angeben, und in dem Formatcode steht dann schon drin, welchen Integer printf zu erwarten hat. Häufig ist das aufgrund der Promotion von Argumenten kein Problem, aber z.B. auf 64-bit-Maschinen mit 32-bit ints kann das ins Auge gehen. Deshalb definiert ANSI den Längencode z, der mit den restlichen Integer-Formatcodes (d, i, o, u, x und Freunde) kombiniert werden darf. Die korrekte Art, einen `size_t` auszugeben, ist also `printf("%zu", sizeof(int))`.

Material zum Spielen:

```
#include <stdio.h>

int main(void)
{ int arr[7], *ptr=arr;
  struct { double foo; char bar;} baz;

  printf("char: %zu, short: %zu, int: %zu, long: %zu, float: "
    "%zu, double: %zu\n", sizeof(char), sizeof(short),
    sizeof(int), sizeof(long), sizeof(float), sizeof(double));
  printf("7-arr of int: %zu, FILE: %zu\n", sizeof(arr), sizeof(FILE));
  printf("struct of double and char: %zu\n", sizeof(baz));
  printf("pointer1: %zu, pointer2: %zu\n", sizeof(ptr), sizeof(void*));
  return 0;
}
```

gibt am gcc/Linux i386/glibc 2.2 aus:

```
char: 1, short: 2, int: 4, long: 4, float: 4, double: 8
7-arr of int: 28, FILE: 148
```

```
struct of double and char: 12
pointer1: 4, pointer2: 4
```

padding

Interessant vor allem die Größe des Structs – Grund für die 3 zusätzlichen Bytes ist das *padding*, mit dem der Compiler dafür sorgt, dass Variablen wenn möglich auf durch vier teilbaren Adressen liegen – Intel-CPUs mögen das, andere bestehen darauf.

Dass diese Größen variieren, zeigt schon der Vergleich mit gcc/MacOS X. Dort gibt das Programm

```
char: 1, short: 2, int: 4, long: 4, float: 4, double: 8
7-arr of int: 28, FILE: 88
struct of double and char: 16
pointer1: 4, pointer2: 4
```

aus. Auf einer 64-bit-Maschine (IBM 260, z.B. aixterm8 im URZ) mit 64-bit-Compiler (und IBM-C-Bibliothek) ergibt sich hingegen

```
char: 1, short: 2, int: 4, long: 8, float: 4, double: 8
7-arr of int: 28, FILE: 88
struct of double and char: 16
pointer1: 8, pointer2: 8
```

Andere Dateioperationen

Einer Datei ist ein Zeiger zugeordnet, der immer auf das nächste Byte zeigt, das gelesen oder geschrieben wird. `long ftell(FILE *stream)` gibt diesen Zeiger zurück,

```
int fseek(FILE *stream, long offset, int whence);
```

setzt ihn. `whence` ist dabei entweder `SEEK_SET`, `SEEK_CUR` oder `SEEK_END`, die den offset als relativ zum Dateianfang, zum augenblicklichen Zeiger, oder zum Dateiende definieren. Das Verstecken von „magischen“ Zahlen hinter mehr oder minder mnemonischen Makros ist gute Programmierpraxis. Leuten, die bei solchen Gelegenheiten `enum` murmeln, sollte man (oft) nicht zuhören. Moderner sind die Funktionen fgetpos und fsetpos, die in mancher Hinsicht portabler sind. Näheres vgl. man-page.

36. Technik: Massenspeicher und Dateisysteme

Der Zugriff auf Dateien ist so komisch gemacht, weil Massenspeicher anders funktionieren als RAM (allerdings erlauben modernere Betriebssysteme, Dateien zu „mappen“ und sie dann wie Arrays zu verwenden – wir werden das in unserer Python-Erweiterung verwenden).

Heute übliche Massenspeicher bestehen aus einer oder mehreren Platten, über denen je zwei Schreib-/Leseköpfe schweben. Mit ihnen kann man je eine Spur anfahren. Jede Spur ist nochmal in Sektoren zu je 512 Byte (oder Vielfachen davon) eingeteilt. Gängige Schnittstellen zu Platten (SCSI) liefern nie Bytes, sondern immer Sektoren.

Nach außen hin sehen nicht allzu antike Festplatten allerdings aus wie eine lange Sequenz von Sektoren – wie sie diese Sektoren intern auf Platten, Spuren und Sektoren verteilen, ist opak, d.h. nicht nach außen sichtbar. Das geht so weit, dass sie „kaputte“ Sektoren einfach aus einer Art Reservepool ersetzen, ohne dass der Rechner überhaupt etwas davon merkt.

Für CDs ist diese Darstellung ohnehin die korrekte, denn sie haben als Erbe ihrer Audio-Herkunft wie die guten alten Schallplatten nur eine Spur, die sich spiralförmig über die ganze nutzbare Fläche windet – der Unterschied zu konzentrisch angeordneten Spuren kann für unsere Zwecke aber vernachlässigt werden.

Die immer üblicher werdenden Flash-Speicher sind eigentlich mehr mit RAM vergleichbar, da bei ihnen die direkte Adressierung einzelner Bytes wenigstens beim Lesen kein Problem wäre. Dennoch sehen die meisten Flash-Speicher nach außen auch wie eine Platte aus, lassen also nur Zugriff auf „Sektoren“ zu. Dies wird vor allem gemacht, weil so gut wie jedes System weiß, wie

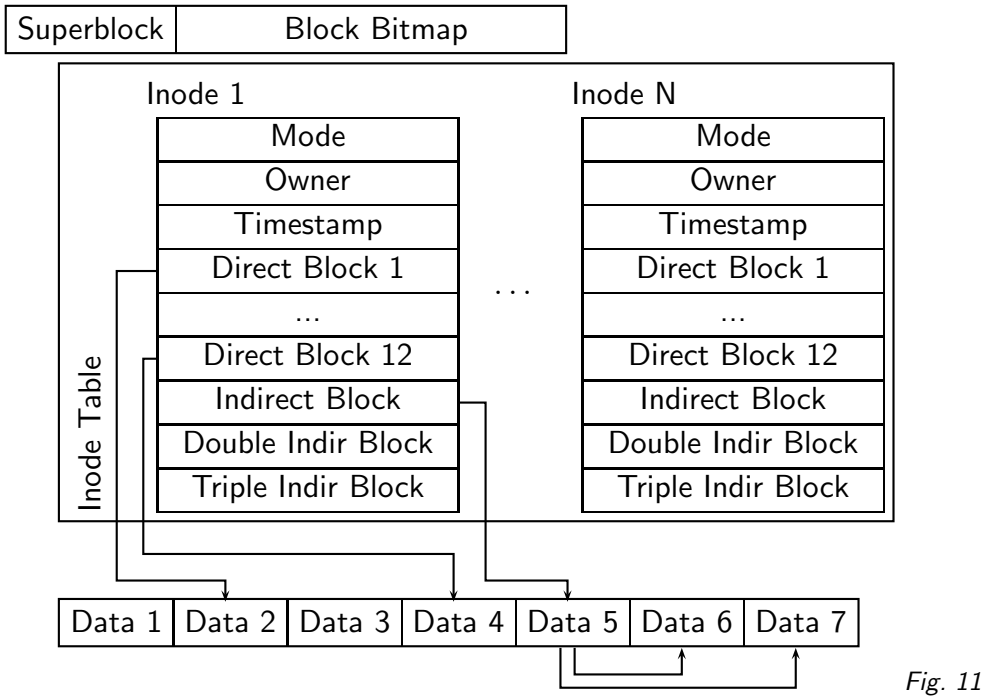


Fig. 11

man mit Platten umzugehen hat – aber auch, weil Flash-Speicher nur sektorenweise beschreibbar ist.

In der Regel ist der Zugriff auf Daten vom Massenspeicher erheblich langsamer als auf Daten im Hauptspeicher. Auch die Bandbreite der Leitung zwischen Platte und Prozessor oder RAM ist deutlich geringer als die zwischen Prozessor und RAM. Unter anderem deshalb unterhalten moderne Betriebssysteme disk caches, in denen vermutlich häufiger gebrauchte Daten von der Platte im RAM gespeichert werden.

Dateisysteme

Um die Daten auf der Platte sinnvoll zu organisieren, verwenden Betriebssysteme ein Dateisystem. Bei dieser Diskussion orientieren wir uns am Standard-Unix-Dateisystem, das weitgehende repräsentativ ist für die Art, in der die meisten Betriebssysteme ihren Massenspeicher verwalten. Als Dateisystem wird dabei sowohl eine konkrete Folge zusammenhängender *Blöcke* (das sind Gruppen von Sektoren konstanter Größe, wobei jeder Block je beispielsweise 4, 8, oder 16 Sektoren umfasst) bezeichnet, die für das Betriebssystem eine Verwaltungseinheit bilden, als auch die Datenstruktur, die zur Organisation der Blöcke dient. Im zweiten Sinn spricht man von einem ext2-Dateisystem, einem NTFS-Dateisystem, einem UFS-Dateisystem, einem ISO-9660-Dateisystem usf – sie unterscheiden sich alle nach der Art, wie sie die Daten im Detail organisieren.

Ein Dateisystem kann so aussehen:

(cf. Fig. 11)

Ein Dateisystem beginnt mit dem Superblock, in dem beispielsweise drinsteht, wo das Wurzelverzeichnis zu finden ist, wie oft das Dateisystem gemountet wurde, wie groß es ist usf. Wegen seiner Wichtigkeit wird der Superblock häufig mehrfach gespeichert.

Es folgt eine Block Bitmap, in der für jeden Block ein Bit steht, das markiert, ob der betreffende Block frei ist oder nicht. Danach folgt eine Liste von so genannten *Inode*s. Jeder Inode entspricht einer Datei und gibt für diese Datei einerseits Metainformationen (Mode sagt etwas über die Zugriffsrechte, Owner ist der/die EigentümerIn der Datei, Timestamp sagt, wann die Datei zuletzt geändert wurde). Andererseits ist hier verzeichnet, in welchen Blöcken die Daten, die zu dieser Datei gehören, gespeichert sind. Dazu hat man zunächst 12 Blöcke, deren Nummern

direkt im Inode stehen. Wenn Dateien wachsen, braucht man mehr Blöcke, für deren Nummern im Inode (dessen Größe ja festliegt) kein Platz mehr ist.

Fragmentierung

hard link

Um Dateien mit mehr als 12 Blöcken verwalten zu können, hat man den Indirect Block, der auf einen Block verweist, in dem weitere Blocknummern stehen, die dann ebenfalls zur Datei gehören. Wenn auch das nicht reicht, kann man in den Double Indirect Block benutzen – einen Block, in dem die Nummern von weiteren Indirect Blocks stehen. Was der Triple Indirect Block ist, könnt ihr euch selbst überlegen. Mithin sorgt das Dateisystem dafür, dass auf kleine Dateien besonders schnell zugegriffen werden kann, weil die zu ihnen gehörenden Blöcke direkt im Inode stehen. Dies ist für Unix-Systeme, auf denen man gerne viele kleine Dateien hat, recht günstig.

Der Rest der Platte wird von den durchnummerierten Datenblöcken eingenommen.

Moderne Dateisysteme sind noch um einiges komplizierter, weil man z.B. eine *Fragmentierung* von Dateien vermeiden will, die dann eintritt, wenn die zu einer Datei gehörenden Blöcke nicht mehr hintereinander liegen und der Lesekopf also wild über die ganze Platte hüpfen muss, um die Daten zu finden.

Ganz moderne Dateisystem (z.B. reiserfs) verlassen das hier dargestellte Konzept fast vollständig und haben evtl. nicht mal mehr wirkliche Inodes. Auf der anderen Seite haben auch ganz primitive Dateisysteme nichts, das wirklich Inodes entsprechen würde (vgl. unten).

Dateien haben keine Namen. Die Namen stehen in speziellen Dateien, den Directories. Von ihnen gehen Zeiger in die Inode Table, die den Namen mit einer Datei verbinden.

Dieses Vorgehen hat unter anderem den Vorteil, dass eine Datei unter mehreren Namen im Filesystem stehen kann (*hard link*).

In einfacheren Dateisystemen werden einige Dinge anderes gehandhabt, wenn auch die meisten Prinzipien ziemlich analog laufen. In FAT-Dateisystemen etwa, die bei DOS üblich waren und nach wie vor für Digitalkameras und ähnliches Standard sind, gibt es keine Inodes. Stattdessen enthält der Directoryeintrag die Metainformationen (bei FAT ist das im Wesentlichen das Datum der letzten Änderung), während die zur Datei gehörigen Blöcke über eine FAT (File Allocation Table) gefunden werden. Dabei hat jeder Block einen Eintrag in der FAT, der auf den ihm *folgenden* Block verweist. Der Verzeichniseintrag enthält nur den Index des ersten Blocks, an dem man dann nachsehen kann, wo der zweite Block liegt, woraus man wiederum die Lage des dritten Blocks ersehen kann und so fort.

Problems

(36.1)* Wenn ihr ein Dateisystem mit Blöcken der Größe 512 Bytes habt und die Gesamtgröße des Dateisystems 1 GiB ist, wie groß ist dann die Block Bitmap in Bytes und in in Blöcken? Welcher Anteil des Gesamtspeichers wird von der Block Bitmap eingenommen? Wie verhält sich dieser Anteil bei wachsendem Gesamtspeicher? **(L)**

(36.2) Wenn wir 512-Byte-Blöcke und 32 Bit lange Blocknummern haben, wie groß kann eine Datei in dem oben dargestellten Dateisystem werden? **(L)**

37. Dynamische Speicherverwaltung

Grundlagen

Bisher: `char text[200];` – aber was, wenn Texte verschiedener Länge bearbeitet werden sollen?

Antwort: Dynamische Speicherverwaltung: Datenstrukturen können zur Laufzeit angelegt werden.

In C geht das „zu Fuß“, jeder Speicherbereich muss eigens reserviert und wieder freigegeben werden.

Tatsächlich sorgt unter Unix das Betriebssystem dafür, dass Speicher und Filehandles (das, was unterhalb von `FILE`s steht) freigegeben werden, wenn das Programm (der Prozess) terminiert – zumindest für FILEs ist es aber schlechter Stil, sich darauf zu verlassen.

In `stdlib.h`:
```
void *malloc(size_t size);
void free(void *ptr);
```

Malloc reserviert size Bytes und gibt einen Pointer darauf zurück (`void*` kann jedem Pointer zugewiesen werden), `free` gibt den Bereich wieder frei. Nach `free` darf auf den Bereich nicht mehr zugegriffen werden.

Ein einfaches Beispiel:
```
int *intptr; /* hier zeigt intptr auf nichts --
  ein Zugriff wird mit einem segfault bestraft. */
intptr = malloc(sizeof(int));
/* Jetzt dürfen wir mit *intptr machen, was wir wollen */
*intptr = 3;
printf("%d\n", *intptr);
free(intptr);
```

Ein etwas sinnvolleres Beispiel:
```
size_t fsize(FILE *f)
{
  size_t sz, pos=ftell(f);
  fseek(f, 0L, SEEK_END);
  sz = ftell(f);
  fseek(f, pos, SEEK_SET);
  return sz;
}

char *readText(FILE *f, int *len)
{
  char *tx;

  *len = fsize(f);
  if ((tx = malloc(*len*sizeof(char)))) {
    fread(tx, 1, *len, f);
  }
  return tx;
}
```

Das sizeof(char) im malloc-Ausdruck tut nichts, weil per definitionem sizeof(char)==1. Trotzdem ist es eine gute Idee, sich die Verwendung von sizeof anzugewöhnen. Ein malloc, in dessen Argument kein sizeof steht, ist verdächtig, denn meistens will man eine Anzahl von Objekten allozieren, nicht eine Anzahl von Bytes.

Darüber hinaus sollte das `f`, das readtext übergeben wird, im Binary Mode geöffnet sein. Unter Unix ist das praktisch immer egal, und DOS/Windows allerdings wird ftell mit nicht-binary geöffneten Dateien hier unter Umständen völligen Unfug zurückgeben. Üblicherweise würde man

die Dateigröße eher in betriebssystemabhängiger Weise bestimmen – es gibt keine voll portable Art, das zu machen. Das liegt nicht zuletzt daran, dass die Zahl der Zeichen, die man aus einer Datei lesen kann, z.B. davon abhängen mag, in welchem Modus sie geöffnet ist.

Unter Unix ist die `stat`-Funktion ein heißer Kandidat für solche Zwecke.

Problems

(37.1)* Schreibt ein Program, das in einer Schleife 10^6 Mal jeweils 10 Bytes alloziert – insgesamt also 10 MB. Natürlich solltet ihr auch aufhören, wenn malloc NULL zurückgibt. Lasst euer Programm dann auf eine Eingabe warten (z.B. mit fgetc – ihr müsst wahrscheinlich return drücken, damit das Programm die Eingabe auch bekommt). Seht euch an, wie viel Speicher das Programm braucht, während es auf die Eingabe wartet.

Wiederholt das Experiment mit einem Programm, das ein Mal 10 MB am Stück alloziert.

Was beobachtet ihr? Habt ihr eine Erklärung?

Anmerkung: Wie ihr den Speicherbedarf eines Prozesses rausbekommt, ist stark systemabhängig. Unter Linux könnt ihr das Kommando `ps u` verwenden und in der VSZ-Spalte nachsehen, unter Windows sagt einem das glaube ich der Prozessmanager, den ihr bekommt, wenn ihr ctrl-alt-del (strg-alt-entf) drückt. **(L)**

38. Dynamische Strings

Dynamische Speicherverwaltung erlaubt uns z.B., eine etwas großzügigere Art von String zu implementieren. Der folgende Code stellt sozusagen Konstruktor und Destruktor für eine Art Klasse dar:

```
typedef struct estr_s {
  int length, cCompatible;
  char *content;
} eStr;

eStr *eStr_new(int len)
{
  eStr *eString = malloc(sizeof(eStr));

  if (eString) {
    eString->length = len;
    if (!(eString->content =
      malloc((len+1)*sizeof(char)))) {
      free(eString); return NULL;
    }
    memset(eString->content, 0, len+1);
    eString->cCompatible = 0;
  }
  return eString;
}

void eStr_free(eStr *eString)
{
  free(eString->content);
  free(eString);
}


eStr *eStr_fromString(char *cString)
{
  eStr *eString=eStr_new(strlen(cString));
  if (eString) {
```

```
    strcpy(eString->content, cString);
    eString->cCompatible = 1;
  }
  return eString;
}
```

name space clash

memory leak

Anmerkungen:

1. Wir definieren uns zunächst einen struct, der die zum Datentyp gehörenden Daten versammelt. Das ist hier nicht schwer: Zunächst wollen wir uns die Länge des Strings merken, dann brauchen wir einen Zeiger auf die Zeichen, aus denen der String besteht, und schließlich richten wir uns noch ein Flag ein, in dem wir notieren, ob der String Nullzeichen enthält oder nicht (wenn nicht, ist `cCompatible` wahr, und einige Standard-Stringoperationen könnten auf die Zeichenfolgen in `content` funktionieren).

2. Wir schreiben vor alle Namen, die wir hier verwenden, ein `eStr_`. Sinn davon ist, *name space clash*es zu verhindern, dass also unser „Modul“ (noch ist es das nicht) Namen überschreibt, die die Programmiererin, die das Modul benutzt, selbst verwendet.

3. Wir haben zunächst einen ganz schlichten Konstruktor `eStr_new`, der einen leeren (mit Nullen gefüllten) String einer bestimmten Länge erzeugt. `memset` ist dabei eine in string.h deklarierte Funktion, die ein Stück Speicher (memory – der Zeiger darauf steht im ersten Argument, seine Länge in chars im letzten) mit dem zweiten Argument füllt. Wir legen im `content`-Feld ein char mehr als nötig an, weil wir „gute Strings“ C-kompatibel speichern wollen. Dieser Konstruktor ist so noch nicht zu viel gut.

4. Die Funktion prüft den Rückkehrwert von malloc sorgfältig. Das ist bei so recht bescheidenen Anforderungen vielleicht overkill, da moderne Maschinen mit virtuellem Speicher praktisch immer ein paar Byte übrig haben; man sollte sich das dennoch angewöhnen, weil am Ende des Tages eben doch schief geht, was schief gehen kann, und wenigstens unsere Basisfunktionen solide arbeiten sollten. Die dauernden Tests werden einem früher oder später auf die Nerven gehen – Pythons Exceptions sind kein überflüssiges Sprachelement.

5. Wenn das Allozieren des Speichers für die Zeichen (`eString->content`) nicht geklappt hat, wird auch der für den `eString` selbst reservierte Speicher freigegeben. Wird sowas vergessen, entsteht ein *memory leak*, ein Leck, durch das auf Dauer immer mehr Speicher verschwindet. Memory leaks gehören zu den häufigsten Fehlern in C-Programmen.

6. Der Konstruktor `eStr_FromString` erzeugt einen `eStr` aus einem normalen C-String. Dabei folgen wir der C-Konvention, das String-Ende durch ein Null zu markieren, um „normale“ Strings C-kompatibel zu halten. Eine Eigenschaft dieser Art sollte natürlich – am besten direkt beim typedef – dokumentiert werden. Wir haben also jetzt schon zwei Konstruktoren für unsere „Klasse“ – das ist auch in wirklichen statisch typisierten objektorientierten Sprachen so: Wenn man polymorphes Verhalten haben will (Funktionen verhalten sich unterschiedlich je nach den Typen ihrer Argumente), muss für jeden Typ, den man behandeln will, eine eigene Methode her. C++ und Java erlauben aber, dass verschiedene Funktionen gleiche Namen haben, solange sie verschiedene Prototypen haben.

7. `eStr_Free` gibt den von einem eStr belegten Speicher frei. Dazu muss man in zwei Schritten vorgehen, und die Reihenfolge dieser Schritte ist wichtig, weil man Speicher, den man freigegeben hat, nicht mehr anfassen darf, auch nicht lesend.

8. Wir haben es hier mit einem wenigstens konzeptionell objektorientierten Design zu tun: Es gibt Konstruktoren, die neue Objekte zurückgeben, und einen Destruktor, der Objekte wieder zerstört. In Python muss man sich in der Regel um das Freigeben der Objekte keine Sorgen machen. Für Fälle, in denen die Freigabe eines Objekts spezielle Operationen braucht (für ein Datei-Objekt könnte das etwa das Schließen der Datei sein), kann man aber die `__del__`-Methode eines Objekts definieren, die dann die Funktion eines Destruktors übernimmt. Man muss allerdings etwas vorsichtig sein, was man in der `__del__`-Methode

treibt, weil der Interpreter bei ihrer Ausführung in einem etwas prekären Zustand sein kann. Mehr dazu in der Python Language Reference[9].

Ein paar Funktionen, die mit diesen eStrs arbeiten:

```
eStr *eStr_add(eStr *op1, eStr *op2)
{
  eStr *dest=eStr_new(op1->length+op2->length);
  if (dest) {
    memcpy(dest->content, op1->content, op1->length);
    memcpy(dest->content+op1->length, op2->content,
      op2->length);
  }
  return dest;
}

int eStr_findChar(eStr *eString, char c, int count)
{
  int i;
  for (i=0; i<eString->length; i++) {
    if (eString->content[i]==c) {
      if (!--count) { return i; }
    }
  }
  return -1;
}
```

Anmerkungen:

1. Wir haben zunächst eine Funktion, die zwei eStrs verkettet, ganz ähnlich dem +-Operator für Strings in Python. Auch hier ist der Code so gemacht, dass die Funktion NULL zurückgibt, wenn etwas schief geht. memcpy funktioniert ähnlich wie memset, nur wird eben eine Anzahl Bytes (im letzten Argument) vom Speicher, auf den das zweite Argument zeigt, in den Speicher, auf den das erste Argument zeigt kopiert.
2. eStr_findChar ist eine Primitivvariante von Pythons find-Methode für Strings (sie kann nur einzelne Chars finden). Bemerkenswert ist allenfalls, wie implementiert ist, dass man nach dem count-ten Auftreten des chars fragen kann.
3. Um den Code auf die Folie zu kriegen, musste ich etwas von meinem üblichen Indentation Style abweichen. Das ist nicht zur Nachahmung empfohlen.

Um daraus ein Programm zu machen, müssen noch die nötigen Header eingebunden (das sind in dem Fall string.h und stdlib.h sowie stdio.h für das folgende main) und z.B. eine Main-Funktion dieser Art dazugeschrieben werden:

```
int main(void)
{
  eStr *kuck=eStr_fromString("Kuckuck, ");
  eStr *ruft=eStr_fromString("rufts aus dem Wald");
  eStr *dest1, *dest2;

  dest1 = eStr_add(kuck, kuck);
  dest2 = eStr_add(dest1, ruft);
  eStr_free(dest1);
  printf("%s %d\n", dest2->content, eStr_findChar(dest2, 'u', 3));
  return 0;
}
```

Dieses Main lässt auch schon ähnen, dass wir ohne weiteres nicht den Komfort von Python werden erreichen können. Einerseits muss für die Stringverkettung natürlich extra eine Funktion aufgerufen werden – der +-Operator lässt sich nicht für eStrs umdefinieren (in C++ und

[9] http://docs.cl.uni-heidelberg.de/python/ref/customization.html

übrigens auch in Python geht dieses so genannte *operator overloading* – vgl. den Exkurs unten). Andererseits, und das wiegt deutlich schwerer, müssen wir uns selbst drum kümmern, Strings, die wir nicht mehr brauchen, freizugeben.

operator overloading

Man kann sich durchaus Systeme überlegen, die nicht mehr benutzte Daten eigenständig freigeben (*Garbage Collectors*) – aber das ist relativ kompliziert. Eine Möglichkeit wäre, regelmäßig über alle allozierten Blöcke zu laufen und nachzusehen, ob das Programm noch irgendwelche Pointer auf diese Blöcke hält und sie, wenn nicht, freizugeben. Problematisch dabei ist, dass man einen Überlick sowohl über die allozierten Blöcke als auch die Pointer des Programms haben muss, und dass die Garbage Collection eine ganze Weile dauern kann. Dennoch funktionieren die meisten Garbarge Collectors nach diesem Prinzip, so etwa die von üblichen Java- oder Lisp-Systemen.

Garbage Collectors

Python hat traditionell ein etwas anderes Verfahren verwendet, nämlich *Refcounting*. Die Idee dabei ist, dass man, wenn man sich einen Pointer auf ein Objekt besorgt, eine dem Objekt zugeordnete Variable, nämlich den Reference Count, inkrementiert, und sie wieder dekrementiert, wenn man den Pointer wieder wegnimmt. Wenn der Refcount eines Objekts auf Null sinkt, wird es freigegeben. Problematisch dabei ist vor allem, dass es einige Disziplin braucht, um die Buchhaltung nicht zu vergessen – in Python selbst merkt man davon natürlich nichts, das Python-System und seine Erweiterungen in C werden davon aber sehr wohl belastet.

Refcounting

Ein anderes Problem des Refcounting sind *zirkuläre Referenz*en. Das ist in folgenem Python-Code zu sehen:

zirkuläre Referenz

```
class Circ:

  def __init__(self):
    self.myself = self
```

Wenn man nun `c=Circ()` sagt, so ist `c.myself` eine zweite Referenz das so erzeugte Objekt, dessen Refcount jetzt also zwei ist. Wenn wir später `c=None` sagen, sollte das Objekt eigentlich freigegeben werden, weil wir damit jede Möglichkeit verloren haben, es nochmal zu benutzen (das Programm hält keine Referenz mehr darauf). Leider ist sein Refcount aber danach immer noch 1, es wird also nicht freigegeben, weil es eben eine Referenz auf sich selbst hat.

Solche Fälle treten nicht sehr häufig auf, und wenn, lassen sie sich meist mit *weak reference*s entschärfen (Modul weakref[10]) – so man denn merkt, dass etwas nicht stimmt und wo es nicht stimmt.

weak reference

Weil sich in komplexere Programme dann aber doch gerne mal (vor allem indirekte) zirkuläre Referenzen einschleichen, haben neuere Python-Fassungen auch einen „richtigen“ Garbage Collector; wer darüber mehr wissen will, sei auf die Dokumentation zum gc-Modul[11] verwiesen.

— Dateien zu diesem Abschnitt in der HTML-Version—

[10] `http://docs.cl.uni-heidelberg.de/python/lib/module-weakref.html`
[11] `http://docs.cl.uni-heidelberg.de/python/lib/module-gc.html`

39. Exkurs: Operator overloading

Wenn man z.B. unsere Edelstrings addieren will, braucht es dazu immer Funktionsaufrufe. Wenn man aber dem +-Operator sagen könnte, was er mit Edelstrings tun soll, könnte man schöne Ausdrücke bauen, und alles würde hübscher aussehen. In C geht das nicht, wohl aber in C++ und Python.

In Python können Klassen durch die Definition geeigneter „magischer“ Methoden sagen, wie Operatoren auf sie wirken sollen – die ganze Liste steht in der Language Reference[12]. Auf diese Weise wird `a+b` zu `a.__add__(b)` und `a%b` zu `a.__mod__(b)`.

Anwendungsbeispiel: Logrithmische Wahrscheinlichkeiten. Im NLP werden Wahrscheinlichkeiten häufig so klein, dass man große Probleme mit der Darstellung dieser Zahlen bekommt. Deshalb rechnet man gerne mit ihren Logarithmen: Der dekadische Logarithmus von 10^{-308} ist eben -308, was natürlich weitaus handhabbarer ist. Üblicherweise macht man dann die Operationen per Hand so, dass sie das richtige für Logarithmen tun. In diesem Fall wollen wir das aber mal Python überlassen. Es gilt:

$$\begin{aligned} L(a) &= \log(a) & L(a \cdot b) &= L(a) + L(b) \\ L(a+b) &= L(a+b) & L(a/b) &= L(a) - L(b) \\ L(a-b) &= L(a-b) & L(a^b) &= L(a) \cdot L(b) \end{aligned}$$

Eine mögliche Implementation:

```
from math import log10

class LogProb:
  def __init__(self, num=1, val=None):
    if val is None:
      self.val = log10(num)
    else:
      self.val = val
  def __repr__(self):
    return "LogProb(10**%f)"%self.val
  def __str__(self):
    return "%f"%(10**self.val)
  def __add__(self, other):
    return LogProb(10**self.val+10**other.val)
  def __sub__(self, other):
    return LogProb(10**self.val-10**other.val)
  def __mul__(self, other):
    return LogProb(val=self.val+other.val)
  def __div__(self, other):
    return LogProb(val=self.val-other.val)
  def __pow__(self, other):
    return LogProb(val=self.val*other.val)

if __name__=="__main__":
  a, b = LogProb(0.4), LogProb(0.9)
  print a*b, b-a, a+b, a/b, a**b
```

Nochmal: In realen Programmen würde man das wahrscheinlich nicht so machen, weil ein solches Programm kreuzlahm laufen würde – glücklicherweise kommen die meisten Algorithmen, für die diese logarithmischen Wahrscheinlichkeiten gut sind, mit Mulitplikationen aus, und so reicht es schon, die Multiplikations- durch Additionsoperatoren zu ersetzen und ggf. noch ein wenig an der Ein- und Ausgabe zu feilen, so dass diese Sorte von Klasse nicht nötig ist. Man könnte aber hier noch sanity checks einbauen (z.B. `num>0` und `num<=1`), was so eine Klasse im Rahmen eines

[12] `http://docs.cl.uni-heidelberg.de/python/ref/specialnames.html`

Experimentiermoduls durchaus sinnvoll wäre, da so Fehler in Algorithmen oder Implementationen schnell auffallen würden.

Objektdatei

Grundsätzlich kann Operator Overloading, wenn es sparsam benutzt wird, durchaus helfen, Programme schöner und verständlicher zu machen. Gerade am Anfang neigt man aber dazu, damit zu großzügig umzugehen und komplett konfuse Overloadings zu machen, die niemand mehr versteht.

Operator Overloading in C++ funktioniert im Groben ganz ähnlich. Normalerweise definiert man dort Methoden der Art

```
int operator+ (double other)
{
  return value+(int)other;
}
int operator+ (int other)
{
  return value+other;
}
```

– hier tritt kein `self` auf, weil C++-Methoden sozusagen von selbst wissen, dass (in diesem Beispiel) `value` sich auf eine Instanzvariable ihres Objekts beziehen (es gibt auch `this`, das etwa Pythons `self` entspricht, nur dass es implizit definiert ist). Aufgrund der Sichtbarkeitsbeschränkungen von Instanzvariablen in C++ geht es aber häufig doch nicht ganz so einfach – aber das gehört nicht hierher.

40. Module

In Python konnten wir unsere Programme in Module gliedern und per `import` auf sie zugreifen. In C ist das etwas komplizierter. Wir wollen unsere Edelstrings zu einem Modul machen.

Vollzieht die Beispiele mit dem Code, der an die Folie „Dynamische Strings“ angehängt ist, nach. Dieser Code hat *keine* Main-Funktion – er soll ja von anderen Programmen benutzt werden, und jedes Programm kann nur eine main-Funktion haben.

Dazu müssen wir zunächst eine *Objektdatei* erzeugen:

```
examples>  make edelstring.o
cc -Wall   -c -o edelstring.o edelstring.c
```

Der Knackpunkt an dem von make erzeugten Kommando ist dabei die Option `-c`, die make auf unsere Anforderung nach `edelstring.o` (anstelle von `edelstring`) hin eingefügt hat. Für den Compiler ist das das Zeichen, nicht den Linker aufzurufen. Da aber erst der Linker weiß, wie man ein ausführbares Programm macht und die Objektdatei jede Menge Referenzen in die C-Bibliothek enthält, die nicht aufgelöst sind, ist `edelstring.o` keine ausführbare Datei:

```
examples> chmod +x edelstring.o
examples> ./edelstring.o
./edelstring.o: Exec format error. Binary file not executable.
Exit 1
```

Der Name Objektdatei hat übrigens nur sehr wenig mit den Objekten mit Sinne von Python zu tun.

In `edelstring.o` befindet sich jetzt der Maschinencode für die Funktionen sowie eine Auflistung der Funktionen, die das Modul bietet, und derer, die es braucht (T bzw. U in der zweiten Spalte, vgl. man nm):

```
examples> nm edelstring.o
00000120 T eStr_add
0000019c T eStr_findChar
...
         U free
00000000 t gcc2_compiled.
```

```
         U malloc
...
```

Keine Prototypen und Typdefinitionen in der Objektdatei – zur Nutzung des Moduls sind sie aber nötig. C kann sie nicht selbst aus der Quelldatei extrahieren. Daher: *Headerfile*

```
/* edelstring.h */
typedef struct estr_s {
  int length;
  char *content;
  int cCompatible;
} eStr;

eStr *eStr_new(int len);
eStr *eStr_fromString(char *cString);
void eStr_free(eStr *eString);
eStr *eStr_add(eStr *op1, eStr *op2);
int eStr_findChar(eStr *eString, char c,
  int count);
```

Im Headerfile stehen also Typen, die von außen sichtbar sein müssen, sowie die Prototypen der Funktionen, die von außen benutzt werden sollen. In der Quelldatei oben muss jetzt die include-Direktive einkommentiert werden, der typedef dafür gelöscht.

Es ist eine *sehr* gute Idee, die Headerdatei eines Moduls in das Modul selbst einzubinden. Hauptgrund ist, dass dadurch Änderungen im Modul, die die durch die Headerdatei definierte Schnittstelle ändern, gleich auffallen. Die Konsistenz von Schnittstelle und Implementation wird *nur* durch die Einbindung der Headerdatei vermittelt, und ein Verlust dieser Konsistenz führt praktisch immer zu wilden Fehlern, die niemand findet.

Um das Modul zu verwenden, bindet man die Headerdatei in das Programm ein, wie wir das schon kennen – nur verwenden wir Anführungszeichen statt spitzer Klammern:

```
#include <stdio.h>
#include "edelstring.h"
int main(void)
{
  eStr *kuck=eStr_fromString("Kuckuck, ");
  eStr *dest1;

  dest1 = eStr_add(kuck, kuck);
  printf("%s %d\n", dest1->content,
    eStr_findChar(dest1, 'u', 3));
  return 0;
}
```

Hintergrund von spitzen Klammers vs. Anführungszeichen ist, dass auch `stdio.h` und Freunde nur Dateien sind. Sie stehen irgendwo im Dateisystem (üblicherweise in der Gegend von /usr/include), und die spitzen Klammern signalisieren dem Präprozessor, er möge sie dort suchen. Mit Anführungszeichen sagen wir dem Präprozessor, dass die Dateien zunächst mal im aktuellen Verzeichnis gesucht werden sollen – und so, wie wir unsere Quelltexte bisher verteilen, wird er bei von uns definierten Headern im Regelfall auch dort fündig werden.

Wenn wir dieses Programm kompilieren, passiert folgendes:

```
examples> make estr_test
cc -Wall    estr_test.c   -o estr_test
/tmp/ccGydNq3.o: In function ‘main’:
/tmp/ccGydNq3.o(.text+0xf): undefined reference to ‘eStr_fromString’
/tmp/ccGydNq3.o(.text+0x27): undefined reference to ‘eStr_add’
/tmp/ccGydNq3.o(.text+0x42): undefined reference to ‘eStr_findChar’
collect2: ld returned 1 exit status
Exit 1
```

– der Linker wirft Fehlermeldungen des Inhalts, dass er die Funktionen in `edelstring.o` nicht gefunden hat. Der Linker sieht eben nicht in den C-Quelltext hinein und kann nicht ahnen, dass wir Funktionen aus `edelstring.o` verwendet haben. Wir müssen ihm das explizit sagen:

```
examples> cc -Wall estr_test.c edelstring.o -o estr_test
examples> ./estr_test
Kuckuck, Kuckuck,  10
```

Dass niemand solche Kommandozeilen eintippen möchte, ist klar. Abhilfe schafft hier make.

make
Makefile
target
prerequisites
Regel
dependencies

41. make I

Das Programm *make* hilft, Projekte zu verwalten. In seinen Steuerdateien, den *Makefile*s, steht, wie Zieldateien oder *target*s aus Quelldateien oder *prerequisites* werden. Make kann damit auch herausfinden, ob ein target neu erzeugt werden muss.

Im einfachsten Fall sieht das so aus:

```
edelstring.o: edelstring.c edelstring.h
  cc -Wall   -c -o edelstring.o edelstring.c
```

Wichtig: Der Whitespace vor dem `cc` ist ein Tab.

Die Bedeutung dieser *Regel* ist: edelstring.o (das target, vor dem Doppelpunkt der ersten Zeile) ist abhängig von edelstring.c und edelstring.h (den prerequisites oder *dependencies* für edelstring.o). Für make bedeutet das, dass – wenn es beschließt, edelstring.o überhaupt zu brauchen – es zunächst mal nachsieht, ob es edelstring.o schon gibt. Wenn das nicht so ist, muss es auf jeden Fall gebaut werden. Wenn es das aber gibt, prüft es zunächst, ob alle prerequisites *älter* sind als edelstring.o – Sinn dieser Prüfung ist, dass, wenn sie erfüllt ist (und nichts komisches mit der Rechneruhr passiert ist), das Ergebnis des Compilerlaufs jetzt nicht verschieden vom letzten Compilerlauf sein dürfte, denn der hat ja etwa zum Zeitpunkt des Timestamps von edelstring.o und damit *nach* der letzten Änderung sowohl von edelstring.c als auch von edelstring.h stattgefunden – und nichts anderes hat (nach diesem Modell) Einfluss auf das Ergebnis der Kompilation. Wenn make diese Situation findet, unterlässt es die Ausführung der Kommandos.

Natürlich ist dieses Modell vereinfacht – das Kompilationsergebnis könnte sich z.B. auch ändern, weil sich das Makefile selbst geändert hat, weil sich der Compilers geändert hat usf. Nun ließen sich viele dieser Parameter im Makefile erfassen (es spricht z.B. nichts dagegen, das Makefile selbst in die Dependencies aufzunehmen, vor allem, wenn man viel am Makefile rumbastelt), doch ist das in der Regel der Mühe nicht wert. Wir kehren später nochmal zu dieser Frage zurück.

Nach der Kopfzeile (die ggf. mit einem Backslash am Ende fortgesetzt werden kann) kommen die Kommandos (die im Wesentlichen einfach der shell übergeben werden), die zur Erzeugung des targets aus den prerequisites nötig sind. Sie werden in der Regel (müssen aber nicht) zur Erzeugung einer Datei mit dem Namen des targets führen.

Es ist wichtig, dass alle Zeilen mit Kommandos mit (mindestens) einem Tab anfangen – diese etwas unglückliche (weil nicht immer sichtbare) Konvention erlaubt make die Unterscheidung der Abhängigkeitszeilen von den Kommandozeilen. Hat man dort z.B. blanks stehen, kommen obskure Fehlermeldungen wie „missing separator“ oder ähnliches.

Was aber, wenn wir plötzlich andere Optionen bräuchten oder lieber mit gcc statt mit cc compilieren möchten? Was, wenn das Programm auf einer anderen Maschine compiliert werden soll, die andere Optionen braucht oder deren C-Compiler vcpp heißt?

Parametrisierung des Makefiles durch Variablen. Make macht Textersetzung, ziemlich analog zum Präprozessor. Konventionell schreibt man makes Variblen groß:

```
CC = gcc
CFLAGS = -Wall
edelstring.o: edelstring.c edelstring.h
  $(CC) $(CFLAGS) -c -o edelstring.o edelstring.c
```

Die Definition von Variablen geht mit einer schlichten Zuweisung, ihre Verwendung ist etwas ungewohnt: Ein $-Zeichen ist das Signal, dass hier eine Variablenersetzung stattfinden soll, und es sind zusätzlich noch Klammern um den Variablennamen nötig.

Meist sind Kommandos für fast alle Regeln gleich. Dafür gibt es Pattern-Rules, die eine Regel für eine ganze Klasse von Transformationen beschreiben, hier z.B. von irgendeiner C-Datei zu irgendeiner Objektdatei:

```
%.o: %.c
  $(CC) $(CFLAGS) -c -o $@ $<
```

Diese Regel besagt, dass, wenn make feststellt, dass es eine Datei foo.o bauen soll, es erstmal sehen soll, ob es eine Datei foo.c sieht. Wenn das so ist, werden die Kommandos ausgeführt, wobei in der Variable `$@` der Name des Targets und in `$<` der Name der ersten Abhängigkeit gespeichert wird.

Normalerweise kennt make schon die üblichsten Pattern-Regeln – was bei einem konkreten make so eingebaut ist, zeigt `make -p`. Wenn man nichts Spezielles vor hat, ist die Regel oben also meist überflüssig.

Die Pattern-Rules sind eine Erweiterung des GNU make über das originale Uralt-make, das für diesen Zweck so genannte suffix rules hatte. Diese funktionieren ganz ähnlich, sehen aber erheblich blöder aus und enthalten mehr Fallen. Es ist heute m.E. durchaus zulässig, von Leuten, die Programme komplilieren wollen, GNU make zu verlangen, so dass ich die Verwendung von Pattern-Rules gegenüber suffix rules empfehle.

Damit sind nun leider unsere Abhängigkeiten verloren gegangen. Make akzeptiert aber „einsame“ Abhängigkeitszeilen ohne Kommandos und fügt die Kommandos dann aus Pattern Rules ein.

```
edelstring.o: edelstring.c edelstring.h
estr_test.o: estr_test.c edelstring.h
```

Das Erzeugen und Aktuellhalten dieser Abhängigkeiten ist auch mühsam; um sich diese Arbeit zu sparen, kann man das Programm makedepend einsetzen, das dies automatisch erledigt. Der Aufruf

```
makedepend -s "# Don't delete this line -- makedepend depends on it" \
  edelstring.c estr_test.c
```

hängt z.B. Zeilen wie

```
# Don't delete this line -- makedepend depends on it
edelstring.o: /usr/include/stdlib.h /usr/include/features.h
edelstring.o: /usr/include/sys/cdefs.h /usr/include/gnu/stubs.h
edelstring.o: /usr/lib/gcc-lib/i686-pc-linux-gnu/2.95.3/include/stddef.h
edelstring.o: /usr/include/string.h edelstring.h
...
```

an das Makefile an. Offensichtlich werden auch die Standard-Header berücksichtigt (es gibt Versionen von makedepend, die das unterdrücken können). Der Sicherheit halber sollte immer die -s-Option angegeben werden, weil verschiedene makedepends verschiedene Default-Markierungszeilen verwenden und ohne diese Option Makefiles, die mit Dependencies ausgeliefert werden (sollte man nicht tun) auf anderen Maschinen komische Fehlermeldungen liefern (weil prerequisites in den Regeln stehen, die es auf der anderen Maschine gar nicht gibt).

Normalerweise wird makedepend aus einer Regel depend im Makefile heraus verwendet, was etwa so aussehen könnte:

```
SRCS = file1.c file2.c ...
depend:
  makedepend -s "# Don't delete this line -- makedepend depends on it"\
    -- $(CFLAGS) -- $(SRCS)
```

Die Idee ist, dass alle Abhängigkeiten auf dem neuesten Stand sind, wenn man `make depend` tippt. Die CFLAGS sind hier nötig, weil man dort etwa zusätzliche Verzeichnisse angeben kann, in denen nach Headerdateien gesucht werden soll – makedepend muss sowas wissen.

Jetzt müssen wir nur noch dafür sorgen, dass unser Programm insgesamt gebaut wird. Eine mögliche Regel dafür könnte so aussehen:

```
OBJS = estr_test.o edelstring.o
estr_test: $(OBJS)
  $(CC) -o $@ $(LDFLAGS) $(CFLAGS) $^
```

Diese Regel ist etwas „verschwenderisch“. Zum einen müsste hier eigentlich nur noch der Linker aufgerufen werden (wir haben ja nur noch Objektdateien), aber die üblichen Compilerfrontends wissen selbst, was sie zu tun und zu lassen haben, wenn sie Objektdateien bekommen. Zum zweiten müssten hier auch nicht mehr die CFLAGS angegeben werden, denn es wird ja nicht mehr kompiliert. Es ist aber auch bequem, sie drin zu lassen, weil z.B. Compilerflags zum Erzeugen von Debugging- oder Profiling-Information durchaus Einfluss auf das Linkergebnis haben können und man auf diese Weise auf der sicheren Seite ist, ohne Schaden anzurichten.

Die von make vordefinierte automatische Variable `$^` fügt einfach alle Dependencies ein. In der Tat sieht in etwa so auch die Pattern Rule von „viele .o“ auf „Datei ohne Extension“ aus, man hätte also auch einfach
`estr_test: $(OBJS)`

ohne weitere Kommandos schreiben können.

Make steuert auch die Erzeugung dieser Folien. Ein paar Ausschnitte aus dem Makefile:

```
%.epsi:%.tex
  tex $<
  dvips $*
  ps2epsi $*.ps
  rm -f $*.ps $*.log $*.dvi
...
view: folien.ps
  gv -seascape -media A4r folien.ps
...
skript.ps: skript.dvi
  $(DVIPS) $(DVIPS_SKR) skript.dvi
  rm skript.dvi

skript: skript.ps
```

Hier habe ich eine weitere automatische Variable verwendet, `$*`, die den Namen des Targets ohne Extension enthält – heißt das Target etwa `fig_foo.epsi`, expandiert `$*` zu `fig_foo`.

make als deklarative Sprache

make ist nicht nur als Hilfsprogramm interessant – tatsächlich ist es ein Prototyp für anwendungsspezifische Spezialsprachen. Das sind, häufig in Anwendungen eingebettete formale Sprachen, die die Definition der in der Anwendung benötigten Daten und Verfahren besonders leicht machen (sollen). Als solche sind sie häufig eher deklarativer Natur, wie eben auch make.

Der Vorteil deklarativer Sprachen wird hier auch recht klar. Ein Makefile-Fragment wie

```
foo: foo.c
  gcc -o foo foo.c
```

(das ja letztlich Abhängigkeiten zwischen verschiedenen Dateien „deklariert“ lässt sich noch recht leicht in eine prodezurale Form bringen (ich verwende ein paar Funktionen, die es so nicht gibt, die aber leicht zu definieren wären):

```
if (!exists("foo") | isnewer("foo.c", "foo")) {
  system("gcc -o foo foo.c");
}
```

(Beachtet, wie praktisch hier die short circuit evaluation ist: Wenn `foo` gar nicht existiert, wird `isnewer` gar nicht gefragt, ob es neuer ist als `foo.c`, und das ist gut, weil `isnewer` bestimmt nicht gutartig reagiert, wenn es nach dem Datum nicht existierender Dateien gefragt wird).

Wenn ihr analoges bei nichttrivialen Makefiles probiert, habt ihr im Nullkommanichts ein unglaubliches Spaghetti von ifs, das niemand mehr durchschaut – oder ihr schreibt euch geschickte

Funktionen, die aber gemeinsam mit den geeigneten Datenstrukturen letztlich wieder eine Art kleine deklarative Sprache bilden werden.

Problems

(41.1) TEX-Quelltext in der Datei onepage.tex wird durch das Kommando

```
tex onepage
```

in eine so genannte DVI-Datei onepage.dvi verwandelt (DVI steht für device independent, Geräteunabhängig). Will man sie drucken, will man in der Regel Postscript haben, das wiederum durch den Befehl

```
dvips onepage.dvi
```

in einer weiteren Datei onepage.ps erzeugt wird. Vielleicht will man jetzt ein (Bitmap-) Bild daraus machen. Dies geht mit dem Kommando

```
pstopnm -stdout onepage.ps > onepage.pnm
```

Nun sind pnm-Bilder nicht gepackt, und so hätte man zur Speicherung gerne png-Bilder. Diese Wandlung besorgt ein Programm pnmtopng:

```
pnmtopng onepage.pnm > onepage.png
```

Schreibt Pattern-Rules für GNU make für jeden Schritt. **(L)**

42. make II

Ein Template-Makefile:

```
CFLAGS += -Wall

.PHONY: all depend clean check install

SRCS = estr_test.c edelstring.c
OBJS = estr_test.o edelstring.o

all: estr_test

estr_test: $(OBJS)
  $(CC) -o $@ $(LDFLAGS) $(CFLAGS) $^

depend:
  makedepend -s "# makedepend depends on me"\
  -- $(CFLAGS) -- $(SRCS)

clean:
  rm -f $(OBJS) estr_test

check:
  @echo "No checks defined, sorry"

install:
  @echo "Nothing to install yet"
```

Es gibt einige Targets, die in allen Makefiles stehen sollten, zumindest `all` (alles bauen, was zu bauen ist) und `clean` (die Resultate von make aufräumen). Nett sind weiter `install` (die erzeugten Programme installieren), `check` (Funktionsfähigkeit der Programme prüfen), `depend` (s.o.), `dist` (baut ein Archiv mit allen zur Kompilation nötigen Dateien).

Wird make ohne Argumente aufgerufen, baut es das erste Target. Es ist in der Regel nicht verkehrt, „all“ als das erste Target im Makefile stehen zu haben.

Das geht noch schöner: Vgl. Makefile Conventions im make-info.

Weitere Automatisierung durch automake, autoconf.

Problems

statisch

dynamisch

shared object

(42.1) Besorgt euch ein Programm aus dem Netz (z.B. irgendwas von freshmeat[13] – wenn euch nichts besseres einfällt, nehmt GNU sed[14]) und seht euch an, wie das gebaut wird. In den meisten Fällen wird eine Datei namens INSTALL (oder ähnlich) beliegen, die euch zu den nötigen Dateien führt. Wenn euer Programm GNU autoconfig verwendet (ein Hinweis darauf ist, dass ihr aufgefordert werdet, „configure“ zu tippen), seht euch das configure-Skript an (aber nicht zu lange). Seht euch die (eventuell generierten) Makefiles an.

43. Verwendung von Bibliotheken I

Die bisher besprochenen Funktionen kommen aus der Standard-C-Bibliothek (unter Unix meist /usr/lib/libc.a oder /lib/libc.a.b.c.so), die zu jedem C-Programm dazugelinkt wird. Es gibt viele andere nützliche Bibliotheken. Ein Beispiel ist die glib[15] (nicht zu verwechseln mit der GNU C library glibc). Dokumentation dazu im Netz[16] oder mit `pinfo glib`. Dass das nicht `man glib` ist, liegt daran, dass die GNU-MacherInnen man-pages (berechtigterweise) etwas angestaubt fanden und ihre Dokumentation in der Regel in TEXinfo schreiben, was zwar besser ist, aber zu mindestens zwei unabhängigen Dokumentationssystemen in der Unix-Welt geführt hat, eben man und info.

Flags

Wenn man eine Bibliothek benutzt,

- muss der Linker wissen, welche Bibliotheken er woher dazulinken soll und
- manchmal auch der Compiler, woher Headerfiles kommen sollen.

Diese Information bekommen Compiler und Linker wieder aus Flags, etwa `-lglib`, um die glib zu linken. Für die glib gibt es ein Programm, das diese Flags ausgibt: `glib-config`. Ein glib-Programm kann folgendermaßen kompiliert werden:

```
gcc   glib_hash.c -o glib_hash\
  ‘glib-config --cflags‘ ‘glib-config --libs‘
```

Leider haben die meisten Bibliotheken keine so netten Programme wie glib-config. Um Flags (und vieles andere) in einer halbwegs systematischen Weise erraten zu können, gibt es das erwähnte GNU autoconfig.

Ab Version 2 verwendet die glib ein Programm pkg-config, dessen Sinn ist, dass nicht jedes Programm ein eigenes xxx-config-Programm mitbringen, sondern nur eine Beschreibungsdatei an einer geeigneten Stelle hinterlassen muss. Wenn ihr nur eine so moderne glib habt, müsst ihr die Aufrufe an glib-config durch `pkg-config --cflags glib` bzw. `pkg-config --libs glib` ersetzen. Die Programme hier sollten auch mit der glib2 kompilierbar sein.

Übrigens führt ein Flag wie `-lglib` unter Unix dazu, dass der Linker Dateien wie `libglib.a` oder `libglib.so` sucht – den Präfix lib impliziert der Linker ebenso wie die Extensionen. Unter Windows sieht das im Groben ähnlich aus, die Extensionen sind aber eher LIB oder DLL.

Der Unterschied zwischen Bibliotheken mit Extensionen .a und .so ist, dass erstere *statisch* gelinkt werden, d.h., dass der Code der verwendeten Funktionen fest ins fertige Programm aufgenommen wird, während letztere *dynamisch* gelinkt werden. Dabei steht die Extension .so für „*shared object*“, was gleich einen Vorteil andeutet: Die Bibliothek wird unter allen Programmen im System geteilt, sogar im Speicher – statisch gelinkter Code steht im Programm und braucht separat Platz sowohl auf der Platte als auch im RAM. Die analoge Windows-Extension DLL heißt übrigens als Dynamically Loaded Library. Für die aus einer dynamischen Bibliothek gelinkten Funktionen

[13] `http://www.freshmeat.net`

[14] `http://download.uni-hd.de/ftp/pub/gnu/sed/`

[15] `http://www.gtk.org/`

[16] `http://developer.gnome.org/doc/API/glib/index.html`

stehen nur Namen im Produkt des Linkers. Erst wenn das Programm aufgerufen wird, werden diese Namen durch die Einsprungpunkte der Funktionen im shared object/der DLL ersetzt.

Hash

associative array

Nachteil des dynamischen Linkens: Wenn die shared objects gelöscht werden oder nur die Binärdatei ohne Bibliothek auf ein anderes System kommt, kann das Programm nicht mehr laufen. Vor allem unter Windows kommt noch das Versionsproblem dazu – auch Bibliotheken entwickeln sich mit der Zeit, und mit dieser Entwicklung ändern sich z.B. Datentypen und Prototypen. Wenn ein Programm gegen eine andere Version des shared objects gelinkt wurde, kann es wieder zu Abstürzen und Fehlfunktionen kommen – die berühmten DLL-Probleme. Unter Unix kann sowas auch ein Problem sein, durch eine etwas geschicktere Verwaltung und ein schlaueres Binärformat aber bei weitem nicht so oft – allerdings hat man hier häufig Probleme, wenn man C++-Bibliotheken mit verschiedenen Compilern baut.

Ein Makefile-Fragment für Programme mit glib:

```
CFLAGS += -Wall ‘glib-config --cflags‘
LDFLAGS += ‘glib-config --libs‘
```

Exkurs: Das lm-Problem

Aus historischen Gründen sind die mathematischen Funktionen häufig nicht in der Standardbibliothek selbst enhalten, sondern in einer Bibliothek namens libm. Das führt dazu, dass ein Programm wie

```
#include <math.h>
int main(void) { return (int)tan(M_PI); }
```

beim Linken Fehlermeldungen wie

```
/tmp/ccKcAW4r.o: In function ‘main’:
/tmp/ccKcAW4r.o(.text+0x16): undefined reference
to ‘tan’
collect2: ld returned 1 exit status
```

liefert. Abhilfe schafft Linken mit `-lm` (im Makefile: `LDFLAGS += -lm`).

44. Hashes

Pythons Dictionaries heißen in anderen Sprachen *Hash*es oder *associative array*s. Sie sind kein Bestandteil von C oder der Standardbibliothek, werden aber von der glib implementiert.

Ein einfaches Python-Programm wie

```
myhash = {"Hallo":1,
  "Komisch":"Wert"}
print myhash["Hallo"]
print myhash.get("Hallo2", 0)
print myhash["Komisch"]
```

mit der Ausgabe

```
1
0
Wert
```

nimmt in C schon recht beängstigende Ausmaße an.

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

int main(void)
{
  int aValue = 1;
  GHashTable *myhash = g_hash_table_new(g_str_hash, g_str_equal);
```

```
  g_hash_table_insert(myhash, "Hallo", &aValue);
  g_hash_table_insert(myhash, "Komisch", "Wert");
  printf("%d\n", *(int*)g_hash_table_lookup(myhash, "Hallo"));
  printf("%x\n", g_hash_table_lookup(myhash, "Hallo2"));
  printf("%s\n", (char*)g_hash_table_lookup(myhash, "Komisch"));
  return 0;
}
```

static

Von der glib kommt hier

- der *opake* (d.h. undurchsichtige, die Benutzerin weiß nicht, woraus er aufgebaut ist) Typ `GHashTable`, und
- die Funktionen `g_hash_table_new`, entspricht etwa `myhash = {}`
- `g_hash_table_insert`, enspricht etwa `myhash[key] = value`
- `g_hash_table_lookup`, entspricht etwa `myhash[key]`

Dabei kann `g_hash_table_lookup` natürlich keinen KeyError werfen, wenn `key` nicht im Hash ist. Stattdessen signalisiert sie die Fehlerbedingung auf C-Art durch Rückgabe von `NULL`.

Genau genommen haben wir noch die von der Bibliothek selbst definierten Callbacks `g_str_hash` und `g_str_equal` verwendet. Im Groben sagen sie, dass unser Hash tatsächlich Strings und nicht was anderes als Indizes verwendt.

Eine offensichtliche Anwendung für diese Hashes ist mal wieder das Wörterzählen. Im Anhang dieser Seite befindet sich Quellcode für ein Programm, das Wörter in der der Standardeingabe zählt. Darin befindet sich eine neue Funktion, `strdup`. Sie nimmt einen String als Argument und kopiert ihn in frisch allozierten Speicher. Wir brauchen das hier, weil die getNextWord-Funktion die Wörter in einem statischen Speicher zurückgibt – mehr dazu auf der nächsten Folie.

— Dateien zu diesem Abschnitt in der HTML-Version—

Problems

(44.1)* Holt euch die an diese Seite angehängten Dateien, versteht sie, kompiliert das Programm und probiert es aus. Schreibt ein Äquivalent in Python. Vergleicht Codelänge und Geschwindigkeit (und Einfachheit der Implementation). **(L)**

45. Scoping und Lebensdauer

Variablen in C haben unterschiedliche Sichtbarkeit (*Scope*) und Lebensdauer.

Der Scope ist weitgehend wie in (neuerem) Python – eine Variable ist sichtbar ab ihrer Deklaration bis zum Ende des Blocks, in dem sie deklariert wurde (für lokale Variablen) oder bis zum Ende der Datei (für globale Variablen). Dabei ist ein Block in C einfach das Material zwischen einer öffnenden und einer schließenden geschweiften Klammer. Mithin ist es durchaus legal, am Anfang etwa eines Schleifenkörpers Variablen zu definieren – sie existieren dann nur innerhalb der Schleife.

Normale Variablen leben auf dem Stack und daher so lange, wie der Block, in dem sie definiert sind, läuft. Globale Variablen sowie als *static* deklarierte lokale Variablen leben bis zum Programmende.

Als `static` deklarierte Funktionen und globale Variablen sind nur in dem Modul sichtbar, in dem sie definiert wurden (d.h. ihre Namen tauchen in der Ausgabe von `nm` nicht auf). Die eine Verwendung des `static`-Schlüsselworts hat mit der anderen recht wenig zu tun.

auto

type qualifier

Code	Sichtbar	Leben
`int a=5;`	a=5	a,c
`void f(void)`		
`{ int b=7;`	a=5, b=7	a,b,c
`  static int c=8;`	a=5, b=7, c=8	a,b,c
`}`		
`int main(void)`		
`{ int a=2;`	a=2	a,c
`  int b=0;`	a=2, b=0	a,b,c
`  f();`	a=2, b=0	a,b,c
`}`		

Fig. 12

(cf. Fig. 12)

In `f` ist das globale `a` sichtbar, *nicht* das aus `main`.

Weitere Speicherklassen:

- `extern` Variable ist hier nur deklariert, die Definition ist anderswo – siehe Module
- `register` Nicht mehr verwenden
- `auto` Tut nichts, unverzierte Variablen haben die Speicherklasse *auto*, ist also überflüssig.

Ähnlich verwendet werden die *type qualifier*s

- `volatile` Variable kann von außen geändert werden (z.B. Temperatursensor einer Hardware, shared memory...)
- `const` Variable darf nicht geändert werden. Der Const-Qualifier ist vor allem im Zusammenhang mit Prototypen von Funktionen interessant, er erlaubt die Definition von „read-only"-Interfaces. Außerdem kann der Compiler consts besser optimieren.
- `restrict` Ist ein Tipp an den Compiler, dass der Speicher, auf den ein restricted pointer zeigt, nur durch diesen Pointer verändert wird

Die type qualifier, allen voran `restrict`, tendieren zu leichter Esoterik, zumal, wenn sie in Kombination und in komplexeren Deklarationen auftauchen. Nach

```
const char * foo;
char const * bar;
```

darf man etwa an `foo` herumschrauben, nicht jedoch an `*foo`, während man an `bar` nicht herumschrauben darf (fast, als wäre es als `char bar[]` deklariert), wohl aber an `*bar`.

Dennoch lohnt es sich, `const` in Prototypen zu verwenden, wann immer möglich, weil es saubere Strukturen fördert und einige Programmierfehler zu vermeiden hilft.

Der `volatile`-modifier ist vor allem bei eher nahe am System liegenden Programmieraufgaben praktisch und bewahrt den Compiler vor Optimierungen, die sich darauf verlassen, dass er den Speicher unter Kontrolle hat. Auch im Zusammenhang mit Threads und Signalen braucht man unter umständen ein volatile.

Der `restrict`-modifier kann von Compilern fröhlich ignoriert werden. Wenn euch restrict unsympathisch ist und ihr auf das letzte Prozent Geschwindigkeit verzichten könnt, könnt ihr den Compilern folgen.

Beispiele dafür:

```
#include <stdio.h>

void countCalls(const char * const message)
/* message ist const, wir dürfen also den Pointer nicht
verändern, und es zeigt auf const char, wir dürfen also
auch die Nachricht nicht verändern. */
{
  static int counter=0; /* wird nur ganz am Anfang initialisiert,
```

```
   bleibt dazwischen erhalten */

  counter++;
  printf("%s %d\n", message, counter);
  /* Das würde einen Fehler geben: message = "Oops";
     und das auch: *message = 0;
  */
}

int main(void)
{
  countCalls("erster Aufruf");
  countCalls("zweiter Aufruf");
  countCalls("upzinkta Aufruf");
  return 0;
}
```

Die Ausgabe davon:

```
erster Aufruf 1
zweiter Aufruf 2
upzinkta Aufruf 3
```

46. Fallstudie: Liste

Wir wollen etwas wie Python-Listen auch in C haben. Dazu werden wir ein Modul schreiben, das einen Teil der Funktionalität – wenigstens mal für Strings – bietet.

Die Implementation einer Liste ist pädagogisch gemeint – Implementationen von Listen in C (oder fast jeder anderen Sprache) gibt es wie Sand am Meer. Gemäß der Golden Rule „Whenever possible, steal code“ sollte niemand mehr wirklich Listen von Grund auf programmieren. Sinn der Übung ist die Demonstration, wie ein Modul geschrieben wird und auch ein kleiner Einblick in die Liste als Datenstruktur.

Übrigens kann man die Python-Bibliothek und damit auch deren Listen auch in eigenen C-Programmen verwenden. Wie, werden wir später sehen.

Die Schnittstelle zu einem Modul, das eine Liste von Strings implementiert:

- List: der Listentyp
- `List *list_new(void);` legt eine neue Liste an
- `int list_free(List *l);` gibt eine Liste frei
- `int list_append(List *l, char *item);` hängt `item` an `l` an
- `char *list_getAt(List *l, int index);` gibt den String an Positon `index` zurück
- `int list_deleteAt(List *l, int index);` löscht das Element bei `index`
- `int list_insertBefore(List *l, int ind, char *it);` fügt `it` vor der Position `ind` ein
- `list_len(List *l);` gibt die Zahl der Elemente der Liste zurück
- `list_sort(List *l);` sortiert die Liste

Wir haben die Funktionen wieder durch Verben bezeichnet – wie schon in Python sollen auch C-Funktionen in erster Linie etwas tun, und wieder haben wir alle Namen unseres Moduls mit dem Präfix `list_` gekennzeichnet – schon `free` statt `list_free` würde mit der Funktion `free` aus der Standardbibliothek kollidieren.

Wir versuchen, so viel von den Prinzipien der Objektorientierung wie möglich nach C zu retten. Trennung von Implementation und Schnittstelle: Headerfile definiert Methoden, die unabhängig von der Implementierung sind.

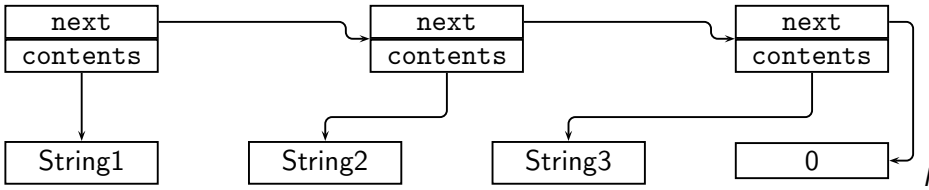


Fig. 13

opak

Module lassen sich einzeln warten, Fehler bleiben lokalisiert, „one code line – one programmer“ auch in großen Projekten, Wiederverwendbarkeit.

Als Beispiel für die Modularisierung größerer Projekte kann die glib dienen, in der arrays, hashes, nodes usf. alle ihre eigenen Module haben. In diesem Fall haben die einzelnen Module nicht eigene Headerdateien – sie sind alle in der zentralen Headerdatei glib.h vereinigt.

Im Headerfile: Deklaration exportierter Typen, Prototypen exportierter Funktionen.

In der Quelldatei: Alles, was nicht unbedingt von außen sichtbar sein muss, ist `static`.

47. Listen: Implementation I

Listen sind zunächst abstrakte Konstrukte. Die *Implementation* in C sieht meistens irgendwie so aus:

(cf. Fig. 13)

Also: Ein Listenelement *(node)* sieht so aus:

```
typedef struct Node_s {
  struct Node_s *next;
  char *cont;
} Node;
```

Die Verwendung von `struct Node_s *` in der Definition von `Node` ist wegen der rekursiven Definition notwendig: Das `typedef` hat der Compiler noch nicht verdaut, wenn wir den Zeiger auf das nächste Listenelement deklarieren wollen.

Für die ganze Liste brauchen wir:

```
typedef struct List_s {
  Node *first; /* Erstes Element */
  Node *last;  /* Letztes Element */
} List;
```

Die Definition von `List` kommt in den Header (Clients sollen den Inhalt aber nicht kennen – `List` ist *opak*). Dass wir die Liste mit `Node`s implementieren (und nicht z.B. ein Array dazu nehmen), soll der Benutzer nicht wissen. Leider muss die Definition des Typen Node trotzdem in den Header, weil sie zur Definition von List nötig ist.

Weiterführend: Tatsächlich könnten wir darauf bestehen, im Headerfile nichts über den Inhalt von `Node` zu verraten. C würde die Deklaration

```
struct Node;
typedef struct List_s {
  struct Node *first; /* Erstes Element */
  struct Node *last;  /* Letztes Element */
} List;
```

friedlich über sich ergehen lassen und euch erlauben, dass ihr `struct Node` erst dort erklärt, wo ihr wirklich auf seine Elemente zugreifen wollt (hier also nicht im Header, sondern im C-Quellcode des Listenmoduls). Sowas ist ein Beispiel für forward declarations, Deklarationen, in denen man nur verspricht, später mehr zu verraten.

Wir wollen hier der Einfachheit halber darauf verzeichten.

Neue Liste erzeugen:

```
List *list_new(void)
{
  List *l = malloc(sizeof(List));

  if (l) {
    l->first = NULL;
    l->last = NULL;
  }
  return l;
}
```

Wenn kein Platz ist, wird `NULL` zurückgegeben.

Die Logik hier ist, dass `malloc` bereits `NULL` zurückgibt, wenn der Speicher nicht alloziert werden kann. Bedingt auf den Erfolg von `malloc` wird dann die Liste initialisiert.

Weil die Funktion kurz ist, ist das hier guter Stil. Ist der Funktionskörper länger, ist es empfehlenswert, bereits beim Auftreten des Fehlers aufzugeben (aber Vorsicht mit memory leaks!):

```
if (!(l = malloc(sizeof(Liste))) {
  return NULL;
}
```

Diese Funktion ist gewissermaßen der Konstruktor unserer List-„Klasse“.

48. Listen: Implementation II

Einen neuen Node erzeugen:

```
static Node *getNodeFromContent(char *item)
{
  char *buf = strdup(item);
  Node *newNode;

  if (!buf) {
    return NULL;
  }
  if (!(newNode = malloc(sizeof(Node)))) {
    free(buf); /* cave memory leak! */
    return NULL;
  }
  newNode->next = NULL;
  newNode->cont = buf;
  return newNode;
}
```

Der von der Benutzerin übergebene String wird in einen eigenen Speicher kopiert, weil man nicht weiß, wie lange der Speicher, auf den `item` zeigt, uns gehört (oder ob das aufrufende Programm daran rumpfuscht).

`getNodeFromContent` ist `static`: Der Name ist von außen nicht sichtbar.

Das ist eine Design-Entscheidung. Wir wollen grundsätzlich nicht, dass die Benutzerin weiß, wie wir unsere Liste implementieren, dass etwa jeder gespeicherte String einen eigenen Knoten hat.

Wir hätten uns hier auch anders entscheiden können: Wenn Benutzer Knoten haben können, sind beispielsweise effiziente `insert_after`-Funktionen möglich, die den Vorgängerknoten als Argument nehmen.

Abwägungen dieser Art sind im Software Engineering nicht selten. Man möchte zwar fast immer so weit wie möglich von der Implementation abstrahieren (da sich diese schnell ändern

kann) – man bezahlt dafür aber mit Schnittstellen, die keine Rücksicht auf die Bedürfnisse und Beschränkungen der konkreten Implementation nehmen können. Die Abwägung der Ansprüche Abstraktheit und Effizienz gehört mit zu den Entscheidungen, die das Design eines Programms ausmachen.

Im Namen `getNodeFromContent` fehlt der Präfix `list_`. Das können wir uns hier leisten, weil die Funktion static ist und wir somit nur unseren eigenen „Namensraum verschmutzen“ – den des nach außen opaken Moduls eben.

Freigeben eines Nodes:

```
static void freeNode(Node *n)
{
  free(n->cont);
  free(n);
}
```

Diese Funktion ist gefährlich, weil sie davon ausgeht, dass der Node schon aus der Liste ausgehängt ist und das nicht prüft – wird aber ein Node, der noch in der Liste ist, freigegeben, sind wüste und praktisch nicht zu findende Fehler quasi sicher. In einer einfach verketteten Liste ist eine Prüfung, ob der Node ausgehängt ist oder nicht aber relativ aufwändig (man muss die Liste komplett nach dem Node durchsuchen), und so verlassen wir uns darauf, dass die Funktion richtig verwendet wird – da sie static ist, kann das ohnehin nur aus unserem Modul heraus passieren.

Anhängen eines Eintrags:

```
int list_append(List *l, char *item)
{
  Node *newNode;

  if (!(newNode = getNodeFromContent(item))) {
    return -1;
  }
  if (!l->last) { /* Liste leer */
    l->last = l->first = newNode;
  } else {
    l->last->next = newNode;
    l->last = newNode;
  }
  return 0;
}
```

Hier wird eine negative Zahl zurückgegeben, wenn dabei was schief geht und 0 sonst. Auch das ist Konvention aus der C-Bibliothek.

49. Listen: Implementation III

wrapper

Die Länge einer Liste bestimmen:

```
int list_len(List *l)
{
  int count=0;
  Node *cur = l->first;

  while (cur) {
    count++;
    cur = cur->next;
  }
  return count;
}
```

Die Liste komplett freigeben:

```
void list_free(List *l)
{
  Node *cur = l->first, *next;

  while (cur) {
    free(cur->cont);
    next = cur->next;
    free(cur);
    cur = next;
  }
  free(l);
}
```

Der Tanz mit `cur` und `next` ist nötig, weil auf `gefreeten` Speicher nicht mehr zugegriffen werden darf.

Eine Hilfsfunktion (`static`!), die den `Node` zu einem Listenindex zurückgibt.

```
static Node* getNodeAt(List *l, int index)
{
  Node *cur=l->first;

  for (; cur && index; index--) {
    cur = cur->next;
  }
  return cur;
}
```

Der *wrapper* für den Zugriff über einen Index:

```
char *list_getAt(List *l, int index)
{
  Node *n=getNodeAt(l, index);

  if (n) {
    return n->cont;
  }
  return NULL;
}
```

In dieser Funktion haben wir zwei `return`-Statements, und der Fehlerfall wird durch das „durchfallen“ der Ausführung zum letzten `return` erzeugt. Das ist nicht verkehrt, aber man sollte, wenn man mehrere (Nichtfehler-)returns in einer Funktion hat, aufpassen, dass die Struktur der

Funktion noch klar bleibt. Lange Funktionen mit vielen returns drin tendieren zur Unverständlichkeit.

Die historische Vorschrift, nach der eine Funktion nur an einer Stelle verlassen werden kann, ist so jedenfalls unsinnig. Eine gute Diskussion dieses Punkts[17] ist auf ist am Wiki von Cunningham and Cunningham zu finden.

50. Listen: Implementation IV

```
int list_deleteAt(List *l, int index)
{
  Node *toDel;

  if (index==0) {
    toDel = l->first;
    l->first = toDel->next;
    if (!l->first) {
      l->last = NULL;
    }
  } else {
    Node *prev=getNodeAt(l, index-1);
    if (!prev) {
      return -1;
    }
    if (!(toDel = prev->next)) {
      return -1;
    }
    prev->next = toDel->next;
    if (!prev->next) {
      l->last = prev;
    }
  }
  freeNode(toDel);
  return 0;
}
```

Golden Rule: If you have too many special cases, you are doing it wrong.

Außerdem enthält dieser Code einen Fehler, den wir später mit einem Debugger finden werden.

In diesem Fall könnte man an Sentinel Values für den Anfang und das Ende denken, um die Spezialbehandlungen überflüssig zu machen. Das komplizierte Handling mit dem Vorgängerknoten `prev` ließe sich durch *doppelte Verkettung* umgehen – dafür aber würden wir mit ansonsten erhöhter Komplexität bezahlen.

```
int list_insertBefore(List *l, char *item,
  int index)
{
  Node *new = getNodeFromContent(item);

  if (index>0) { /* Somewhere within the list */
    Node *cur=getNodeAt(l, index-1);
    if (cur) {
      new->next = cur->next; cur->next = new;
    } else { freeNode(new); return -1; }
  }
```

[17] `http://c2.com/cgi/wiki?SingleFunctionExitPoint`

```
  else if (index==0) { /* Insert at start */
    if (!l->first) l->last = new;
    new->next = l->first; l->first = new;
  } else { freeNode(new); return -1; }
  return 0;
}
```

Laufzeitbewertung

Laufzeiten werden in der Informatik üblicherweise in O-Notation in Abhängigkeit von der Menge der Daten gegeben. $O(n)$ bedeutet, dass eine Funktion für große n doppelt so lange läuft, wenn sie doppelt so viele Daten zu bearbeiten hat, bei $O(n^2)$ ist das Verhältnis 4 : 1, bei $O(\exp(n))$ ist es grob 8 : 1. Optimal sind $O(1)$-Funktionen: Ihre Laufzeit ist unabhängig von der Datenmenge.

Hintergrund der Notation ist der Grenzwertbegriff. Eine Funktion $f(n)$ ist $O(h(n))$ – dabei soll $h(n)$ eine einfachere Funktion sein –, wenn

$$\lim_{n\to\infty} \frac{f(n)}{h(n)} = \text{const},$$

das Verhältnis der beiden Funktionen also für eine wachsendes n konstant wird. Für uns ist der Wert der Funktion die Zahl der „Operationen“, die zur Berechnung der Funktion ausgeführt werden müssen (dabei vernachlässigt man üblicherweise, dass verschiedene Operationen verschieden lang dauern, weil das ohnehin stark von der verwendeten Maschine abhängt), während n die Datenmenge repräsentiert; bei der Liste wäre n die Zahl der Listenelemente.

Beispiel: $f(n) = 7n^2 + 4n + 1400$. Für kleine n dominiert natürlich der konstante Anteil, schon für $n = 6$ hat aber der quadratische Term überholt und dominiert das Geschehen. $f(n)/n^2$ geht gegen die Konstante 7, wir haben also $O(n^2)$.

Funktionen, die $O(n^i)$ sind, haben *polynomiale Laufzeit*, was für theoretische InformatikerInnen „gut“ ist – schlimm sind dort *exponentielle Laufzeiten* der Art $O(\exp(n))$. Probleme, für die nur Algorithmen mit exponentieller Laufzeit bekannt sind, heißen NP-hart.

In der Realität ist schon $O(n^3)$ ein guter Grund, nach besseren Methoden Ausschau zu halten, es sei denn, wir wissen, dass n klein ist. In der Tat bemühen sich viele NumerikerInnen, bessere Alternativen oder Näherungen für $O(n^2)$-Algorithmen zu finden, beispielsweise bei N-Körpersimulationen. Für häufig verwendete Operationen, bei uns etwa das Anhängen an die Liste oder den Zugriff auf ein Element, will man sogar $O(1)$ oder wenigstens $O(\log(n))$.

Für reale Programme ist es nicht immer ganz einfach, die Komplexität abzuschätzen und vor allem, zu sehen, wo die ganze Rechenzeit hinläuft, selbst wenn man die Komplexität der einzelnen Algorithmen kennt. Der Computer kann dabei helfen, und zwar mit Programmen, die Profiler heißen. Wir kommen dazu, sobald wir wissen, was wir mit dieser Liste tun wollen.

This list sucks...

Die meisten Operationen, die diese Liste zur Verfügung stellt, sind $O(n)$, weil die Liste immer durchsucht werden muss, um zu einem bestimmten Index zu kommen.

Zu wenig Fehlersicherheit.

Zu viel `malloc` (Verwaltungsoverhead) – Abhilfe könnte z.B. sein, in jedem Node z.B. 10k Daten oder 50 Strings zu speichern.

Wie gesagt: Bessere Listenimplementationen gibt es überall (z.B. in der glib – allerdings haben diese Listen ihre eigenen Tücken, vor allem, wenn man Python gewohnt ist).

— Dateien zu diesem Abschnitt in der HTML-Version—

51. Debugging I

Debugging

Coredumps

Gegeben sei folgendes Programm:

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

int main(int argc, char **argv)
{
  List *l=list_new();
  char *val;

  argv++;
  while (*argv) {
    list_append(l, *argv++);
  }
  list_sort(l);
  while ((val=list_getAt(l, 0))) {
    printf("%s\n", val);
    list_deleteAt(l, 0);
  }
  list_deleteAt(l, 0);
  list_free(l);
  return 0;
}
```

Wir müssen das mit dem Listenmodul linken, als Regeln für Make bieten sich an:

```
%: %.o
  $(CC) -o $@ $(LDFLAGS) $(CFLAGS) $^

crash: crash.o liste.o
```

(das Programm soll in crash.c stehen).

Beim Ausführen des Programms passiert folgendes:

```
examples> crash dab bad abd
abd
bad
dab
Segmentation fault
Exit 139
```

– das Programm versucht, auf Speicher zuzugreifen, der ihm nicht gehört.

Will man solche Fehler finden (*Debugging*), helfen häufig Programme, die Debugger heißen.

Man hört oft, das Wort „Bug“ (Käfer, Wanze) für einen Fehler in Computerprogrammen komme daher, dass in einem der ersten Rechner ein fast unfindbarer Fehler durch eine wirkliche Wanze oder Motte, die es sich in einem seiner Relais bequem gemacht hatte, verursacht wurde. Das ist so nicht zutreffend, vgl. den ensprechenden Eintrag im Jargon File[18].

Will man nicht Maschinensprache debuggen, muss der Debugger wissen, welcher Code aus welcher Quellzeile kommt. Diese Information schreibt der Compiler nur auf Anfrage, und zwar mit dem –g-Flag.

Für Nicht-Unix-Compiler mag das anders aussehen, aber auch diese schreiben Debugging-Informationen in der Regel nicht automatisch, schon, weil das eine endlose Platzverschwendung wäre.

Benutzt man Make wie hier empfohlen, fügt man einfach -g zu den CFLAGS und lässt `make clean; make` laufen.

Wenn ein Programm unter Unix abstürzt, schreibt es normalerweise eine Datei `core`, in der steht, wie der Speicher des Programms am Ende aussah.

Tatsächlich schalten viele Administratoren und Distributionen das aus, weil die meisten Menschen heute nicht mehr viel mit diesen *Coredumps* anfangen können und sie nur Platz auf der Platte

[18] `http://www.jargon.net/jargonfile/a/aliasingbug.html`

verbrauchen. Wenn nach einem Segfault oder ähnlichem kein core auf der Platte liegt, kann man durch das Kommando `ulimit -c unlimited` (o.ä., je nach Shell; es kann sein, dass ihr unlimited nicht dürft, probiert es dann mit einer großen Zahl, vielleicht 4000000) dafür sorgen, dass Coredumps geschrieben werden.

Nach einem Coredump kann gdb sagen, wo der Absturz passiert ist:

```
examples > gdb crash core
GNU gdb 5.1
...
#0  in list_deleteAt (l=0x8049e90, index=0) at liste.c:110
110         l->first = toDel->next;
(gdb)
```

und auch, wie das Programm zur Absturzstelle gekommen ist:

```
(gdb) where
#0  in list_deleteAt (l=0x8049e90, index=0) at liste.c:110
#1  in main (argc=4, argv=0xbffff934) at crash.c:18
#2  in __libc_start_main (main=0x8048540 <main>, argc=4,
    ubp_av=0xbffff924, init=0x8048374 <_init>,...
    at ../sysdeps/generic/libc-start.c:129
```

So wissen wir immerhin mal, dass der Absturz in Zeile 110 in list.c in der Funktion `list_deleteAt` passiert ist, und dass der Aufruf aus aus der Zeile 18 in crash.c kam – das ist der letzte Aufruf. Man überlegt sich leicht, dass die Liste zu diesem Zeitpunkt leer ist, und das sollte in der Regel schon reichen, um den Fehler zu finden.

Aber der gdb kann noch viel mehr. So kann man sich z.B. zunächst die Umgebung der Absturzstelle ansehen:

```
(gdb) list
105     {
106       Node *toDel;
107
108       if (index==0) {
109         toDel = l->first;
110         l->first = toDel->next;
111         if (!l->first) {
112           l->last = NULL;
113         }
114       } else {
```

und sich Variablen ansehen:

```
(gdb) print index
$1 = 0
(gdb) print toDel
$2 = (Node *) 0x0
(gdb) print l->first
$3 = (Node *) 0x0
```

All das passiert vorläufig in der Funktion, in der der Absturz passiert ist. Wir können aber auch am Stack eine Funktion „raufgehen“:

```
(gdb) up
#1  0x080485e5 in main (argc=4, argv=0xbffff934) at crash.c:18
18              list_deleteAt(l, 0);
(gdb) list
13              list_sort(l);
14              while ((val=list_getAt(l, 0))) {
15                      printf("%s\n", val);
16                      list_deleteAt(l, 0);
17              }
18              list_deleteAt(l, 0);
19              list_free(l);
20              return 0;
21      }
```

und sich Variablen in diesem Kontext ansehen:

```
(gdb) print argc
$4 = 4
(gdb) print *argv
```

```
$5 = 0x0
```

– dass `argv` ein Nullpointer ist, zeigt uns wieder, dass die erste Schleife durchgelaufen ist.

Auf diese Weise lässt sich häufig rekonstruieren, wie es zu einem Absturz kam.

Der gekonnte Einsatz von Debuggern ist letztlich eine Kunst, die man durch viel Erfahrung lernt – und manchmal ist es besser, einfach ein printf an eine strategisch gute Position zu legen.

52. Debugging II

Debugger können auch laufende Programme beobachten und sie an Stellen (*Breakpoints*) unterbrechen, an denen man nachsehen möchte, was passiert:

```
examples> gdb crash
GNU gdb 5.1
...
(gdb) set args dab bad abd
(gdb) break list_deleteAt
Breakpoint 1 at 0x8048892: file liste.c, line 108.
(gdb) run
Starting program: examples/crash dab bad abd
abd

Breakpoint 1, list_deleteAt (l=0x8049e90, index=0)
at liste.c:108
108       if (index==0) {
(gdb)
```

An dieser Stelle kann man sich „in Zeitlupe“ ansehen, was passiert:

```
(gdb) step
109         toDel = l->first;
(gdb) print toDel
$1 = (Node *) 0x400164a8
(gdb) step
110         l->first = toDel->next;
(gdb) print toDel->next
$2 = (struct Node_s *) 0x8049ed0
(gdb) print toDel->next->cont
$3 = 0x8049ec0 "bad"
```

Insbesondere beim Debuggen können grafische Benutzeroberflächen hilfreich sein. Ein besonders nettes Frontend zum gdb heißt ddd:

(cf. Fig. 14)

Entsprechende Programme existieren auch unter Windows oder MacOS – häufig bereits in IDEs integriert. Auch wenn manche Dinge anders aussehen, die prinzipiellen Elemente sind ähnlich. Coredumps und das damit verbundene „post-mortem-debugging“ sind allerdings außerhalb der Unix-Welt nicht so üblich.

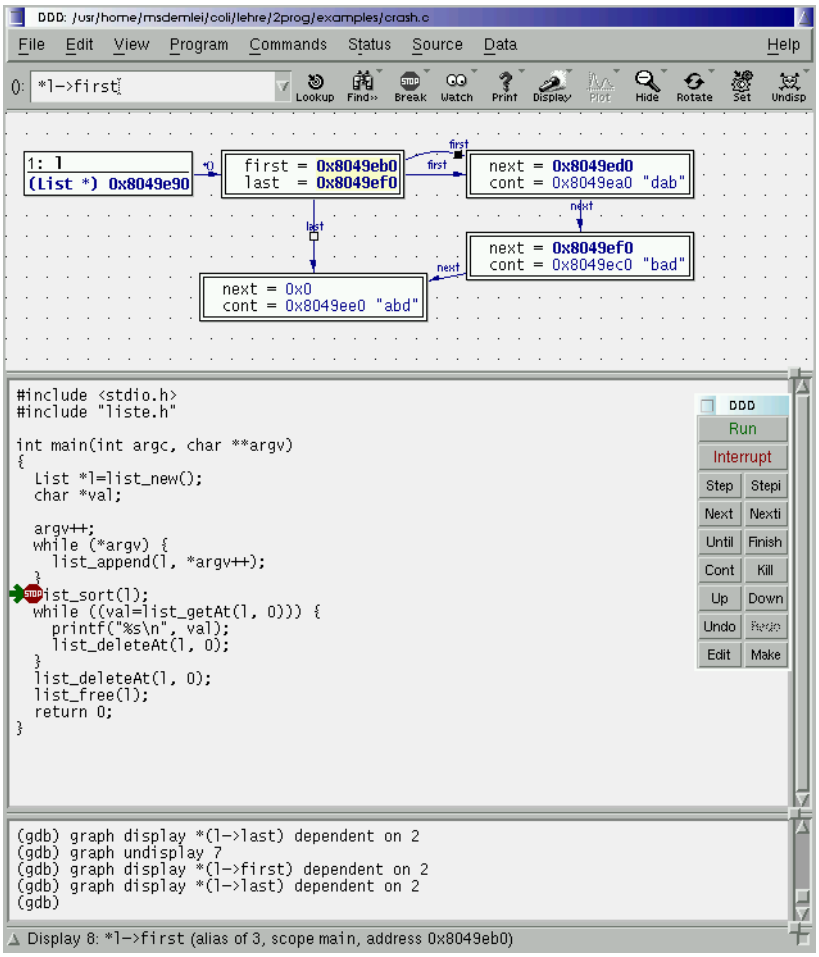


Fig. 14

53. Listen: Sortieren I

Die meisten guten Sortieralgorithmen gehen davon aus, dass man wahlfrei auf alle Elemente der zu sortierenden Datenstruktur zugreifen kann. Für Listen ist das nicht der Fall. Wir wollen dennoch unsere Liste direkt sortieren und implementieren dazu ein Selection Sort:

Idee des Selection Sort ist, einfach immer das kleinste Element aus der Menge der noch nicht sortierten Elemente zu suchen und es an das Ende es bereits sortierten Teils der Liste zu setzen. Letztere Operation ist für Listen sehr einfach, erstere auch nicht schwieriger als bei Arrays.

```
static Node *findAndRemSmallest(Node **toSortPtr)
{
  Node *smallest=*toSortPtr,
    **smallestChaining=toSortPtr;
  Node *curNode=*toSortPtr,
    **curChaining=toSortPtr;

  while (curNode) {
    if (strcoll(curNode->cont, smallest->cont)<0) {
      smallest = curNode;
      smallestChaining = curChaining;
    }
    curChaining = &(curNode->next);
    curNode = curNode->next;
  }
  *smallestChaining = smallest->next;
```

```
  return smallest;
}
void list_sort(List *l) /* Selection Sort */
{
  Node **curElPtr=&(l->first);
  Node *smallest=NULL;

  while (*curElPtr) {
    smallest = findAndRemSmallest(curElPtr);
    smallest->next = *curElPtr;
    *curElPtr = smallest;
    curElPtr = &(smallest->next);
  }
  l->last = smallest;
}
```

Wir haben unser Sortierproblem in zwei Funktionen unterteilt: Einerseits `findAndRemSmallest`, das einfach beim übergebenen Node anfängt, die Liste nach dem kleinsten Element zu druchsuchen. Dazu muss es natürlich immer bis zum Ende der Liste laufen. Es führt dabei Buch über den *Verweis* auf das kleinste Element, weil es das kleineste Element aus der Liste aushängen soll. Dafür muss man bei einfach verketteten Liste eben den Vorgänger kennen – doppelt verkettete Listen (es gibt jeweils noch einen Zeiger auf den Vorgänger) haben dieses Problem natürlich nicht.

Schließlich gibt `findAndRemSmallest` einen Zeiger auf das – mittlerweile nicht mehr in der Liste befindliche – kleinste Element zurück. In diesem Moment ist das die einzige Referenz auf den Node, wir sollten sie also nicht verlieren. Die Funktion `list_sort` nun hängt dieses Element an die richtige Stelle, nämlich an das Ende der schon sortierten Daten. Das tut es ein wenig raffiniert, nämlich, indem es jeweils den `next`-Pointer des Vorgängers dieses Zielelements speichert (im `Node **curElPtr`). Am Anfang ist dies ein Pointer auf `l->first`, danach jeweils ein Pointer auf das `next`-Feld des zuletzt bearbeiteten Knotens. Auch dies ist vor allem nötig, weil wir in der doch recht simplen einfach verketteten Liste arbeiten und könnte mit doppelter Verkettung etwas durchsichtiger gelöst werden.

Das Sortieren wird fortgesetzt, bis das Ende der Liste erreicht ist – die Funktion merkt das, in dem sie nachsieht, ob `curElPtr` auf einen Nullpointer zeigt. Dieser Nullpointer ist dann das `next`-Feld des aktuellen Elements, und dies war gerade unser Endemarker.

Die verwendete Vergleichsfunktion `strcoll` übrigens vergleicht zwei Strings gemäß den Regeln des augenblicklichen Locales. Wer das mit dem de_DE-Locale macht, kriegt die Umlaute an der richtigen Stelle einsortiert.

Mögliche Anwendung:

```
int main(void)
{
  FILE *inF=fopen("test.txt", "r");
  List *words=list_new();
  char *w;

  setlocale(LC_ALL, "");
  while ((w=getNextWord(inF))) {
    list_append(words, w);
  }
  list_sort(words);
  printList(words);
  return 0;
}
```

Dabei soll `getNextWord` das nächste Wort aus der Datei oder NULL am Dateiende zurückgeben. Eine mögliche Implementation (die überlange Wörter abschneidet) wäre

```
char *getNextWord(FILE *src)
{
  static char buf[TOKENS_WORDLEN];
  char *cp=buf;
  int ch;

  do {
    ch = fgetc(src);
  } while (!isalpha(ch) && ch!=EOF);
  if (ch==EOF) {
    return NULL;
  }
  while (isalpha(ch) && ch!=EOF &&
    cp-buf<TOKENS_WORDLEN-1) {
    *cp++ = ch;
    ch = fgetc(src);
  }
  *cp = 0;
  return buf;
}
```

– diesen Code gab es auf einer der vorigen Folien als tokens.c.

Die Funktion `printList` soll einfach die komplette Liste ausgeben und ist in einer naiven Implementation trivial.

Man kann das Programm mit einem Text von 6200 Wörtern ausprobieren; auf meiner Maschine ergibt sich eine Laufzeit von rund 4 Sekunden. Das ist ziemlich viel.

54. Listen: Sortieren II

Unser Sortieralgorithmus war ziemlich langsam. Es geht auch viel schneller, etwa mit folgendem Code:

```
static Node **getNodeArr(List *l, int *arrLen)
{
  Node **nodeArr, **arrEnt;
  Node *cur=l->first;

  *arrLen = list_len(l);
  if (!(nodeArr =malloc((*arrLen+1)*sizeof(Node*)))) {
    return NULL;
  }
  for (arrEnt=nodeArr; cur; cur=cur->next) {
    *arrEnt++ = cur;
  }
  return nodeArr;
}

static int nodeCompare(const void *a, const void *b)
{
  Node *node1=*(Node**)a, *node2=*(Node**)b;

  return strcoll(node1->cont, node2->cont);
}
```

Zwei Hilfsfunktionen: `nodeCompare` vergleicht die Strings in zwei Node-Pointern, `getNodeArr` füllt ein Array mit allen Node-Pointern der Liste – dieses Array kann mit leistungsfähigen Sortieralgorithmen sortiert werden. Wir verwenden `qsort`, ein Quicksort aus der C-Standardbibliothek.

```
int list_quicksort(List *l)
{
  int arrLen, i;
```

```
  Node **nodeArr=getNodeArr(l, &arrLen);
  Node *cur;

  if (!nodeArr) {
    return -1;
  }
  qsort(nodeArr, arrLen, sizeof(Node*), node_compare);

  cur = l->first = nodeArr[0];
  for (i=1; i<arrLen; i++) {
    cur->next = nodeArr[i];
    cur = cur->next;
  }
  nodeArr[arrLen-1]->next = NULL;
  l->last = cur;
  free(nodeArr);
  return 0;
}
```

Die Funktion `qsort` wird aufgerufen mit

1. Einem Pointer auf die Daten, die sortiert werden sollen
2. Der Anzahl der Datensätze
3. Der Größe eines Datensatzes
4. Einem Pointer auf eine Vergleichsfunktion, die zwei Pointer auf die Datensätze nimmt.

Man sieht, dass diese Quicksort-Implementation so in etwa alles nach allen denkbaren Kriterien sortieren kann, was im Speicher hintereinander in gleich großen Blöcken steht. Diese Flexibilität macht den Aufruf etwas kompliziert, und in der Regel will man sie auch gar nicht ausnutzen. Größe Speicherblöcke umherschieben kostet Zeit, und so will man meist nur Pointer auf die Speicherblöcke sortieren.

Das passiert auch hier – wir sortieren die Pointer im `nodeArr`, das wir uns eigens für diesen Zweck besorgen. Aus diesem Array bauen wir danach die Liste neu auf. Da nun die Vergleichsfunktion Pointer auf die Datensätze bekommt und unsere Datensätze Pointer sind, bedeutet das, dass `nodeCompare` Pointer auf Pointer bekommt, obwohl im Prototyp `void *` angegeben ist. Wir reparieren das über einen Cast, und es ist empfehlenswert, bei eigenen Vergleichsfunktionen diesem Muster zu folgen.

Die Mühe hat sich gelohnt: Die Laufzeit des Programms auf der letzten Folie reduziert sich mit dieser Funktion von 4 auf 0.4 Sekunden. Die Laufzeit der reinen Sortierfunktion hat sich noch drastischer von 3.68 auf 0.04 Sekunden reduziert – eine Verbesserung um einen Faktor 100, die bei größeren Datensätzen auch noch deutlicher ausfallen würde.

Problems

(54.1)* Überlegt euch, wie ihr qsort aufrufen müsst und wie eure compare-Funktion aussehen muss, wenn ihr ein String-Array gemäß dem Rezept auf der Pointer III-Folie sortieren wollt.

55. Profiling

Wir haben gesehen, dass die Laufzeit unseres Programms von der Sortierfunktion dominiert war. Wie bekommt man so etwas heraus?

Wenn man eine Ahnung hat, wo man hinsehen muss, kann man in C die Funktion `clock()`, in Python `time()` aus dem Modul `time` verwenden.

Dabei ist `clock` in time.h deklariert. Die Funktion gibt die Zahl der „clock ticks“ sein Programmstart in einem Typ `clock_t` zurück, den man in der Regel bedenkenlos auf long casten kann. Wie lang ein clock tick ist, hängt vom System ab und kann dem Makro `CLOCKS_PER_SEC` entnommen werden. Das kann dann etwa so aussehen:

```
#include <time.h>
...
  long start;
...
  start = (long)clock();
<Code, dessen Laufzeit man messen möchte>
  printf("Laufzeit: %f\n", ((long)clock()-start)/(double)CLOCKS_PER_SEC);
...
```

In Python hat time.time() normalerweise schon ausreichend Auflösung. Ein entsprechendes Programmfragment könnte also sein:

```
import time
...
  start = time.time()
<Code, dessen Laufzeit man messen möchte>
  print "Laufzeit": start-time.time()
```

Sollte die Laufzeit des Codes zu kurz sein, kann man ihn häufig 10, 100 oder 1000 Mal hintereinander laufen lassen – für unsere Sortierroutine müsste das allerdings mit einiger Sorgfalt gemacht werden, denn viele Sortieralgorithmen haben ein anderes Laufzeitverhalten, wenn sie auf schon sortierte Daten laufen, und quicksort gehört nachhaltig zu dieser Kategorie. Profis können in Python übrigens auch das timeit-Modul für solche Dinge benutzen.

Besser ist meistens der Einsatz eines *Profiler*s. Ein Profiler ist ein Programm, das einem anderen Programm bei der Ausführung zusieht und dabei misst, wie viel Zeit dieses Programm für die Ausführung welcher Funktionen benötigt.

Profiling kann gibt es in zwei Geschmäckern: *Deterministisches Profiling* misst die Zeit jeweils beim Ein- und Austritt aus der Funktion und berechnet daraus ihre Laufzeit. Nachteil ist, dass das den Progammablauf erheblich verlangsamen kann, und auch, dass in Mulittasking-Umgebungen die Zeit nicht unbedingt in der Funktion selbst vergangen sein muss. Der Python-Profiler arbeitet so. *Statistisches Profiling* hingegen sieht in regelmäßigen Abständen nach, wo im Programm sich der Program Counter gerade befindet und berechnet endlich daraus individuelle Laufzeiten der Funktionen. Nachteil dieser Methode ist, dass sie als statistische Methode natürlich beliebig falsch liegen kann. Wenn wir z.B. alle 20 ms samplen und wir zwei Funktionen haben, die jeweils genau 10 ms laufen, so wird am Ende höchstwahrscheinlich der einen Funktion die ganze Rechenzeit zugeschlagen, der anderen gar keine, obwohl bei exakt die gleiche Menge Rechenzeit verbraucht haben. Der C-Profiler, den wir hier behandeln, `gprof`, ist ein statistischer Profiler.

Um ein C-Programm mit gprof zu profilen, muss es (mit allen Modulen) mit der Compileroption `-pg` kompiliert und gelinkt werden. Wir schreiben also `CFLAGS += -pg` im Makefile und machen `make clean;make`. Wenn das entstandene Programm gelaufen ist, steht die Profiling-Information in einer Datei `gmon.out`. Um deren Information auszuwerten, verwendet man das Programm `gprof`. Um auch Bibliotheksfunktionen profilen zu können, muss weiter `LDFLAGS += -static -lc_p` angegeben werden, damit eine fürs Profiling vorbereitete C-Bibliothek gelinkt wird. Dadurch wird das Binary im Allgemeinen erheblich größer und die Ausgabe des Profilers erheblich unübersichtlicher. Linux-Distributionen haben in der Regel ein eigenes Paket, das diese fürs Profilen gedachte Bibliothek enthält – Debian hat etwa libc6-prof.

```
examples> make; listSortTest
```

```
...
examples> gprof listSortTest
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
100.00      0.27     0.27     6178    43.70    43.70  getNodeAt
  0.00      0.27     0.00     6179     0.00     0.00  getNextWord
  0.00      0.27     0.00     6178     0.00     0.00  getNodeFro...
  0.00      0.27     0.00     6178     0.00     0.00  list_append
  0.00      0.27     0.00     6178     0.00    43.70  list_getAt
  0.00      0.27     0.00        2     0.00     0.00  list_len
  0.00      0.27     0.00        1     0.00     0.00  getNodeArr
  0.00      0.27     0.00        1     0.00     0.00  list_new
  0.00      0.27     0.00        1     0.00     0.00  list_quicksort
  0.00      0.27     0.00        1     0.00 270000.00  printList
...
```

Entsprechend mit selection sort:

```
 52.83      0.28     0.28     6178    45.32    45.32  findAndR...
 47.17      0.53     0.25     6178    40.47    40.47  getNodeAt
  0.00      0.53     0.00     6179     0.00     0.00  getNextWord
  0.00      0.53     0.00     6178     0.00     0.00  getNodeFro...
  0.00      0.53     0.00     6178     0.00     0.00  list_append
  0.00      0.53     0.00     6178     0.00    40.47  list_getAt
  0.00      0.53     0.00        1     0.00     0.00  list_len
  0.00      0.53     0.00        1     0.00     0.00  list_new
  0.00      0.53     0.00        1     0.00 280000.00  list_sort
  0.00      0.53     0.00        1     0.00 250000.00  printList
```

Als erstes fällt auf, dass das ganz anders aussieht als wir das oben behauptet haben. Die Gesamtlaufzeit von `list_sort` (Spalte total us/call) wird als 260 ms angegeben, im Gegensatz zum Ergebnis aus Messungen mit `clock`, die 3860 ms behauptet haben, die von `list_quicksort` als 0.0, obwohl wir 40 ms gemessen hatten. Wo ist die Rechenzeit hingekommen?

Sie steckt in den Bibliotheksfunktionen, vor allem in `strcoll`, für `list_quicksort` auch in `qsort`; linkt man diese nach obigem Rezept dazu, ergibt sich, dass beim selection sort 19073230 Aufrufe von strcoll gemacht werden, beim quicksort aber nur 70021, was schon ziemich viel des Laufzeitunterschiedes erklärt. Die Laufzeit von qsort selbst ist mit ein paar Mikrosekunden vernachlässigbar, ganz im Gegensatz zum findAndRemSmallest, das, da es jeweils ein gutes Stück Liste durchlaufen muss, im Mittel immerhin 42 Mikrosekunden pro Aufruf im eigenen Code läuft (die Zeit fürs strcoll kommt da natürlich noch dazu) und das 6171 Mal.

Dass die Zeiten für die Bibliotheksfunktionen ohne spezielle Standardbibliothek nicht einfach der aufrufenden Funktion zugeschlagen werden, liegt an der Funktionsweise von gprof – es mag zwar wissen, dass das Programm zu einem bestimmten Zeitpunkt in strcoll war, weiß aber nicht, woher strcoll aufgerufen wurde, denn strcoll steht nicht im so genannten call graph, einem Graphen, der zeigt, welche Funktion von welcher aufgerufen wurde und der in den sich eine Funktion sozusagen aktiv eintragen muss. Genau dafür aber muss eine Bibliothek modifiziert werden.

Die Bedeutung der Felder im oben gezeigten „flat profile“: Ganz vorne steht der Anteil der Zeit, den das Programm in der Funktion verbracht hat, danach die „cumulative seconds“, eine Summe der Zahlen in der dritten Spalte – sie ist hier weniger interessant. Die dritte Spalte, „self seconds“ ist die Zeit, die in der Funktion selbst verbracht wurde (die Zeit aufgerufener Funktionen zählt also nicht mit). Die vierte Spalte sagt, wie oft eine Funktion aufgerufen wurde, die fünfte sagt, wie viele Mikrosekunden bei jedem Aufruf in der Funktion selbst verbraten wurden, die vorletzte, wie viele Mikrosekunden in der Funktion selbst und den von ihr dabei aufgerufenen Funktionen, soweit diese im call graph auftauchen.

Was man an obigem flat profile noch sieht: Die zweite große Bremse im Programm ist `getNodeAt` bzw. `printList`. Wenn die Programme komplexer werden, mag es sein, dass nicht von vorneherein klar ist, dass `getNodeAt` hier von `printList` aufgerufen wurde. Dann hilft der etwas weiter unten in der gprof-Ausgabe stehenden call graph weiter. Informationen dazu gibt die Info-Seite von gprof.

Der Python-Profiler

Iterator

Pattern

Wenn man gprof verstanden hat, ist auch der Python-Profiler kein Geheimnis mehr. Bei normaler Verwendung durch

```
python /usr/local/lib/python2.2/profile.py npbayes.py
```

– der Pfad zu den Modulen der Python-Bibliothek ist dabei natürlich systemabhängig, und `npbayes.py` ist hier das zu profilende Skript – oder direkt im Skript durch

```
import profile
profile.run('main()')
```

– `main()` ist hier die zu profilende Funktion – werden gleich ein dem flat profile des gprof entsprechende Daten ausgegeben:

```
   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1000    0.020    0.000    0.020    0.000 npbayes.py:10(binomial)
     1000    0.020    0.000    0.050    0.000 npbayes.py:107(getProbability)
       20    0.000    0.000    0.120    0.006 npbayes.py:111(oneMapStep)
...
```

– man ahnt schon, dass hier die Zahl der Aufrufe, die Zeit, die insgesamt in der Funktion vergangen ist, die Zeit in der Funktion pro Aufruf, die Gesamtzeit mit aufgerufenen Funktionen und die Gesamtzeit mit aufgerufenen Funktionen pro Aufruf stehen.

Man kann sich die Profile-Daten aber auch ausgeben lassen und mit allerlei raffinierten Methoden auswerten. Wie das geht, verrät der Abschnitt über das Profile-Modul[19] in der Python Library Reference.

56. Iteratoren

An `printList` haben wir gesehen, dass das Iterieren über alle Elemente unserer Liste ziemlich langsam ist. Man könnte das beschleunigen, indem man Clients den Zugriff auf `Node`s gestattet, würde damit aber die Abstraktion ruinieren.

Alternative: *Iterator*en. Dabei besorgt man sich ein Objekt, das die Liste kennt und eine Methode hat, die das jeweils nächstes Objekt in der Liste zurückgibt.

Für unsere Liste muss der Iterator etwas von der internen Struktur der Liste wissen und muss deshalb ins gleiche Modul. Sein Interface:

- `Iterator` – ein öffentlicher Typ
- `Iterator *list_iteratorNew(List *l)` gibt einen Iterator für die Liste `l` zurück
- `void list_iteratorFree(Iterator *it)` gibt einen Iterator frei
- `char *list_iteratorNext(Iterator *it)` gibt das nächste Element zurück

Der Iterator ist übrigens ein klassisches Beispiel eines *Pattern*s, einer immer wieder anwendbaren Softwaretechnik.

```
typedef struct Iterator_s {
  List *list;
  Node *current;
} Iterator;

Iterator *list_iteratorNew(List *l)
{
  Iterator *it=malloc(sizeof(Iterator));

  if (it) {
    it->list = l;
```

[19] `http://docs.cl.uni-heidelberg.de/python/lib/profile.html`

```
    it->current = l->first;
  }
  return it;
}

void list_iteratorFree(Iterator *i)
{
  free(i);
}

char *list_iteratorNext(Iterator *it)
{
  char *res=NULL;

  if (it->current) {
    res = it->current->cont;
    it->current = it->current->next;
  }
  return res;
}
```

tradeoff

Damit lässt sich ein schnelleres `printList` so schreiben:

```
int printListFast(List *l)
{
  Iterator *it=list_iteratorNew(l);
  char *item;

  if (!it) {
    return -1;
  }
  while ((item=list_iteratorNext(it))) {
    printf("%s\n", item);
  }
  list_iteratorFree(it);
  return 0;
}
```

Das Problem ist jetzt, dass das Erzeugen des Iterators wegen des zusätzlichen Speicherbedarfs natürlich schief gehen kann, weshalb wir jetzt besser etwas zurückgeben, so dass aufrufende Funktionen ggf. prüfen können, ob auch alles geklappt hat. Es ist typisch, dass man einen *tradeoff*, einen Kompromiss also, zwischen Speicherbedarf und Geschwindigkeit machen muss – in diesem Fall allerdings ist der zusätzliche Speicher vernachlässigbar.

Das Ergebnis dieser Mühen ist folgendes Profile:

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00     6179     0.00     0.00  getNextWord
  0.00      0.00     0.00     6179     0.00     0.00  list_iteratorNext
  0.00      0.00     0.00     6178     0.00     0.00  getNodeFro...
  0.00      0.00     0.00     6178     0.00     0.00  list_append
  0.00      0.00     0.00        1     0.00     0.00  getNodeArr
  0.00      0.00     0.00        1     0.00     0.00  list_iteratorFree
  0.00      0.00     0.00        1     0.00     0.00  list_iteratorNew
  0.00      0.00     0.00        1     0.00     0.00  list_len
  0.00      0.00     0.00        1     0.00     0.00  list_new
  0.00      0.00     0.00        1     0.00     0.00  list_quicksort
  0.00      0.00     0.00        1     0.00     0.00  printListFast
```

– auf dem aktuellen Rechner braucht also kein Teil unseres Programms messbar Rechenzeit. Die

von der csh gemessene Laufzeit des Programms ist nun 0.02 Sekunden, gegenüber 0.4 Sekunden mit dem alten `printList` und 1.2 Sekunden mit dem alten `list_sort`.

Optimierung

Optimierung kann also viel bringen, die großen Sprünge gehen aber im Allgemeinen nur durch Änderung des Algorithmus'. Genau das verlangt aber Nachdenken und kostet Zeit, so dass man eigentlich nur dort Optimieren möchte, wo es etwas bringt. Bei der Bestimmung dieses Ortes helfen Profiler.

Der Iterator hat übrigens das Problem, dass er von Veränderungen an der Liste nichts mitbekommt – es ist z.B. ohne weiteres möglich, die Liste freizugeben und den Iterator weiter zu verwenden, was wiederum zu Abstürzen führen würde. Eine Möglichkeit, das zu unterbinden, wäre ein Callback auf einen gesetzten Iterator in der Liste; die Liste könnte, wenn sie verändert wird, dem Iterator sagen, dass er nicht mehr gültig ist. Ob sich dieser Aufwand lohnt, ist allerdings fragwürdig – man könnte auch einfach sagen, dass BenutzerInnen, die eine Liste, auf der ein Iterator werkt, freigeben, nichts besseres als einen Absturz verdient haben.

57. map, filter, und reduce

Iteratoren erlauben uns auch, die aus Python bekannten Funktionen `map` und `filter` zu schreiben, ohne etwas von der Listenimplementation wissen zu müssen.

```
List *map(char *(*mapFun)(char *arg), List *l)
{
  List *new=list_new();  char *item;
  Iterator *it=list_iteratorNew(l);

  while ((item=list_iteratorNext(it))) {
    list_append(new, mapFun(item));
  }
  list_iteratorFree(it);
  return new;
}

List *filter(int (*filterFun)(char *arg),
  List *l)
{
  List *new=list_new(); char *item;
  Iterator *it=list_iteratorNew(l);

  while ((item=list_iteratorNext(it))) {
    if (filterFun(item)) {
      list_append(new, item);
    }
  }
  list_iteratorFree(it);
  return new;
}
```

Auch reduce wäre denkbar – allerdings ist das für Strings etwas müßig. Mit Zahlen ist sowas interessanter, allerdings verlieren wir dann etwas von der Eleganz des reduce-Konzepts. Brauchbar könnte sowas sein:

```
int reduceFloat(double (*reduceFun)(double accum, char *arg2), List *l)
{
  double result=0;
  Iterator *it=list_iteratorNew(l);
  char *item;
```

```
  while ((item=list_iteratorNext(it))) {
    result = reduceFun(result, item);
  }
  list_iteratorFree(it);
  return result;
}
```

– es wird also eine Funktion mit einem „Akkumulator“ übergeben, in dem jeweils das Ergebnis der Anwendung auf das vorherige Element steht. Wozu das gut sein kann, mag im unten stehenden Beispiel klar werden.

Wir haben hier auf Fehlerbehandlung verzeichtet, an sich müssten natürlich die Rückgabewerte von `list_iteratorNew` und Freunden geprüft werden.

Nützlich sind außerdem Varianten, die die alten Listen gleich freigeben:

```
List *mapAndFree(char *(*mapFun)(char *arg), List *l)
{
  List *res = map(mapFun, l);

  list_free(l);
  return res;
}

List *filterAndFree(int (*filterFun)(char *arg), List *l)
{
  List *res = filter(filterFun, l);

  list_free(l);
  return res;
}
```

Eine Funktion, die die mittlere Länge der groß geschriebenen Wörter in einem Text berechnet, könnte dann etwa so aussehen:

```
int isupperStr(char *s)
{
  return isupper(*s);
}

double addLen(double accum, char *s)
{
  return accum+strlen(s);
}

void processStuff(char *fname)
{
  List *wordList = getWordList(fname);
  List *upCaseWords;

  if (!wordList) { return; }
  printf("Now looking at %s...\n", fname);
  upCaseWords = filterAndFree(isupperStr,
    mapAndFree(strip, wordList));
  printf("Avg length of uppercase words: %f\n",
    reduceFloat(addLen, upCaseWords)/
    (float)list_len(upCaseWords));
  list_free(upCaseWords);
}
```

Nachzutragen bleibt noch die Implementation für die verwendeten Funktionen `getWordList` und `strip`:

```
#define MAX_WORD_LEN 80

char *strip(char *word)
{
  static char stripped[MAX_WORD_LEN];
  char *sp=word, *dp=stripped;

  while (isspace(*sp)) {
    sp++;
  }
  while (*sp && (dp-stripped)<MAX_WORD_LEN-1) {
    *dp++ = *sp++;
  }
  *dp = 0;
  while (isspace(*--dp)) {
    *dp = 0;
  }
  return stripped;
}

List *getWordList(char *fname)
{
  FILE *inF = fopen(fname, "r");
  List *words = list_new();
  char *w;

  if (!inF) {
    if (words) {
      list_free(words);
    }
    return NULL;
  }
  while ((w=getNextWord(inF))) {
    list_append(words, w);
  }
  fclose(inF);
  return words;
}
```

58. Python mit C erweitern

Manche Aufgaben, die Python in zwei Zeilen lösbar sind, brauchen in C Dutzende und sind schwierig zu machen – dann wieder hätte man gern die rohe Kraft von C. Ideal wäre eine Kombination aus beiden.

Dazu kann man Python-Module in C schreiben. Einfachstes Beispiel: Ein Modul `hello`, das die Funktion `helloWorld` exportiert. Das könnte so aussehen:

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

static PyObject *hello_printHello(PyObject *self, PyObject *args)
{
  if (!PyArg_ParseTuple(args, "")) {
    return NULL;
  }
```

```
  printf("Hello world!\n");
  Py_INCREF(Py_None);
  return Py_None;
}

static PyMethodDef helloMethods[] = {
  {"printHello", hello_printHello, METH_VARARGS,
    "Prints 'Hello World'."},
  {NULL, NULL, 0, NULL},
};

void inithello(void)
{
  Py_InitModule("hello", helloMethods);
}
```

Python-Module müssen etwas anders gelinkt werden als normale C-Programme, und leider auf jeder Plattform etwas anders. Die Python-Distutils[20] nehmen einem viel von dieser Arbeit ab.

Dazu schreibt man eine Datei `setup.py`:

```
from distutils.core import setup, Extension

setup(name = "hello", version = "1.0",
  ext_modules = [
    Extension("hello", ["hello.c"])
    ]
)
```

und eine weitere Datei `setup.cfg`

```
[build_ext]
inplace=1
```

Zur Namensgebung: Wenn man `import hello` sagt, sucht Python zunächst nacheinander nach `bla.so`, `blamodule.so` und `bla.py` – module wird also ggf. hinzugefügt (ähnlich wie das lib vom Linker bei der -l-Option), wenn nach Erweiterungsmodulen gesucht wird. Die Extension .so steht für shared object und ist unter Unix für zur Laufzeit ladbare Objektdateien reserviert. Das Pendant unter Windows ist .pyd. Dank der Distutils müssen wir uns darüber aber auch keine Sorgen machen.

Die Distutils-Steuerung erfolgt über das kleine Python-Skript `setup.py`, das eigentlich nur eine Funktion (eben `setup`) aufruft. Diese Funktion bekommt potentiell unzählige Keyword-Argumente, hier etwa den Namen der Distribution (der euch zunächst genauso egal sein kann wie die Version) und eine Liste von Erweiterungsmodulen, die gebaut werden sollen; jedes Modul ist dabei eine Instanz der Klasse Extension, die im einfachsten Fall den Zielnamen und eine Liste der dazu zusammenzubauenden Dateien enthält. Damit lässt sich natürlich viel mehr machen – für die Details verweise ich auf die oben verlinkte vollständige Dokumentation.

Die Aktionen von Setup lassen sich weiter steuern durch die zusätzliche Datei `setup.cfg`, die keinen Python-Code enthält, sondern in Sektionen unterteilte Key/Value-Paare. Alles, was hier einstellbar ist, wäre auch über Kommandozeilenoptionen machbar, und vieles ist relativ esoterisch. Der oben gezeigte Eintrag dient nur dazu, das Modul, das wir bauen, nicht irgendwo in temporären Verzeichnissen zu verstecken, sondern wirklich in unserem Verzeichnis zu hinterlassen.

Was passiert nun in unserem Modul? Zunächst binden wir die Prototypen für allerlei Python-spezifische Funktionen und „Klassen“ ein. Alles, was mit Python zu tun hat, fängt mit `Py` oder `PY` an.

Danach definieren wir eine Python-Funktion. Funktionen, die von Python aus aufgerufen werden, habem *immer* einen Prototypen dieser Art: Sie nehmen zwei `PyObjectss` und geben ein `PyObject` zurück. In einem `PyObject` (in Wirklichkeit eine Struktur) kann irgendein Python-Wert stehen.

[20] `http://docs.cl.uni-heidelberg.de/python/dist/dist.html`

Das erste Argument wird dabei nur bei Methoden (also an Objekte gebundene Funktionen) verwendet und entspricht genau dem `self`, das wir in den Methoden immer verwendet hatten. Das zweite Argument ist ein Python-Tupel, das die Argumente enthält, die an die Python-Funktion übergeben wurden. Diese Argumente werden üblicherweise mit der Funktion `PyArg_ParseTuple` in C-Variablen übersetzt. `PyArg_ParseTuple` funktioniert etwa wie `sscanf`: Im ersten Argument kommt das `PyObject`, dann ein Formatstring (der hier aber ohne % auskommt, weil keine Schablone nötig ist). Die verschiedenen Formatcodes sind der Dokumentation zu parseTuple[21] zu entnehmen.

Wir wollen hier nur sicher stellen, dass wir kein Argument bekommen und prüfen deshalb auf einen leeren Formatstring. Wenn `PyArg_ParseTuple` einen Fehler bemerkt (hier also etwa übergebene Argumente, aber z.B. auch falsche Argumente), gibt sie NULL zurück. Das ist das Signal, dass etwas nicht geklappt hat, und wenn eine Funktion NULL an Python zurückgibt, wird eine Exception ausgelöst. Welche, lässt sich beispielsweise mit der Funktion `PyErr_SetString` angeben – näheres dazu unter Errors and Exceptions[22]. Hier macht das aber bereits `PyArg_ParseTuple` für uns, wir brauchen nur noch die NULL weiterzureichen, um die Exception auch auszulösen.

Dann tun wir, was wir tun wollen – in dem Fall lediglich ein printf – und geben `None` zurück, das von C aus als `Py_None` erscheint. Zur INCREF-Magie siehe die nächste Folie.

Als nächstes definieren wir die Schnittstelle zu Python. Wir hatten `hello_printHello` als `static` deklariert – wir teilen unseren Namespace hier mit dem Interpreter und möglicherweise anderen Erweiterungen, wollen da also möglichst wenig Namen nach außen sichtbar machen. Damit Python trotzdem weiß, wo es die Funktion findet, definieren wir uns eine Art „Sprungleiste“, eine Tabelle, in der die Funktionen drinstehen, die wir exportieren wollen. In diesem Fall ist das nur eine. Wir initialisieren dabei ein Array von Structs, wobei jeder Struct aus dem Python-Namen der Funktion, einem Pointer auf die Funktion, der Konstante `METH_VARARGS` (etwas anderes wollt ihr da wirklich nicht stehen haben) sowie einem Docstring für die Funktion besteht. Abgeschlossen wird die Liste durch einen Struct mit lauter Nullen.

Beachtet, dass damit die C-Namen von den Python-Namen völlig entkoppelt sind – wir hätten `hello_printHello` auch `foo` nennen können und es Python als `printHello` verkaufen können. Dennoch ist es eher eine gute Idee, die C-Funktionen so zu nennen, dass sie erahnen lassen, als was sie später ggf. Python zu sehen bekommt. Auch der Namen der Sprungleiste (hier `helloMethods`) ist beliebig, sollte konventionell aber `<modulname>Methods` sein.

Dieser Name spielt dann in der einzigen nichtstatischen Funktion des Moduls eine Rollen, `inithello`. Diese Funktion hat auch als einzige einen festen Namen, nämlich `init<modulname>`. Sie wird bei einem `import` des Moduls aufgerufen und kann alles mögliche tun, was das Modul so braucht, bevor irgendwelche Funktionen aufgerufen werden können. Zumeist reicht es aber, die Funktion `Py_InitModule` mit dem Modulnamen und der Sprungleiste aufzurufen. Sie sorgt dafür, dass Python weiß, welche Fuktionen das Modul nun wirklich exportiert (auf die Sprungleiste kann es ja nicht direkt zugreifen, da sie ja statisch ist).

Wir können dann folgendes tun:

```
examples> setup.py build
running build_ext
building 'hello' extension
gcc -DNDEBUG -g -O3 -Wall -Wstrict-prototypes -fPIC ...
gcc -shared build/temp.linux-i686-2.2/hellomodule.o ...
examples> python
>>> import hello
>>> hello.printHello()
Hello world!
>>> hello.printHello("Quatsch")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: function takes exactly 0 arguments (1 given)
>>> hello.printHello
<built-in function printHello>
```

[21] http://docs.cl.uni-heidelberg.de/python/ext/parseTuple.html
[22] http://docs.cl.uni-heidelberg.de/python/ext/errors.html

Built-in function! Toll.

Reference Count

owned reference

borrowed reference

Übrigens machen die Distutils auch keine schwarze Magie; Python-Erweiterungsmodule können auch mit ganz normalen Bordmitteln gebaut werden, etwa mit einem Makefile dieser Art (hier für Systeme mit den GNU binutils):

```
PYTHON_COMPILE = -I /usr/local/include/python2.2
CFLAGS += -Wall
CFLAGS += $(PYTHON_COMPILE)

%: %.o
  $(CC) -o $@ $(LDFLAGS) $(CFLAGS) $^

%.so: %.o
  $(CC) -shared $^ -o $@

hellomodule.so: hellomodule.o
```

Dabei muss dann allerdings der Pfad in `PYTHON_COMPILE` jeweils per Hand angepasst werden – er könnte z.B. auch `/usr/include/python2.2` lauten. Natürlich geht das unter Windows ganz anders, und andere Unixe wollen auch andere Optionen haben. Das ist eine unglückliche Situation, weshalb eben die AutorInnen von Python sich die Distutils ausgedacht haben.

— Dateien zu diesem Abschnitt in der HTML-Version—

Problems

(58.1)* Holt euch den Quellcode des Hello-Moduls von der Vorlesungsseite und baut es mit `python hellosetup.py build`. Probiert das entstandene Modul. Probiert, ob ihr im etwas interessantere Funktionalität einhauchen könnt.

59. Refcounting

Eine dynamische Sprache wie Python verlangt, dass sich der Rechner darum kümmert, dass Objekte, die nicht mehr gebraucht werden, freigegeben werden. Python erreicht dies üblicherweise über *Reference Count*s. Dabei hat jedes Objekt einen Zähler, in dem die Zahl der jeweils auf das Objekt gehaltenen Referenzen steht. Wenn die Zahl der Referenzen auf Null geht, wird das Objekt freigegeben.

Wenn man also Python-Werte verwendet, muss man das in der Regel durch einen Aufruf von `Py_XINCREF` mit dem Objekt als Argument kundtun, wenn man mit dem Objekt fertig ist, muss man `Py_XDECREF` aufrufen. Das X in diesen Namen steht da, weil es auch noch Varianten ohne X gibt. Der Unterschied ist, dass man den X-Varianten bedenkenlos Nullpointer übergeben kann, was bei den Varianten ohne X zu einem Segfault führt. Dafür sind letztere schneller.

Die meisten Funktionen, die Python-Objekte aus C-Variablen erzeugen, geben bereits ge-INCREF-te Objekte zurück (*owned reference*), man darf sie also nicht selbst INCREFen. Andere Funktionen geben *borrowed reference*s zurück; diese müssen und dürfen wir nicht DECREFen, (wenn wir sie nicht durch INCREFen zu owned references gemacht haben).

In der Python/C API[23] wird jeweils angegeben, ob eine Funktion eine owned oder eine borrowed Reference zurückgibt. Wer C-Module programmiert sollte auf jeden Fall das Refcount-Kapitel[24] in Extending and Embedding[25] gelesen haben.

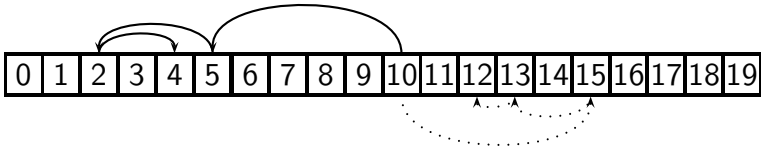
Zwei Beispiele: Wenn wir einen C-String aus Python zurückgeben wollen, verwenden wir

```
PyObject *getAString(void)
{
  return PyString_FromString("Ein String");
```

[23] `http://docs.cl.uni-heidelberg.de/python/api/api.html`

[24] `http://docs.cl.uni-heidelberg.de/python/ext/refcounts.html`

[25] `http://docs.cl.uni-heidelberg.de/python/ext/ext.html`



flat file

binary search

Fig. 15

```
}
```

– kein INCREF, weil die Funktion eine New Reference zurückgibt.

Wenn wir ein Element aus einem Tupel zurückgeben wollen, verwenden wir

```
PyObject *getElement(PyObject *tuple, int index)
{
  PyObject *el=PyTuple_GetItem(tuple, index);

  Py_XINCREF(el);
  return el;
}
```

– INCREF, weil die Funktion eine Borrowed Reference zurückgibt und wir keine geborgten Referenzen an Python geben wollen.

In diesem Beispiel sieht man auch, warum die X-Varianten verdienstvoll sind. Wen z.B. das GetItem einen IndexError auslöst (was ja gut möglich ist), ist `el` NULL, ein INCREF würde einen Segfault auslösen. XINCREF tut aber gar nichts, die NULL wird an Python zurückgegeben, eine Exception wird ausgelöst, und GetItem hat vermutlich sogar den richtigen Fehlerstring gesetzt: Alles wunderbar.

60. Binary Search

Wir wollen nun eine tatsächlich nützliche Erweiterung schreiben. Dazu brauchen wir einige Präliminarien.

Ein häufig vorkommendes Problem ist die Suche in großen Datenmengen. Datenbanken (in Python kann man auf jeden Fall anydbm verwenden) sind ein Möglichkeit, es zu lösen, häufig ist es aber einfacher, die Daten in *flat file*s zu halten, also einfachen Textdateien, in denen jede Zeile einem Eintrag entspricht. Um darin schnell Einträge zu finden, bietet sich *binary search* an.

Dabei müssen die Daten sortiert sein, und man geht vor wie beim Zahlenraten: Nimm den Eintrag in der Mitte, wenn der Suchschlüssel größer ist, mach in der oberen Hälfte weiter, sonst in der unteren, höre auf, wenn du den Eintrag gefunden hast oder sicher bist, dass du ihn nicht mehr findest. Das Python-Modul `bisect` implementiert das schon. Eine Funktion daraus:

```
def bisect_right(a, x, lo=0, hi=None):
    if hi is None:
        hi = len(a)
    while lo < hi:
        mid = (lo+hi)//2
        if x < a[mid]: hi = mid
        else: lo = mid+1
    return lo
```

(cf. Fig. 15)

Wir wollen das für Textdateien haben, die z.B. so aussehen könnten:

```
a AP
AAA FW
Aarhus NP
Aaron NP
aback AVP
```

```
abacus NN
abalone NP
abandon VB
```

Dabei kommt als Komplikation gegenüber dem einfachen Fall oben hinzu, dass wir nicht wissen, wo die Datenfelder jeweils anfangen, denn die Textzeilen haben ja verschiedene Längen. Das Unix-Programm look kann damit aber umgehen. Wir möchten es jetzt in Python verwenden, und zwar ohne `os.system` oder Freunde.

Der Quelltext des BSD-Programms ist z.B. im Paket util-linux zu finden (vgl. Freshmeat[26]) oder auch auf der Webseite zu dieser Vorlesung.

Problems

(60.1)* Wie viele Schritte braucht ein Binary Search im schlimmsten Fall, um ein gesuchtes Element aus einer Liste von N Elementen zu finden? **(L)**

61. Was tut look?

Ausgangspunkt: Die main-Funktion von `look.c`:

```
int
main(int argc, char *argv[])
{
  struct stat sb;
  int ch, fd, termchar;
  char *back, *file, *front, *p;

  setlocale(LC_ALL, "");
  bindtextdomain(PACKAGE, LOCALEDIR);
  textdomain(PACKAGE);

  setlocale(LC_ALL, "");

  file = _PATH_WORDS;
  termchar = '\0';
  string = NULL;    /* just for gcc */

  while ((ch = getopt(argc, argv, "adft:")) != EOF)
    switch(ch) {
    case 'a':
      file = _PATH_WORDS_ALT; break;
    case 'd':
      dflag = 1; break;
    case 'f':
      fflag = 1; break;
    case 't':
      termchar = *optarg; break;
    case '?':
    default:
      usage();
    }
  argc -= optind;
  argv += optind;
  switch (argc) {
  case 2:        /* Don't set -df for user. */
    string = *argv++;
    file = *argv;
    break;
  case 1:        /* But set -df by default. */
    dflag = fflag = 1;
    string = *argv;
```

[26] http://www.freshmeat.net

```
    break;
  default:
    usage();
  }
  if (termchar != '\0' && (p = strchr(string, termchar))
    != NULL)
    *++p = '\0';

  if ((fd = open(file, O_RDONLY, 0)) < 0 || fstat(fd, &sb))
    err("%s: %s", file, strerror(errno));
  front = mmap(NULL, (size_t) sb.st_size, PROT_READ,
#ifdef MAP_FILE
         MAP_FILE |
#endif
         MAP_SHARED, fd, (off_t) 0);
  if
#ifdef MAP_FAILED
     (front == MAP_FAILED)
#else
     ((void *)(front) <= (void *)0)
#endif
    err("%s: %s", file, strerror(errno));

  back = front + sb.st_size;
  return look(front, back);
}
```

Was passiert hier? Am Anfang wird das locale gesetzt – das sollten wir im Python-Modul nicht mehr tun, Python hat das schon für uns getan, und wir sollten die Weisheit des Interpreters nicht mutwillig in Frage stellen. Gleiches gilt für die Aufrufe mit „textdomain“. Bei ihnen geht es um die automatische Übersetzung von Fehlermeldungen, was uns zu schwierig ist. Warum das locale zwei Mal gesetzt wird, entzieht sich meiner Kenntnis.

Dann werden einige Defaults gesetzt. Die Manpage zu look verrät, dass look entweder als `look <string>` oder als `look <string> <file>` aufgerufen werden kann und im ersten Fall `/usr/dict/words` (was hier im Makro `_PATH_WORDS` steht) als Datei angenommen wird. In termchar kann ein Zeichen angegeben werden, das als „Feldtrenner“ dient – dies ist zunächst das Nullbyte, also das Stringende, womit der ganze String als relevant für den Match angesehen wird. Die Initialisierung von `string` – nach diesem String wird später gesucht werden – dient nur der Befriedigung von Compilern, die sich ansonsten beschweren könnten, dass `string` möglicherweise nicht korrekt initialisert worden ist, da die eigentliche Zuweisung in einem switch-Statement erfolgt.

Dann werden die Kommandozeilenoptionen geparst, und zwar mit Hilfe der Bibliotheksfunktion `getopt` („Get options“). Die Funktion interpretiert Elemente aus `argv` der Form `-<char>`, wobei char aus dem dritten Argument kommen muss, der als eine Art Formatstring dient – ist hinter einem Zeichen ein Doppelpunkt, so nimmt die entsprechende Option einen Parameter. Näheres ist der Manpage von getopt zu entnehmen. Das Codefragment hier kann durchaus als Schablone dienen, die für eigene Programme angepasst werden kann.

Was die einzelnen Optionen bedeuten, steht in der Manpage. Für uns ist auf jeden Fall wichtig `-f`, das festlegt, ob Groß-/Kleinschreibung beim Vergleich ignoriert werden soll oder nicht (case folding). Repräsentiert wird das hier durch eine globale Variable `fflag`. Dass hier mit globalen Variablen operiert wird, sollte nicht zur Nachahmung anregen.

Nach dem Parsen der Optionen werden argc und argv so manipuliert, dass nun nur noch normale Argumente sichtbar sind. Diese werden im folgenden switch ausgewertet; bemerkenswert daran ist allenfalls, dass, wenn nur ein Argument vorhanden ist, die d- und f-Flags künstlich gesetzt werden – die Manpage verlangt das so.

Danach geht die eigentliche Arbeit los. Die Details brauchen uns hier nicht zu interessieren, look verwendet hier eine Funktion, die es erlaubt, Dateien „in den Speicher einzublenden“, so dass wir in Dateien herumfahren können wie in Arrays. Dazu müssen die Dateien mit einer Funktion `open` geöffnet werden, die quasi unter dem bekannten `fopen` liegt und daher auch gefährlicher

ist. Gibt sie eine negative Zahl zurück, ist ein Fehler aufgetreten und es wird eine entsprechende Medldung ausgegeben.

Danach erfolgt der Aufruf von `mmap` (Memory Map – „bilde in den Speicher ab“). Die Details interessieren hier wie gesagt nicht. Die folgende Erfolgskontrolle ist aber bemerkenswert – hier wird in Abhängigkeit von einem Präprozessormakro die Prüfbedingung verändert. Das ist nötig, weil die Semantik des `mmap`-Aufrufs verschiedener Systeme nicht einheitlich ist – manchmal deutet eine negative Rückgabe ein Problem an, manchmal die Rückgabe des magischen Werts `MAP_FAILED`.

Schließlich wird noch der Pointer auf das Ende des Suchbereichs gesetzt (das ist natürlich das Ende der gemappten Datei) und dann die eigentliche look-Funktion aufgerufen.

62. Look als Modul I

Um uns das Leben zunächst maximal einfach zu machen, wollen wir jetzt einfach look.c:main durch einen Wrapper ersetzen, der das, was wir von main brauchen, tut und dann das alte look.c:look aufruft.

```
static PyObject *look1_look(PyObject *self, PyObject *args)
{
  struct stat sb;
  int fd;
  void *res;
  char *file;
  char *front, *back;

  if (!PyArg_ParseTuple(args, "ss", &file, &string)) {
    return NULL;
  }
  if ((fd = open(file, O_RDONLY, 0)) < 0 || fstat(fd, &sb)) {
    PyErr_SetFromErrno(PyExc_IOError);
    return NULL;
  }
  /* ... weitgehend wie gehabt ... */
  res = look(front, back);
  munmap(front, sb.st_size);
  close(fd);
  if (res!=NULL) {
    Py_INCREF(Py_None);
    res = Py_None;
  }
  return res;
}
```

Änderungen:

- Wir wollen Fehler von Python behandeln lassen. Deshalb haben wir den alten `err`-Aufruf durch das Setzen einer Exception ersetzt.
- Gleiches gilt für Fehler, die in look selbst auftreten. Deshalb gibt look NULL zurück, wenn etwas schiefgeht.
- Das alte look verließ sich auf das Betriebssystem, um aufzuräumen (den map rückgängig machen und die Datei schließen). Da Python nach dem Funktionsaufruf weiter läuft, müssen wir jetzt selbst aufräumen.
- Alle globalen Variablen wurden statisch definiert.

Den veränderten Quellcode gibt es auf der Vorlesungsseite. Um alle Änderungen am Code zu sehen, könnt ihr das Unix-Kommando `diff look.c look1module.c` laufen lassen.

— Dateien zu diesem Abschnitt in der HTML-Version—

63. Look als Modul II

state

Wunschliste:

1. Nicht jedes Mal die Datei neu aufmachen und mappen müssen
2. Ergebisse nicht drucken, sondern als Liste von Strings zurückgeben

Für (1) brauchen wir *state*, einen Zustand: Wir müssen uns merken, welche Datei wir bearbeiten. Es ist meistens nicht gut, Zustände in Modulen zu speichern – dafür haben wir Objekte. Zur Simulation einer Klasse definieren wir die Typstruktur:

```
static PyTypeObject Lookertype = {
  PyObject_HEAD_INIT(&PyType_Type)
  0,  /* never mind */
  "Looker", /* Name */
  sizeof(lookerobject),
  0,  /* never mind */
  (destructor)Looker_destroy,
  (printfunc)0,
  (getattrfunc)Looker_getattr,
  (setattrfunc)0,
  (cmpfunc)0,
  (reprfunc)0,
  0, /* as number */
  0, /* as sequence */
  0, /* as mapping */
};
```

Im Wesentlichen haben wir hier wieder eine Sprungleiste, in der verschiedene Standardmethoden stehen, die der Typ (die Klasse) so unterstützen sollte. Es gibt tatsächlich noch eine ganze Ecke mehr, die uns aber hier nicht interessieren.

Für uns wichtig sind erstmal nur getattr und destroy.

Das sind erst ein paar Methoden – wo stehen die Daten? Im `lookerobject`:

```
typedef struct {
  PyObject_HEAD
  int dflag, fflag;
  int stringlen;
  char *string;
  char *comparbuf;
  char *g_front,*g_back;
  int fd;
  int searchcount;
} lookerobject;
```

Das merkwürdige `PyObject_HEAD` expandiert zu Platzhaltern für z.B. Refcounts. Was da drin steht, interessiert uns nicht. Der Rest sind einfach die Nutzdaten. Wir können sie hier alle in C halten, würden wir hier Python-Objekte drin haben, müssten wir auf jeden Fall Refcounting betreiben.

Und wo sind unsere Methoden? Üblicherweise in einer weiteren Sprungleiste:

```
static struct PyMethodDef looker_methods[]={
{"look", (PyCFunction)Looker_look, METH_VARARGS},
{NULL, NULL}
};
```

Das ist aber nur eine Konvention, da der einzige Platz, an dem wir das brauchen, unsere __getattr__-Methode ist:

```
static PyObject *Looker_getattr(
  lookerobject *self,char *name)
{
```

```
  return Py_FindMethod(looker_methods,
    (PyObject*)self,name);
}
```

64. Look als Modul III

In C müssen wir selbst aufräumen:

```
static void looker_destroy(lookerobject *self)
{
  munmap(self->g_front,
    self->g_back-self->g_front);
  close(self->fd);
  if (self->comparbuf) {
    free(self->comparbuf);
  }
  PyObject_Del(self);
}
```

Das Ganze muss natürlich noch ein Modul werden:

```
static PyObject *looker_new(PyObject *self, PyObject *args)
{  char *fname;

  if (!PyArg_ParseTuple(args, "s", &fname)) {
    return NULL;
  }
  return fillLookerObjectStruct(fname);
}

static PyMethodDef lookerMethods[] = {
  {"Looker", looker_new, METH_VARARGS,
    "Creates a new looker instance."},
  {NULL, NULL, 0, NULL},
};

void initLooker(void)
{
  Py_InitModule("Looker", lookerMethods);
}
```

Der Konstruktor steht hier also etwas allein, nämlich als Funktion im Modul und nicht als Methode der Klasse. Das muss auch mehr oder weniger so sein, weil sein Name nach außen sichtbar ist.

Die Funktion `fillLookerObjectStruct`, die wir im Konstruktor verwenden, ist eine relativ langweilige Hilfsfunktion, die Teile der alten Initialisierung verwendet und ansonsten die früher globalen Variablen setzt:

```
static PyObject *fillLookerObjectStruct(char *fname)
{  struct stat sb;
  lookerobject *self;

  self = PyObject_New(lookerobject, &Lookertype);
  if (!self) { /* If this fails, we're too hosed to even bother */
    return NULL; /* setting an error string */
  }

  if ((self->fd = open(fname, O_RDONLY, 0)) < 0 | fstat(self->fd, &sb)) {
    PyErr_SetFromErrno(PyExc_IOError);
    return NULL;
  }
  if ((void *)(self->g_front = mmap(NULL,
          (size_t)sb.st_size,
          PROT_READ,
          MAP_FILE|MAP_SHARED,
          self->fd,
```

```
        (off_t)0)) <= (void *)0) {
    PyErr_SetFromErrno(PyExc_IOError);
    return NULL;
  }
  self->g_back = self->g_front + sb.st_size;
  self->dflag = 0;
  self->fflag = 1;
  self->comparbuf = NULL;
  self->string = NULL;
  self->searchcount = 0;
  return (PyObject*)self;
}
```

Um das Ergebnis als String zurückzugeben (Wunsch 2), müssen wir die alte `print_from`-Funktion verändern, etwa so:

```
static PyObject*
collect_from(lookerobject *self, char *front, char *back)
{
  int eol;
  char *strbeg;

  strbeg = front;
  while (front < back && compare(self, front, back,
    self->fflag) == EQUAL) {
    if (compare(self, front, back, self->fflag) == EQUAL) {
      eol = 0;
      while (front < back && !eol) {
        if (*front++ == '\n')
          eol = 1;
      }
    } else
      SKIP_PAST_NEWLINE(front, back);
  }
  return PyString_FromStringAndSize(strbeg, front-strbeg);
}
```

Damit müssen natürlich auch die darunterliegenden Funktionen etwas verändert werden. Ihr findet den Quelltext im Anhang dieser Seite.

In `collect_from` packen wir einfach alles, was früher ausgegeben wurde, in einen Python-String. Dazu müssen wir noch nicht mal verstanden haben, was eigentlich alles im look vor sich geht.

In der Tat ist das auch etwas verwickelt, wie ihr seht, wenn ihr die (guten) Kommentare im Original-Quelltext lest. Für Interessierte hier dennoch der grobe Ablauf:

1. look macht zunächst wirklich eine Binärsuche, wobei es jeweils den Pointer *p* nimmt, der genau zwischen *front* und *back* liegt. Weil das in der Regel kein Zeilenanfang sein wird, läuft er vor einem Vergleich mit dem *p* zuerst bis nach dem ersten newline (im Makro `SKIP_PAST_NEWLINE`).
2. Die Binärsuche bricht ab, wenn das *p* im Laufe der Binärsuche gleich *back* wird. Einerseits ist dann nämlich eine Zeile länger als die Hälfte von *back* − *front*, und da unsere Zeilen hoffentlich nicht so furchtbar lang ist, bedeutet das, dass wir nicht mehr viel durchsuchen müssen, weil wir zweitens nach der Definition der Binärsuche wissen, dass das gesuchte Element zwischen *front* und *back* ist.
3. Weil wir wissen, dass wir nicht mehr viel durchsuchen müssen (das n aus $O(n)$ ist klein), können wir jetzt linear durchsuchen (das ist die Funktion `linear_search`). Darin gucken wir hinter jedem Newline, ob wir das gesuchte gefunden haben, ob wir noch davor sind (und weitersuchen wollen) oder ob das, was wir sehen, schon kleiner ist als das, was wir suchen (dann gibt es keine passende Zeile, und wir signalisieren den Fehlschlag).
4. Wenn wir etwas gefunden haben, geben wir es aus (bzw. zurück), und zwar in `collect_from` (bzw. im Original `print_from`). Dabei wollen wir einfach linear Zeile um Zeile auf einen Match prüfen und aufhören, wenn wir entweder *back* erreicht haben (dahinter kann es per Definitionem keine Matches mehr geben) oder der Vergleich fehlschlägt (auch danach darf nichts mehr sein, das passen würde).

Wenn man lange genug darüber nachdenkt, findet man, dass das äußere `if` überflüssig ist, weil seine Bedingung nie falsch werden kann (das stimmt aber auch nur, weil compare keine Seiteneffekte hat). Vermutlich war hier irgendwann mal anderer Code, mit dem zusammen die Abfrage Sinn hatte. Merke: Auch veröffentlichter Code ist nicht immer perfekt. threads

Draußen in der Welt

Wenn ihr den Eindruck habt, dass es relativ haarig ist, Python-Erweiterungen in C zu schreiben, so liegt ihr richtig. Das haben sich auch andere Menschen gedacht. In der Realität wir man darum in der Regel Hilfsmittel verwenden, die die Erzeugung von solchen Erweiterungen leichter machen. Dazu gehört z.B. SWIG, das beim Schreiben von „wrappern“ um bestehende C- oder C++-Bibliotheken hilft, oder Pyrex, das die Grenze zwischen Python und C recht erfolgreich verwischt. Googelt danach.

— Dateien zu diesem Abschnitt in der HTML-Version—

65. Fehlermeldungen von Bibliotheksfunktionen

Im Rückkehrwert der meisten Bibliotheksfunktionen ist nur kodiert, ob „es“ funktioniert hat oder nicht. Der konkrete Grund des Scheiterns steht in `errno`, definiert in `errno.h`.

Was `errno` wirklich ist, ist implementationsabhängig. Früher mal war es einfach ein int, mittlerweile kann auch etwas Komplizierteres dahinterstecken. Grund dafür sind vor allem *threads*, in denen ein Prozess mehrere Dinge quasi-gleichzeitig erledigen kann. Weil aber in einem Prozess ein Variablenname immer einer Speicheradresse zugeordnet ist, könnte es passieren, dass ein Thread einen Fehler auslöst, dann ein anderer Thread zum Zug kommt, ebenfalls einen Fehler auslöst und damit errno setzt und so der erste Thread eine falsche errno sieht.

All das ist für euch aber nicht mehr relevant, ihr könnt mit errno umgehen wie mit einem int.

Scheitert eine Bibliotheksfunktion, setzt sie `errno` auf einen Wert. Dabei sollte dokumentiert sein, welche Werte sie setzen kann, und zwar in Präprozessorsymbolen, die auch in `errno.h` definiert sind und die alle mit „E“ anfangen: `EPERM`, `ERANGE`, `EAGAIN` usf.

Will man einfach nur eine Fehlermeldung ausgeben, helfen `char *strerror(int errnum)` (aus `string.h`) und `void perror(char *msg)` (aus `stdio.h`). Erstere wandelt `errno` in einen (sprachspezifischen) String um, zweitere gibt die passende Fehlermeldung mit vorangestelltem `msg` aus.

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

int main(int argc, char **argv)
{
  setlocale(LC_ALL, "");
  if (!malloc(2000000000)) perror(argv[0]);
  if (!fopen("xxy.yxx", "r")) perror(argv[0]);
  if (!fopen("/etc/junk", "w")) perror(argv[0]);
  if (rename("errdemo.c", "/tmp/errdemo.c"))
    perror(argv[0]);
  return 0;
}
```

Ich setze hier voraus, dass ihr nicht wirklich 2 Gigabyte Speicher allozieren könnt, dass die Datei xxy.yxx nicht existiert, dass ihr nicht auf `/etc` schreiben könnt (das sollte besser so sein...) und dass /tmp und euer Arbeitsverzeichnis auf zwei verschiedenen Filesystemen liegen.

Dass perror immer argv[0] übergeben bekommt, liegt an der Konvention, dass Unix-Programme bei Fehlermeldungen möglichst den Namen des Programms, das den Fehler verursacht hat, mit ausgeben sollten. Weil das ein bisschen ein Tanz ist, wenn die Fehler in tieferen Funktionen auftreten (ihr wollt bestimmt nicht argv[0] durch alle Funktionen durchziehen), definiert die glibc (im Gegensatz zu den meisten anderen C-Bibliotheken) noch char *program_invocation_name und char *program_invocation_short_name, die global den vollständigen Pfadnamen und den Basisnamen des aufrufenden Programms enthalten. Es ist nicht schwierig, entsprechendes nachzuschreiben.

soft link

Das Ergebnis:

```
examples> ./errdemo
./errdemo: Cannot allocate memory
./errdemo: No such file or directory
./errdemo: Permission denied
./errdemo: Invalid cross-device link
examples> export LANG=de_DE
examples> ./errdemo
./errdemo: Nicht genügend Hauptspeicher verfügbar
./errdemo: Datei oder Verzeichnis nicht gefunden
./errdemo: Keine Berechtigung
./errdemo: Ungültiger Link über Gerätegrenzen
hinweg
```

66. Systemprogrammierung I: Verzeichnisse

Viele Dinge, die man eigentlich häufig braucht – z.B. Verzeichnisse auslesen, Prozesse erzeugen, Netzwerkverbindungen öffnen –, sind leider (wenigstens in C) nicht wirklich standardisiert. Auch wenn das nicht portabel zu machen ist, sind doch die Metaphern oft ähnlich. Deshalb diskutiere ich hier einige Techniken im Linux/glibc-Umfeld, die Übertragung auf andere Plattformen bleibt euch überlassen.

In Python sind die meisten dieser Funktioen im Modul os enthalten. Dort sorgt die Bibliothek für eine Umsetzung auf die für die jeweilige Plattform angemessenen Aufrufe, so dass viele der Aufgaben hier in Python portabel ausgeführt werden können.

Verzeichnisse

Kennt man den Namen einer Datei und will sie einfach löschen, gibt es dafür die Funktion remove(char *pathname) aus stdio.h.

Die Funktion remove ist portabel, d.h. in ANSI C definiert. Unter Unix hieß sie konventionell unlink (und ist so immer noch in unistd.h deklariert, tut aber exakt das Gleiche wie remove), weil die Datei nicht zwingend gleich gelöscht wird. Wie bei den Dateisystemen diskutiert, ist es bei Unix-Dateisystemen durchaus möglich, eine Datei zu löschen (d.h. ihren Verzeichniseintrag zu entfernen), ohne dass der zugehörige inode (oder was immer) freigegeben wird – etwa im Falle von hard links.

Warnung: *soft link*s sind quasi nur Dateien, in denen der Name der Zieldatei drinsteht. Wenn auf eine Datei, die ihr löscht, ein soft link existiert, kümmert das Unix nicht, es löscht die Datei, und der soft link zeigt ins Leere.

Das „Auslesen“ eines Directories geht ziemlich analog zu FILEs: Es gibt (aus dirent.h und ggf. sys/types.h)

- einen Typ DIR
- DIR *opendir(char *dirname)
- int closedir(DIR *dir)

- `struct dirent *readdir(DIR *dir)`

Es gibt dabei natürlich keine Mode-Flags in `opendir` (auf ein Verzeichnis schreiben ist reichlich sinnlos), und die Eingabe gibt keine chars oder char* zurück, sondern einen Pointer auf einen (statisch in DIR liegenden) struct. Diesen sollte man zwischen zwei Aufrufen von `readdir` ausgewertet haben, weil die Daten überschrieben werden, ganz wie in unserer Implementation von `getNextWord`.

In dem struct ist das Feld `char *d_name` definiert, der Name des Verzeichniseintrags, der gerade betrachtet wird. Die Reihenfolge, in der die Namen kommen, ist nicht definiert.

Ein einfaches Programm, das einen Verzeichnisinhalt ausgibt:

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

int main(int argc, char **argv)
{
  DIR *dir;
  struct dirent *entry;

  if (argc==2 && (dir=opendir(argv[1]))) {
    while ((entry=readdir(dir))) {
      puts(entry->d_name);
    }
    closedir(dir);
  } else {
    fprintf(stderr, "You messed it up.\n");
  }
  return 0;
}
```

Anlegen bzw. Löschen von Verzeichnissen mit
`mkdir(char *path)` bzw. `rmdir(char *path)`.

Dateien und Verzeichnisse lassen sich mit
`rename(char *oldname, char *newname)`

umbenennen. Dabei darf sich auch das Verzeichnis ändern, solange Quell- und Zielname auf demselben Filesystem liegen.

67. Systemprogrammierung II: Dateiattribute

Dateien haben unter modernen Betriebssystemen eine Unzahl von *Metainformationen* (d.h., Informationen über ihren eigentlichen Inhalt hinaus: Name, Zugriffsrechte, Änderungsdatum usf.). Zugriff auf sie bietet `stat` aus `sys.stat.h`. Es füllt eine `struct stat` aus. Aus ihr lassen sich u.a. entnehmen

- Typ und Zugriffsrechte (`st_mode`)
- Eigentümer (`st_uid`)
- Änderungszeit (`st_mtime`)
- Größe (`st_size`)

Die Zeiten sind dabei im üblichen Unix-Format kodiert, nämlich Zahl der Sekunden seit dem 1.1.1970. Damit kann man die Daten einfach wie ganze Zahlen vergleichen.

Der Eigentümer der Datei ist numerisch kodiert – um auf wirkliche Namen zu kommen, kann man die Funktion `getpwent` verwenden.

In `st_mode` ist zusätzlich kodiert, was für eine „Datei“ sich hinter dem Namen verbirgt. Makros für Zugriff auf `st_mode`:

- S_ISDIR(m) Wahr, wenn Verzeichnis
- S_ISLNK(m) Wahr, wenn symbolischer Link
- S_ISREG(m) Wahr, wenn normale Datei

Analog Makros für die verschiedenen Zugriffsrechte:

- S_IRUSR Lesen für Eigentümer
- S_IWUSR Schreiben für Eigentümer
- S_IXUSR Ausführen für Eigentümer
- S_IRGRP Schreiben für Gruppe
- S_IROTH Lesen für andere...

Mit diesen Makros und der Funktion `chmod(char *filename, mode_t mode)` können die Rechte für eigene Dateien auch gesetzt werden. Das mag z.B. praktisch sein, um der eigenen Gruppe auf jeden Fall Schreibrecht zu geben. In solchen Fällen kann man aber auch an den Einsatz von `mode_t umask` denken.

Im folgenden Beispiel werden zwei Funktionen definiert, die jeweils auf einzelne Felder einer stat-Struktur zugreifen. Im Hauptprogramm wird dann bestimmt, wie die Altersrelation zwischen zwei in der Kommandozeile erwähnten Dateien ist. Sowas könnte z.B. nützlich sein, um zu bestimmen, ob eine Datei neu erzeugt werden muss, ganz ähnlich, wie Make das macht.

Bei der Gelegenheit verwenden wir den einzigen ternären Operator von C, das Fragezeichen. Der Ausdruck `a?b:c` hat einfach den Wert `b`, wenn `a` wahr ist, und sonst den Wert `c` – wir haben hier also einen einfachen Spezialfall von Selektion. Laut Präzendenztabelle bindet ? lächerlich schwach, so dass hier Klammern fast immer angesagt sind.

```
#include <sys/stat.h>
#include <stdio.h>
#include <time.h>

size_t getFSize(char *name)
{
  struct stat s;
  if (stat(name, &s)) {
    perror(name); exit(1);
  }
  return s.st_size;
}

time_t getMTime(char *name)
{
  struct stat s;
  if (stat(name, &s)) {
    perror(name); exit(1);
  }
  return s.st_mtime;
}

int main(int argc, char **argv)
{
  if (argc==3) {
    time_t t1=getMTime(argv[1]),
      t2=getMTime(argv[2]);
    printf("%s (%d Bytes) ist %s als %s (%d Bytes).\n",
      argv[1], getFSize(argv[1]), (t1>t2?"neuer":"älter"),
      argv[2], getFSize(argv[2]));
  } else {
    return 1;
  }
  return 0;
}
```

68. Systemprogrammierung III: Signale

Signal
Handler
asynchron

*Signal*e teilen einem Programm mit, dass

- Ein Fehler aufgetreten ist (`SIGSEGV`, `SIGFPE`, ...)
- Ein externes Ereignis aufgetreten ist (`SIGHUP`, `SIGINT`, ...)
- Ein Prozess gerne etwas von uns hätte (`SIGTERM`, `SIGABRT`, `SIGCHLD`, ...)

Signale werden in der Regel duch ihre symbolischen Namen bezeichnet. Welche es da gibt, steht samt Erklärung bei signal(7). Die oben erwähnten bedeuten in etwa folgendes:

- `SIGSEGV`: „Segmentation Violation“, das Programm hat versucht, auf Speicher zuzugreifen, der ihm nicht gehört.
- `SIGFPE`: „Floating Point Exception“, bei einer mathematischen Operation (nicht notwendig Fließkomma) ist ein Fehler aufgetreten (z.B. Division durch Null)
- `SIGHUP`: „Hangup“, das Terminal, von dem aus der Prozess kontrolliert wird, hat „aufgelegt“, existiert also nicht mehr (Dämonen, die ohnehin kein kontrollierendes Terminal haben, benutzen das Signal gerne als „Konfiguration neu laden“)
- `SIGINT`: „Interrupt“, jemand hat Control-C (oder den Interrupt Key) gedrückt.
- `SIGTERM`: „Terminate“, jemand will, dass das Programm möglichst schnell den Geist aufgibt
- `SIGABRT`: „Abort“, jemand will, dass das Programm abbricht, wird meistens vom Programm selbst in Verbindung mit assert geworfen
- `SIGCHLD`: Ein Kindprozess ist fertig geworden.

Die Bibliothek richtet für jedes Signal einen *Handler* ein, der das Signal ignoriert, den Prozess beendet, einen core dumpt usf.

Es ist aber auch möglich, eigene Handler einzurichten. Dazu dient die Funktion
`void *signal(int sigNum, sighandler_t action)`

Sie nimmt in `sigNum` eine Signalnummer (eines der `SIGxxxx`-Symbole, in `action` entweder einen Signalhandler oder eines der Symbole `SIG_DFL` (Default wiederherstellen) oder `SIG_IGN` (Signal ignorieren). Von letzterem solltet ihr vorläufig die Finger lassen.

Die Funktion gibt das, was vorher als action gesetzt war, zurück. Auf diese Weise kann man den vorherigen Handler wiederherstellen, wenn das Signal keine spezielle Behandlung mehr braucht.

Signale können grundsätzlich *asynchron* kommen, d.h. zu einem beliebigen Zeitpunkt im Programmablauf. Damit ist man eigentlich schon im Bereich des concurrent programming, in dem allerlei Probleme auftreten, die mit dem gleichzeitigen Zugriff verschiedener Programme auf gemeinsame Ressourcen zusammenhängen – Schlagworte sind hier Reentranz, Race Conditions oder Locking. Generell sollten Laien nur die Signale `SIGHUP`, `SIGINT`, `SIGTERM` und evtl. `SIGQUIT` behandeln und in Signalhandlern nur einen Flag setzen (aber Vorsicht, vgl. unten), der dann im Hauptprogramm ausgewertet wird.

Wer mehr machen will, muss mindestens den Abschnitt über Signale in der glibc-Dokumentation lesen, besser noch den Stevens und ein Buch über Betriebssysteme.

Bei der Anwendung von signal kommt erschwerend hinzu, dass verschiedene Unices verschiedene Dinge tun, wenn ein Signalhandler aufgerufen wurde, das Programm aber weiterläuft. Manche richten danach wieder den Default-Handler ein, andere nicht. Bei rezeptgemäßer Anwendung muss euch dieser Unterschied nicht wesentlich kümmern – der schlimmstmögliche Fall ist, dass euer Signalhandler ein möglicherweise erneut auftretendes Signal nicht mehr fängt und dann nicht aufgeräumt wird. Im Zweifelsfall besser, aber auch komplizierter, ist die Anwendung von `sigaction`.

Häufige Verwendung: Aufräumen nach externem Programmabbruch, etwa nach folgendem Muster.

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

```
volatile sig_atomic_t done=0;

void handleTerm(int sig)
{
  done = 1;
}

int main(void)
{
  signal(SIGINT, handleTerm);
  while(!done) {
    printf("Still going\n");
    sleep(1);
  }
  printf("Cleanup.\n");
  return 0;
}
```

Wichtig hierbei vor allem die Definition von `done`. Erstens ist es volatile. Das ist nötig, weil der Compiler sonst in der Main-Schleife diagnostizieren könnte, dass `done` im Schleifenkörper nie verändert wird und die ganze Abfrage herausoptimieren. Generell: Was asynchron (d.h. außerhalb des normalen Programmablaufs) verändert werden kann, muss auch volatile sein.

Zweitens hat es den Typ `sig_atomic_t`. Im Groben ist dadurch garantiert, dass Zuweisungen zu Variablen dieses Typs nicht von Signalen unterbrochen werden können, was für andere Typen nicht sicher ist. Wenn aber eine Zuweisung unterbrochen wird und der Signalhandler selbst zuweist, ist das Ergebnis meistens Mist – stellt euch folgendes Szenario vor:

`strcpy(someStr, "abcdefg")` wird von einem Signal unterbrochen, gerade, wenn es das `c` geschrieben hat. Der Signalhandler führt `strcpy(someStr, "0000000")` aus und kehrt zurück. Das ursprüngliche strcpy macht beim `d` weiter, das Ergebnis ist, dass `someStr` den Wert `"000defg"` hat – weder das, was vom Hauptprogramm aus drinstehen sollte, noch das, was der Signalhandler reingeschrieben hat. Dies ist ein Spezialfall einer so genannten race condition.

Im oben vorgestellten Muster ist das aber auch ohne *atomare* Zugriffe kein Problem, weil nur der Signalhandler schreibend auf `done` zugreift.

Eine weitere für Laien „erlaubte“ Anwendung ist `alarm`, etwa für Timeouts.

Hierbei wird es aber schon etwas kitzlig. Relativ einfach ist es noch, wenn man das Programm einfach abbrechen will:

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

#define NAME_LEN 80

void handleAlarm(int sig)
{
  fprintf(stderr, "\nYou're sleeping.\n");
  exit(1);
}

int main(void)
{
  char name[NAME_LEN];
  void *oldAlrmHandler;

  oldAlrmHandler = signal(SIGALRM, handleAlarm);
```

```
  alarm(2);
  printf("Name: ");
  fgets(name, NAME_LEN, stdin);
  alarm(0);
  signal(SIGALRM, oldAlrmHandler);
  printf("Name: ");
  printf("\nHi, %s\n", name);
  return 0;
}
```

Hier wollen wir den alten Signalhandler restaurieren, merken uns also, was uns signal zurückgibt. Dann sagen wir, dass wir in zwei Sekunden geweckt werden wollen und warten auf eine Eingabe der Benutzerin. Wenn diese rechtzeitig kommt, löschen wir die Alarmanforderung (das tut das Argument Null) und restaurieren den Signalhandler, sonst kommt unser Handler ins Spiel.

Entgegen der Aussagen oben verwende ich hier durchaus Bibliotheksfunktionen. Das darf ich hier, weil ich sicher weiß, dass weder ein fprintf auf stderr noch ein exit aktiv sind, wenn der Handler aufgerufen wird – in gewissem Sinn kommt das Signal hier nicht asynchron, sondern irgendwie doch programmgesteuert.

Wenn das Programm nachher weiterlaufen soll, wirds komplizierter. Näheres dazu in der entsprechenden man- oder info-Seite.

In Python ist letzteres übrigens erheblich einfacher, weil man Exceptions hat:

```
import signal, sys

class WakeUp(Exception):
  pass

def raiseWakeUp(signo, stackFrame):
  raise WakeUp, "Ring Ring"

signal.signal(signal.SIGALRM, raiseWakeUp)
signal.alarm(2)
try:
  print "Name: ",
  name = sys.stdin.readline()
  print "Hi, %s"%name
except WakeUp:
  print "\nYou're sleeping!"
  sys.exit(1)
```

Man sieht, dass der Signalmechanismus in Python weitgehend ähnlich funktioniert wie der in C, nur der Signalhandler bekommt ein zusätzliches Argument. Durch die Exception können wir aber aus dem Signalhandler Nachrichten zurück an das Hauptprogramm schicken, ohne uns verrenken zu müssen.

Außerdem ist jede Python-Instruktion atomar, was einerseits gut ist, weil auf diese Weise viele der richtig saftigen Concurrency-Probleme gar nicht auftreten, andererseits aber auch blöd, weil Signale so lange warten, bis ein Block C-Code ausgeführt ist und der Interpreter wieder die Kontrolle hat. Es kann also manchmal sein, dass Signale in Python doch nicht ganz das tun, was wir von ihnen hätten.

SIGINT übrigens löst in Python einfach eine KeyboardInterrupt-Exception aus (wenn man keinen Signalhandler für INT installiert hat...).

69. Systemprogrammierung IV: Tochterprozesse

Koprozess

Häufig möchte man ein externen Programm aufrufen. einfachste Möglichkeit: `system(char *cmd)` aus `stdlib.h`; führt `cmd` aus wie die Shell, gibt Rückgabewert des externen Programms zurück.

Zusätzlich stdin *oder* stdout des erzeugten Prozesses bedienen: `FILE *popen(char *command, char *type)`, `type="r"` oder `"w"`. Schließen des erhaltenen `FILE`s mit mit `pclose(FILE *stream)` schließen, Ergebnis ist der Rückgabewert des aufgerufenen Programms.

Ein Beispiel dafür könnte sein, dass wir ein Jpeg-Bild schreiben wollen, aber zu faul sind, den Jpeg-Encoder per Bibliothek zu betreiben. Bei der offenen Jpeg-Bibliothek ist hingegen ein Programm namens cjpeg dabei, das die Quelldatei im pbm-Format liest (bei debian ist das im Paket libjpeg-progs, andere Distributionen werden ähnliche Namen haben; auch der Quellcode der Bibliothek[27] ist am Netz verfügbar). Ein pbm-Bild zu schreiben, ist trivial: Man gibt eine Signatur („P6“ steht für Farbdaten), dann Höhe und Breite des Bildes und den maximalen Wert, den jeder Farbkanal annehmen kann, jeweils in ASCII. Nach einem Zeilenvorschub kommen dann ganz einfach die Daten, hier je ein Byte für Rot, Blau und Grün.

Man könnte das über eine Zwischendatei und `system` verhandeln, aber Zwischendateien sind immer schwierig. Mit `popen` geht das relativ einfach:

```
#include <stdio.h>

int main(int argc, char **argv)
{
  FILE *output;
  int i, j, fail=0;

  if ((output=popen("cjpeg > colours.jpg", "w"))) {
    fprintf(output, "P6\n255 255 255\n");
    for (i=0; i<255; i++) {
      for (j=0; j<255; j++) {
        fprintf(output, "%c%c%c", i, j, 255);
    } }
    fail = pclose(output);
  } else { fail = 1;}
  if (fail) {
    perror(argv[0]);
  }
  return 0;
}
```

(cf. Fig. 16)

fork und exec

Sowohl `system` als auch `popen` sind relativ bequem, haben aber ein paar Probleme. Was ist z.B. wenn wir einen *Koprozess* wollen, bei dem *sowohl* stdin als auch stdout auf irgendwelche Deskriptoren von unserem Programm zeigen? Was ist, wenn wir den Tochterprozessen Signale schicken wollen? Erschwerend kommt hinzu, dass wir immer Sicherheitsprobleme haben, wenn eine Shell im Spiel ist – wenn unser Programm mehr Rechte hat als der Mensch, der sie aufruft (und das ist z.B. immer der Fall, wenn wir mit dem Netz reden), könnte er/sie durch gemeine Trickserei mit Umgebungsvariablen oder Parametern, die wir so übergeben, plötzlich Dinge mit den Rechten unseres Programms ausführen.

Kurz: Wer mehr will, muss auf die `popen` und `system` zugrunde liegenden Funktionen zurückgreifen: `fork` und `exec`.

Die Funktion `pid_t fork(void)` „verzweigt“ die Ausführung eines Programms – es entstehen zwei identische Prozesse. Das Kind sieht als Ergebnis von fork 0, das Elter die PID des Kindes.

PID steht dabei für Program Identification; dies ist eine Zahl (sichtbar z.B. in der Ausgabe von ps), die für jeden Prozess, der zu einer gegebenen Zeit auf einer Maschine läuft, eindeutig ist.

[27] `http://www.ijg.org/files/`

deadlock

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main(void)
{
  pid_t childPid;

  childPid = fork();
  if (childPid==0) { /* child */
    printf("Hier spricht das Kind.\n");
  } else {
    printf("Mein Kind heißt %d.\n", childPid);
  }
  return 0;
}
```

Um damit Dinge wie `system` machen zu können, braucht man noch Funktionen aus der exec-Familie und ggf. `wait`.

`fork` führt einfach das gleiche Programm zwei Mal aus. Das kann sinnvoll sein (z.B. ein Steuerprogramm, das immer weiter läuft und Arbeiterprogramme abspaltet, um ihre Ergebnisse in irgendeiner Weise einzusammeln oder auch nicht – apache z.B. arbeitet so). In der Regel möchte man aber andere Programme ausführen. Dazu kann man ein Prozessimage komplett durch ein anderes ersetzen, und zwar mit der Funktionen aus der exec-Familie – gelingt eine dieser Funktionen, so ist das „alte“ Programm komplett vergessen, d.h. exec kehrt wenn überhaupt nur bei einem Fehler zurück.

Der aufrufende Prozess kann dass mit `wait` – ähnlich wie `system` das tut – auf das Ende des Tochterprozesses warten – oder auch weiterlaufen. Wenn ein Tochterprozess terminiert, bekommt der Mutterprozess ein `SIGCHLD` geschickt, das er tunlichst annehmen sollte. Wird das Signal nicht verarbeitet (was bei Verwendung von wait nicht passieren kann), entstehen *Zombies*, Prozesse, die eigentlich nicht arbeiten, aber auch nicht sterben können, weil Unix noch wartet, bis der Mutterprozess anerkennt, dass sie gestorben sind. Sinn dieser Regelung ist, dass der Mutterprozess noch an Daten des Tochterprozesses interessiert sein könnte und diese nicht mehr zur Verfügung stehen, wenn der Prozess aufgeräumt wurde, wohl aber, solange er als Zombie eine unheilige Existenz fristet.

Auch dabei können, wenn die beiden Prozesse miteinander reden, viele Concurrency-Probleme auftreten, etwa *deadlock*s bei Koprozessen. Wenn beispielsweise der Tochterprozess darauf wartet, von der Mutter etwas zu lesen, die Mutter aber selbst etwas von der Tochter lesen möchte, bevor sie selbst schreibt, geht natürlich nichts mehr weiter. Es ist einfacher als man glaubt, solche Zustände zu erzeugen.

70. Systemprogrammierung V: File Descriptors

Unter Unix sehen fast alle Dinge wie Dateien aus: Dateien selbst, Netzwerkverbindungen, Geräte...
Für viele Aufgaben reicht aber stdio nicht aus.

File Descriptors

File Descriptor

pollen

Unterhalb der FILEs der C-Bibliothek liegen unter Unix Integers, die *File Descriptor*s. In jedem Prozess stehen 0 für stdin, 1 für stdout und 2 für stderr, weitere werden nach Bedarf vergeben. Die Funktion `int fileno(FILE *stream)` gibt den FD für einen FILE zurück.

Manipulation von FDs mit

```
int open(char *name, int flags);
int close(int fd);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, void *buf, size_t count);
FILE *fdopen(int fd, char *mode);
```

aus `unistd.h` oder `fcntl.h`. Die flags bei open sind ein wenig mit dem Modestring in fopen vergleichbar. Sie sind hier allerdings Integers, die zusammengeodert werden. Es gibt unter anderem

- `O_RDONLY`, `O_WRONLY`, `O_RDWR` für nur lesenden oder schreibenden oder zugleich lesenden und schreibenden Zugiff. Einer davon muss angegeben werden.
- `O_CREAT` – die Datei wird angelegt, wenn es sie noch nicht gibt
- `O_TRUNC` – wenn es die Datei schon gibt, wird sie vorm Schreiben gekürzt (es gibt auch `O_APPEND`, dann wird angehängt, ansonsten einfach überschrieben)
- `O_NONBLOCK` – wenn nicht sofort gelesen oder geschrieben werden kann, kommen read und write sofort zurück, auch ohne EOF (Non-Blocking I/O – sowas will man z.B. machen, um Deadlocks bei Koprozessen zu vermeiden). Vorsicht: Unix puffert normalerweise Ein- und Ausgaben, so dass dieser Flag allein noch nicht reicht, um etwas wie KeyPressed zu implementieren. Dafür sollte man eher Bibliotheken wie curses einsetzen.

Open gibt entweder eine positive Zahl zurück – eben den FD – oder aber -1 bei einem Fehler. Man kann open auch mit einem dritten Argument aufrufen, den anfänglichen Zugriffsrechten, die aus den mode-bits von stat zusammengeodert werden.

read und write geben jeweils die Zahl der gelesenen Zeichen zurück, oder -1 bei einem Fehler. Beim Zugriff auf nichtblockende FDs kann -1 zurückkommen, wenn keine Daten verfügbar sind, errno ist dann `EAGAIN`.

Die Funktion `fdopen` gibt einen FILE* für einen Deskriptor zurück. Das kann praktisch sein, wenn man irgendwelche low-level-Manipulationen an einem FD gemacht hat, danach aber wieder den Komfort von stdio haben will. Wichtig ist das vor allem beim Vererben von FILEs nach einem fork/exec: die FDs werden vom Elter auf das Kind vererbt, die Streams als Datenstrukturen der jeweiligen Programme verschwinden aber nach einem exec.

select

Manchmal will man auf Ereignisse in mehreren Datenströmen gleichzeitig reagieren. Dafür könnte man *pollen*, d.h. regelmäßig nachsehen, ob ein nichtblockendes read Daten liefert. Besser:

```
int select(int nfds, fd_set *rdfds, fd_set
  *wrfds, fd_set excfds, struct timeval *timeout)
```

Das folgende Programm demonstriert den Einsatz von rohen Files und select. Zunächst wird eine Funktion `cpChr` definiert, die Zeichen von einem Deskriptor auf einen anderen kopiert. Sie geht davon aus, dass die Deskriptoren nicht blocken und rechnet damit, dass zumindest read auch mal einen EAGAIN als Fehler setzt – in dem Fall ist natürlich keine Fehlermeldung angesagt.

Die main-Funktion besorgt sich zunächst die Deskriptoren für stdin und stdout und definiert Variablen vom Typ `fd_set` und `struct timeval`. Ersteres ist eine „Menge“ von Deskriptoren – wir brauchen das, um select mitzuteilen, auf welche FDs es aufpassen soll. Etwas weiter unten wird die so definierte Variable `inSet` mit einem Makro `FD_ZERO` zunächst „geleert“. Der Name des Makros suggeriert schon, dass `fd_set` so implementiert ist, wie wir bei den bitweisen Operatoren die Implementation von Mengen geplant hatten, nämlich als eine Art großen Integer, in dem

gesetzte Bits die Mitgliedschaft in einer Menge symbolisieren. Entsprechend setzen wir weiter unten die FDs, auf die select hören soll, mit einem Makro `FD_SET`.

Die Variable `timeout` vom Typ `struct timeval` ist demgegenüber eher schlicht. Sie soll eine Zeitspanne repräsentieren, die select warten soll, bis es erstmal aufgibt. Die Designer von select wollten die Möglichkeit eröffnen, nur Bruchteile von Sekunden zu warten, andererseits aber ggf. auch Stunden, und wollten keine Fließkommazahlen verwenden, weil das in Schnittstellen zum Betriebssystemkern unfein ist. So haben sie die Zeitspanne unterteilt in einen Sekundenanteil (`tv_sec`) und einen Anteil von Mikrosekunden (`tv_usec`). Dass select wirklich auf einer Zeitskala von Mikrosekunden genau ist, ist natürlich Unsinn.

Danach wird die Kommandozeile ausgewertet – das Programm erwartet zwei Dateinamen. Vom ersten liest es, auf das zweite schreibt es, und entsprechend werden die FDs per open erzeugt. Nach allen Präliminarien gehen wir in eine Endlosschleife, in der wir immer wieder timeout und inSet setzen (sie werden by reference an select übergeben, und in der Tat macht select sie auch kaputt), um dann select aufzurufen.

Select bekommt dann zunächst immer `FD_SETSIZE` übergeben – in diesem Makro steht letztlich, wie viele bits in einem `fd_set` sind, und darauf hat man keinen Einfluss. Dann kommen Pointer auf der Mengen von FDs – uns interessiert nur die erste, in der steht, auf welchen FDs select auf Eingabe warten soll. Im zweiten könnte man FDs übergeben, bei denen select aufpassen soll, wann eine Ausgabe passiert (z.B. interessant, wenn man wartet, bis ein Server am Netz Daten annimmt), im letzten könnten FDs übergeben werden, auf denen „etwas Besonderes“ passieren soll; darunter kann man sich Nachrichten wie „guck jetzt bitte ganz schnell hierher“ vorstellen.

Den Rückkehrwert von select (-1 bei einem Fehler, die Zahl der FDs, auf denen etwas passiert ist, sonst) ignorieren wir hier (sollten wir aber nicht tun), stattdessen werten wir aus, was select aus unserem inSet gemacht hat. Wir sehen mit `FD_ISSET` nach, ob einer der beiden FDs, die wir überwachen wollen, etwas unternommen hat und kopieren von ihm zu seiner entsprechenden Ausgabe, wenn das so ist. Haben beide nichts gesehen, ist offenbar der Timeout aktiv geworden, und wir geben einfach einen Punkt auf die Fehlerausgabe aus.

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <errno.h>

#define CP_SZ 80


int cpChr(int srcfd, int targfd)
{
  static char buffer[CP_SZ];
  int bytesRead;

  if ((bytesRead=read(srcfd, buffer, CP_SZ))==-1) {
    if (errno!=EAGAIN) perror("read");
  } else {
    if (bytesRead==0)
      exit(0);
    if (write(targfd, buffer, bytesRead)==-1) {
      perror("write");
    }
  }
  return 1;
}

int main(int argc, char **argv)
{
  int targwr, targrd;
  int srcwr=fileno(stdout), srcrd=fileno(stdin);
  fd_set inSet;
  struct timeval timeout={2, 0};
```

```
  if (argc!=3) { exit(1);}
  if (0>(targrd=open(argv[1], O_RDONLY|O_NONBLOCK))) {
    perror(argv[1]); exit(1);
  }
  if (0>(targwr=open(argv[2], O_WRONLY))) {
    perror(argv[2]); exit(1);
  }
  fcntl(targrd, O_NONBLOCK);
  FD_ZERO(&inSet);
  while (1) {
    timeout.tv_sec = 2;
    FD_SET(targrd, &inSet);
    FD_SET(srcrd, &inSet);
    if (!select(FD_SETSIZE, &inSet, NULL, NULL, &timeout))
      fprintf(stderr, ".");
    if (FD_ISSET(targrd, &inSet)) cpChr(targrd, srcwr);
    if (FD_ISSET(srcrd, &inSet)) cpChr(srcrd, targwr);
    }
  }
  return 0;
}
```

Was kann nun dieses Programm? Sehr wenig, zunächst. Um zu sehen, wozu es gut sein kann, könnt ihr zunächst zwei spezielle Einträge am Dateisystem machen, so genannte Fifos oder named pipes, etwa mit

```
mknod fif1 p
mknod fif2 p
```

Diese Fifos haben die Eigenschaft, dass man Daten in sie schreiben und sie auch wieder aus ihnen lesen kann, und zwar so, dass das, was zuerst reingeschrieben wurde, auch wieder zuerst rauskommt. Daher kommt auch der Name, Fifo steht für First in, First out, der Gegenbegriff ist Lifo (Last in, First out), die entsprechende Datenstruktur könnte etwa ein Stack sein.

Über diese Fifos können jetzt zwei Instanzen unseres Programms kommunizieren. Wenn ihr das Programm selectdemo genannt habt, könnt ihr jetzt in einem Fenster

```
selectdemo fif1 fif2
```

und im anderen

```
selectdemo fif2 fif1
```

laufen lassen. Das Ergebnis ist, dass alles, was ihr im einen Fenster tippt, im anderen erscheint und umgekehrt. Das Programm fummelt nicht an den Terminaleinstellungen herum, so dass ihr in der Regel erst Return drücken müsst, bevor die Mitteilungen auf der anderen Seite ankommen. Wer das ändern will, müsste sich mit der Dokumentation zu termios oder besser curses auseinandersetzen.

Das ist noch nicht so spannend. Das Modell trägt aber im Prinzip auch für Fälle, in denen Programm tatsächlich sinnvoll kommunizieren müssen (stellt euch etwa einen Lemmatisierer vor, dem man ein Wort in seine Fifo schreibt und der das zugehörige Lemma zurückgibt), und zwar, wie wir auf der nächsten Folie sehen werden, ggf. auch über Maschinengrenzen hinweg.

Eine andere Anwendung kann z.B. die Steuerung eines Modems sein – wenn ihr ein Hayes-kompatibles Modem an einer seriellen Schnittstelle habt, könnt ihr

```
selectdemo /dev/ttyS0 /dev/ttyS0
```

probieren (statt der 0 muss ggf. etwas anderes stehen, je nach dem, an welcher Seriellen das Modem hängt). Wenn ihr jetzt AT tippt, sollte das Modem OK zurücksagen, und wenn ihr `AT DP 06221 543248` tippt, pfeift euer Modem mir ins Ohr.

71. Systemprogrammierung VI: Sockets

Socket

Ein *Socket* dient der Kommunikation zwischen Prozessen. Anders als bei Pipes müssen die verschiedenen Prozesse aber nicht viel gemein haben, insbesondere nicht die Maschine, auf der sie laufen. Netzwerkkommunikation läuft fast immer über Sockets.

Dieses Feld ist weit – wir wollen nur mal sehen, wie man eine Netzwerkverbindung öffnet und damit spielt. Zunächst brauchen wir eine Funktion, die Hostnamen in Internetadressen „auflöst“:

```
void init_sockaddr(struct sockaddr_in *name,
  char *hostname, unsigned short port)
{
  struct hostent *hostinfo;

  name->sin_family = AF_INET;
  name->sin_port = htons(port);
  if (!(hostinfo = gethostbyname(hostname))) {
    perror(hostname); exit(1);
  }
  name->sin_addr = *(struct in_addr*)hostinfo->h_addr;
}
```

Die Idee dahinter ist, dass Menschen lieber Namen wie tucana.cl.uni-heidelberg.de haben als Zahlen wie 147.142.207.26. Der Rechner aber braucht die Nummern, um mit dem Rechner sprechen zu können. Für diese „Auflösung“ sorgt im Internet ein kompliziertes System namens DNS, auf anderen Netzwerken kann das auch etwas anderes sein, und auch die Adressen können dort anders aussehen als die 32-bit-Zahlen, die das Internet-Protokoll (Version 4) verwendet – schon in der Version 6 des Internet-Protokolls werden 128 bit lange Zahlen verwendet werden.

Wir beschäftigen uns hier nur mit dem „alten“ Internet (es ist nach wie vor schwierig, IPv6-Netze zu finden). In diesem Rahmen ist so eine Adresse in einer `struct sockaddr_in` untergebracht, und die Funktion oben füllt so eine Struktur, die drei Felder hat: `sin_family`, in der ein Symbol für den „Namensraum“ steht, hier nämlich `AF_INET` (eben das Internet); `sin_port`, eine Zahl, die einen „Port“ auf dem Zielrechner symbolisiert, quasi einen von vielen virtuellen Netzeingängen, die ein Rechner haben kann; und schließlich `sin_addr`, eben die IP-Nummer des Zielrechners, etwas wie 147.142.207.26.

Zum Setzen des Ports: IP sieht vor, dass jeder Rechner 65536 Ports haben kann – das sollten auch so viele sein, weil jede Netzwerkverbindung in der Regel auf einem eigenen Port läuft. Jedenfalls braucht man zwei Bytes, um das darzustellen, und damit hat man das Problem der Endianness (vgl. Folie „Pointer I“) – am Internet müssen Rechner verschiedener Endianness miteinander reden, und die Bytefolge 00 50 hex würde von den einen als 80 dez, von den anderen als 32768 dez interpretiert werden. Um dem vorzubeugen, reden Rechner am Netz immer in „Network Byte Order“. Die Funktion htons („Host to Net for Shorts“) besorgt gerade die Wandlung der Endianness der aktuellen Maschine zur Network Byte Order.

Die Funktion gethostbyname nimmt einen String wie tucana.cl.uni-heidelberg.de und gibt einen `struct hostent*` zurück. Diese hat ein Feld (`h_addr`), in dem die gesuchte numerische IP-Adresse steht, die wir mit einem passenden Cast in die `sockaddr_in`-Struktur schreiben können. Damit wissen wir, wohin wir uns verbinden wollen.

Die eigentlichen Sockets müssen zunächst mit der Funktion `socket` erzeugt und dann – im Beispiel – mit dem Zielrechner verbunden werden. Die Flexibilität der Sockets bringt mit sich, dass dabei viele Parameter im Spiel sind.

Im folgenden Programm ist unser (leicht modifiziertes) select-Beispielprogramm mit Sockets kombiniert. Das Wesentliche steht dabei in der Main-Funktion. Sockets haben dabei wie File Descriptors den Typ int, und wir erzeugen uns zunächst einen Socket. Als Argument übergeben wir `PF_INET`, was, analog zum `AF_INET` bei der Namensauflösung, bedeutet, dass dieser Socket ins Internet gehen soll – es wäre auch beispielsweise `PF_LOCAL` für Sockets auf der lokalen Maschine denkbar, dann müssten aber auch andere „Adressen“ (in dem Fall dann ein Pfad im Dateisystem) angegeben werden.

SOCK_STREAM bedeutet hier, dass wir einen „abgesicherten Strom“ haben möchten, der sich ähnlich wie eine Datei verhält. Auf der Netzwerkseite entspricht das dem TCP (Transmission Control Protocol), das richtige Verbindungen zwischen Rechnern macht – alternativ käme hier etwa SOCK_DGRAM in Frage, das nur einzelne Pakete („Datagramme“) verschickt und sich nicht weiter kümmert, ob sie auch ankommen; das entsprechende Protokoll auf der Netzwerkseite heißt dann UDP (User Datagram Protocol) und wird beispielsweise gern für Netzwerkdateisysteme wie NFS verwendet.

Das letzte Argument von socket würde die Auswahl verschiedener Protokolle für SOCK_STREAM oder SOCK_DGRAM erlauben – im Internet ist es aber unüblich, etwas von den beiden oben erwähnten Protokollen abweichendes zu verwenden, so dass die Null hier eine sichere Wahl ist.

Die eigentliche Verbindung öffenen wir mit connect, wenn wir aus dem ersten Argument eine numerische Adresse gemacht haben. Die Funktion connect nimmt dazu den Socket, die von sockaddr_in* auf das generischere sockaddr* zurückgecastete Adresse und die Größe der Daten, auf die dieser Pointer zeigt. Danach überlassen wir die Arbeit einer Funktion, die sich eng an unser select-Beispiel anlehnt.

```
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <fcntl.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define CP_SZ 80
#define HTTP_PORT 80

int copyChars(int srcfd, int targfd)
{
  static char buffer[CP_SZ];
  int bytesRead;

  if ((bytesRead=read(srcfd, buffer, CP_SZ))==-1) {
    if (errno!=EAGAIN) perror("read");
  } else {
    if (bytesRead==0) {
      exit(0);
    }
    if (write(targfd, buffer, bytesRead)==-1) {
      perror("write");
    }
  }
  return 1;
}


int communicate(int socket)
{
  int srcwr=fileno(stdout), srcrd=fileno(stdin);
  fd_set inSet;

  fcntl(srcrd, O_NONBLOCK);
  fcntl(socket, O_NONBLOCK);
  FD_ZERO(&inSet);
  while (1) {
    FD_SET(socket, &inSet); FD_SET(srcrd, &inSet);
    select(FD_SETSIZE, &inSet, NULL, NULL, NULL);
    if (FD_ISSET(socket, &inSet)) copyChars(socket, srcwr);
    if (FD_ISSET(srcrd, &inSet)) copyChars(srcrd, socket);
    }
  }
  return 0;
}
```

```
void init_sockaddr(struct sockaddr_in *name,
  char *hostname, unsigned short port)
{
  struct hostent *hostinfo;

  name->sin_family = AF_INET;
  name->sin_port = htons(port);
  if (!(hostinfo = gethostbyname(hostname))) {
    perror(hostname); exit(1);
  }
  name->sin_addr = *(struct in_addr*)hostinfo->h_addr;
}


int main(int argc, char **argv)
{
  int sock;
  struct sockaddr_in servername;

  if (argc!=2) return 1;
  if ((sock = socket(PF_INET, SOCK_STREAM, 0))<0) {
    perror("open socket"); return 1;
  }
  init_sockaddr(&servername, argv[1], HTTP_PORT);
  if (0>connect(sock, (struct sockaddr*)&servername,
   sizeof(servername))) {
    perror("connect"); return 1;
  }
  communicate(sock);
  close(sock);
  return 0;
}
```

Mit diesem Programm lassen sich schon ganz lustige Dinge tun, wenn man die Netzwerkprotokolle kenn, die wiederum auf TCP aufbauen. Im Beispiel verbinden wir fest mit Port 80, auf dem normalerweise ein Webserver lauscht (in der Tat heißt die Funktion, mit der man einen Server aufsetzt `listen` und wird in Servern statt connect verwendet). Im folgenden Beispiel lasse ich mir die Homepage des Instituts geben:

```
examples> socketdemo www.cl.uni-heidelberg.de
GET / HTTP/1.0

HTTP/1.1 200 OK
Date: Tue, 28 Jan 2003 13:36:58 GMT
Server: Apache/1.3.26 (Unix) Debian GNU/Linux
Connection: close
Content-Type: text/html

<HEAD>
<LINK href="http://www.cl.uni-heidelberg.de/style.css"
...
```

– in HTTP fragt erwartet der Server zunächst eine Zeile mit einer Anfrage, dann ggf. noch einige Key-Value-Paare (in denen der Client zum Beispiel angeben kann, welche Sprachen er gerne hätte) und dann eine Leerzeile, woraufhin der Server anwortet, wiederum mit einer Kopfzeile, in der das Ergebnis der Operation verkündet wird, einigen Key-Value-Paaren, in denen z.B. verraten wird, welche Sorte von Daten jetzt kommen (hier ist das HTML-Text) und schließlich nach einer Leerzeile wieder die Nutzdaten. Viele Protokolle im Netz sind syntaktisch ähnlich einfach gestrickt. Einen Server oder Client für die Protokolle zu schreiben, ist aber dennoch nicht ganz einfach, weil die Syntax eben nicht alles ist.

Signal

callbacks

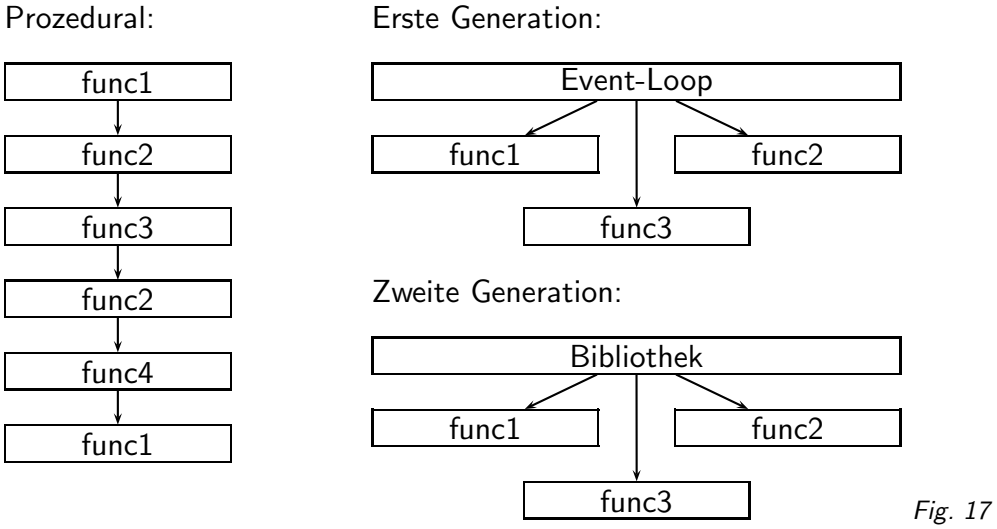


Fig. 17

72. GUI-Programmierung I

Grundlagen

Wie in Python greift man auch bei der GUI-Programmierung in C auf einen Toolkit zurück.

Zur Erinnerung:

(cf. Fig. 17)

Struktur eines GTK-Programms

1. Widgets des Hauptfensters anlegen
2. Callback-Funktionen für die Signale, die ein Widget emittiert, dem Widget zuordnen
3. Das Hauptfenster malen lassen
4. GTK den Rest der Arbeit überlassen

*Signal*e (sie entsprechen grob den Events von Tk) werden von Widgets losgeschickt, wenn auf/mit ihnen etwas passiert: Mausklick, Tastendruck, Fokus etc. Funktionen (*callbacks*) können mit solchen Signalen durch die Funktion
`gint gtk_signal_connect(widget, signalName, callback, data)`

verbinden lassen und werden dann aufgerufen, wenn das Signal geschickt wurde; `data` sind Hilfsdaten für den Callback.

„Die Arbeit überlassen“ geht in GTK durch Aufruf der Funktion `gtk_main()` – dies ist das Analogon zu Tks mainloop.

73. GUIs II: Ein erstes Programm

GUI-Programme, vor allem in prozeduralen Sprachen, neigen zur Verfettung. Folgende Funktion öffnet ein leeres Fenster:

```
GtkWindow *openMainWindow(char *title, int xSz, int ySz,
  GtkBox **mainBox)
{
  GtkWindow *window=GTK_WINDOW(
    gtk_window_new(GTK_WINDOW_TOPLEVEL));

  gtk_signal_connect(GTK_OBJECT(window), "destroy",
    GTK_SIGNAL_FUNC(gtk_main_quit), "WM destroy");
  gtk_window_set_title(window, title);
  gtk_widget_set_usize(GTK_WIDGET(window), xSz, ySz);

  *mainBox = GTK_BOX(gtk_vbox_new (FALSE, 1));
  gtk_container_border_width(
    GTK_CONTAINER(*mainBox), 1);
  gtk_container_add (GTK_CONTAINER(window),
    GTK_WIDGET(*mainBox));
  gtk_widget_show(GTK_WIDGET(*mainBox));
  gtk_widget_show(GTK_WIDGET(window));

  return window;
}

int main(int argc, char *argv[])
{ GtkWindow *window; GtkBox *mainBox;
  gtk_init(&argc, &argv);
  window = openMainWindow("Mein", 300, 200, &mainBox);
  gtk_main(); return 0;
}
```

Gtk ist natürlich nicht in der Standardbibliothek, und auch die Header sind üblicherweise nicht an den Stellen, die der Compiler ohne weiteres durchsucht. Daher braucht man Flags, die hier das Programm gtk-config ausspuckt. In einem Makefile könnte man dazu sagen

```
CFLAGS += ‘gtk-config --cflags‘
LDFLAGS += ‘gtk-config --libs‘
```

Beobachtungen

`gtk_init` wird mit den Kommandozeilenargumenten aufgerufen, damit gtk automatisch Optionen wie `--display` interpretieren kann.

Ein Window ist automatisch auch ein Widget (es ist „abgeleitet“). Wenn eine Funktion ein `GtkWidget` braucht, kann man durch Anwendung des Makros `GTK_WIDGET` einen `GtkWindow*` auf einen `GtkWidget*` casten.

Was auf was castbar ist, steht in einer Klassenhierarchie[28], wobei a auf b castbar ist, wenn a „unter“ b steht.

`GtkWindows` dürfen nur ein Widget einbetten (das ist eine Charakteristik der Oberklasse `GtkBin`). Deshalb packen wir gleich eine Box – `mainBox` – rein, die mehrere Widgets einbetten kann (`GtkBox` erbt direkt von `GtkContainer`).

Der Aufruf von `gtk_signal_connect` verbindet hier das Signal „zumachen“ mit einer Funktion, die den GTK-Event Loop beendet – ohne den Aufruf wäre das Fenster nicht zu schließen.

[28] `http://docs.cl.uni-heidelberg.de/gtk/gtk-index.html`

74. Struct Magic

Die Wandlung zwischen zwei Objekten ist einfach ein Cast – aus `GTK_BIN(foo)` wandelt der Präprozessor in `(GtkBin*)foo` um. Warum funktioniert das?

Aus den gtk-Headern:

```
#ifdef GTK_NO_CHECK_CASTS
#define GTK_CHECK_CAST(tobj, cast_type, cast) \
  ((cast*) (tobj))
#else /* !GTK_NO_CHECK_CASTS */
#define GTK_CHECK_CAST(tobj, cast_type, cast) \
      ((cast*) gtk_type_check_object_cast \
      ((GtkTypeObject*) (tobj), (cast_type)))
#endif /* GTK_NO_CHECK_CASTS */

#define GTK_BIN(obj) \
  (GTK_CHECK_CAST ((obj), GTK_TYPE_BIN, GtkBin))

struct _GtkObject {
  GtkObjectClass *klass;
  guint32 flags;
  guint ref_count;...
}

struct _GtkWidget {
  GtkObject object;
  guint16 private_flags;...
}

struct _GtkContainer {
  GtkWidget widget;
  GtkWidget *focus_child;...
}
```

Was passiert hier? Zunächst erbt eine „Klasse“ von der übergeordneten jeweils einfach durch „Einbetten“ des structs der „Oberklasse“ *an erster Stelle* in seinem eigenen struct, um danach die Felder zu definieren, die sie selbst braucht. Und das ist auch schon der ganze Trick, denn das Speicherlayout ist das gleiche, egal, ob der struct ausdefiniert ist oder nur eingebettet.

In einem einfacheren Beispiel:

```
struct _S1 {
  char b;
  int i;
}

struct _S2 {
  struct _S1 embedded;
  float extension;
}

...
struct _S1 s1; struct _S2 s2;
s1.b = 'c'; s1.i = 5;
s2.embedded.b = 'c'; s2.embedded.i = 5
s2.extension = 1e8;
```

Im Speicher sieht das dann etwa so aus:

(cf. Fig. 18)

(natürlich kann es auch irgendwie anders aussehen, der C-Standard schreibt die Details nicht vor, wohl aber, dass die Felder rechts und links gleich aufgeteilt sein müssen).

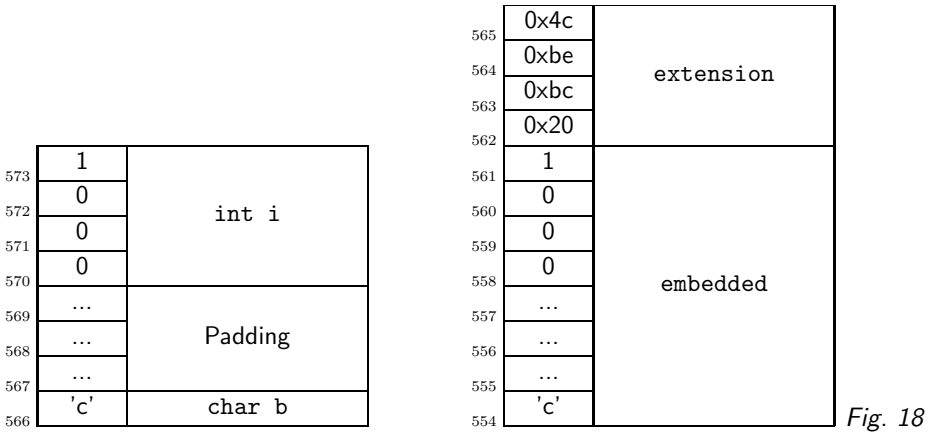


Fig. 18

Weil im Speicher das Gleiche steht, können wir C beruhigt vorspiegeln, dass in `s2` ein `struct _S1` steht. Umgekehrt geht das aber nicht, weil `((struct _S2*)&s1)->extension` auf Speicher zugreifen würde, der genau nicht zu `s1` gehört.

75. GUIs III: Ein Menü

Wir wollen eine Standard-Menüzeile haben. Die zweitbequemste Art, das in GTK zu machen, ist über eine Item Factory:

```
static void doLs(GtkWidget *w, gpointer data)
{
  if (data) system("ls -l");
  else system("ls");
}

GtkWidget *makeMenuBar(GtkWindow *window)
{
  static GtkItemFactoryEntry menu_items[] = {
    { "/_File", NULL, NULL, 0, "<Branch>"},
    { "/File/_Dir", "<control>d", doLs, 0, NULL},
    { "/File/_Long dir", "<control>l", doLs, 1, NULL},
    { "/File/_Quit", "<control>q", gtk_main_quit,
      0, NULL},
  };
  GtkAccelGroup *accel_group = gtk_accel_group_new();
  GtkItemFactory *item_factory = gtk_item_factory_new(
    GTK_TYPE_MENU_BAR, "<main>", accel_group);

  gtk_item_factory_create_items(item_factory,
    sizeof(menu_items)/sizeof(GtkItemFactoryEntry),
    menu_items, NULL);
  gtk_window_add_accel_group(window, accel_group);
  return gtk_item_factory_get_widget(item_factory,
    "<main>");
}
```

Fig. 19

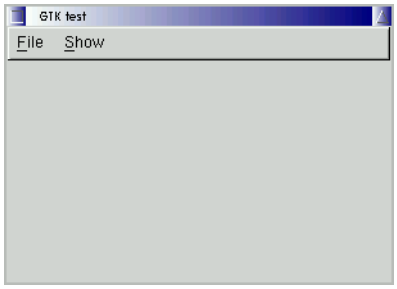


Fig. 20

(cf. Fig. 19)

(cf. Fig. 20)

Es wird ein Array von structs angelegt; in jedem struct steht vorne ein „Pfad“ zum Menüeintrag, dann ein Tastatur-Shortcut, ein Callback (vgl. doLs), zusätzliche Daten, die dem Callback übergeben werden sollen (hier zur Unterscheidung von ls und ls -l) und ein Typ, hier entweder `NULL` für einen Menüeintrag oder `<Branch>` für einen Menütitel.

Die Factory wird angewiesen, aus diesen Daten das Widget `<main>` zu erstellen. Das Ergebnis holen wir mit `gtk_item_factory_get_widget` und geben es zurück.

Das ganze Geeiere mit der `accel_group` ist nötig, weil die Tastaturereignisse vom Fenster und nicht von der Menüzeile empfangen und verarbeitet werden – sonst würden sie nur „gesehen“, wenn die Menüzeile den Fokus hat.

Dass bei der Konstruktion der `item_factory` gesagt werden muss, dass eine Menüzeile rauskommen soll, deutet schon an, dass die Funktion auch zu anderen, ähnlichen Zwecken verwendet werden kann. Solche Factories, die Objekte erzeugen, sind wieder mal klassische Patterns.

— Dateien zu diesem Abschnitt in der HTML-Version—

76. GUIs IV: Geometrie

Die Geometrie-Manager von Tk finden sich auch in Gtk wieder, und sie funktionieren auch sehr analog:

- place (absolute Positionen und Größen),
- pack (Widgets berühren sich) und
- grid (auch table oder ähnlich – Widgets werden an einem Gitter ausgerichtet)

Gtk definiert noch etliche weitere Geometrie-Manager, mit den erwähnten sollte man aber schon recht weit kommen.

Geburt eines Widgets:

1. Widget erzeugen (`gtk_xxx_new`)
2. Callbacks einrichten (`gtk_signal_connect`)
3. Widget einem Geometriemanager unterstellen (etwa `gtk_box_pack_start`)
4. Widget anzeigen (`gtk_widget_show`)

Zum Beispiel:
```
GtkWidget *button =
  gtk_button_new_with_label("Directory");
gtk_signal_connect(GTK_OBJECT(button), "pressed",
  GTK_SIGNAL_FUNC(doLs), 0);
gtk_box_pack_start(targetBox, button, 1, 1, 0);
gtk_widget_show(button);
```

Was soll der Geometrie-Manager tun, wenn das Widget wachsen sollte?

- Mehr Platz geben (`expand`)?
- Das Widget mitwachsen lassen (`fill`)?

Dementsprechend der Prototyp:
```
void gtk_box_pack_start(GtkBox *parent,
  GtkWidget *child, gboolean expand,
  gboolean fill, guint padding);
```

`parent` ist das Widget, in das `child` gepackt wird (der *Container*), `padding` lässt zusätzlichen Platz um das Widget.

Buttons

Buttons können genau ein weiteres Widget enthalten (etwa ein Bild). Normalfall ist aber ein `label`, deswegen spezielle Funktion:
```
GtkWidget*  gtk_button_new_with_label (const
  gchar *label)
```

Das interessanteste Signal, das Buttons senden, ist `pressed` – der Knopf wurde gedrückt.

Fig. 21

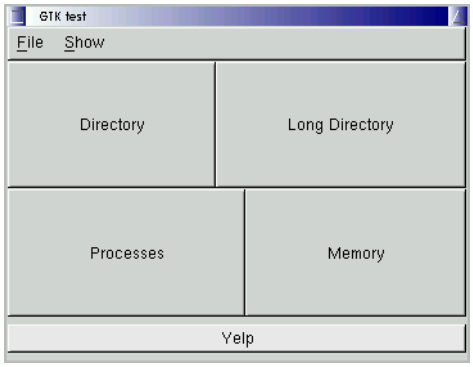


Fig. 22

77. GUIs V: Unfug

So wirkt der Pack-Geometrie-Manager, wenn alle Widgets außer dem Quit-Button mit fill und expand auf 1 gepackt wurden, der Quit-Button aber in die (vertikale) `mainBox` mit fill und expand auf 0 und mit padding 5 gepackt wurde (wenn der Quit-Button grundsätzlich seine natürliche Größe haben soll, müsste er noch in eine horizontale Box gepackt werden).

(cf. Fig. 21)

(cf. Fig. 22)

Widgets können viele Signale verschicken, und damit kann allerlei Unfug getrieben werden – Buttons, die vorm Mauszeiger fliehen etwa.

Ein harmloses Beispiel mit `enter` und `leave`: Ein Button, der seinen Text verändert, wenn der Mauszeiger darübersteht:

```
static void changeText(GtkWidget *w,
  gpointer data)
{
  GtkLabel *label = GTK_LABEL(GTK_BIN(w)->child);
  gtk_label_set_text(label, (gchar*)data);
}

GtkWidget *makeFancyQuitButton(void)
{
  GtkWidget *button =
    gtk_button_new_with_label("Quit");
  gtk_signal_connect(GTK_OBJECT(button),
    "pressed",GTK_SIGNAL_FUNC(gtk_main_quit), 0);
  gtk_signal_connect(GTK_OBJECT(button), "enter",
    GTK_SIGNAL_FUNC(changeText), "Yelp");
  gtk_signal_connect(GTK_OBJECT(button), "leave",
    GTK_SIGNAL_FUNC(changeText), "Quit");
  return button;
}
```

Wie auch sonst oft gilt aber am GUI-Design: Weniger ist mehr.

Ein Programm, das alle hier diskutierten Fragmente zusammenfasst, gibt es auf der Kursseite[29].

78. Mahnende Worte

5.10.1960 – Alarm bei NORAD: Ein neu installiertes Frühwarnsystem auf Grönland meldet mit 0.01% Irrtumswahrscheinlichkeit einen massiven Angriff sowjetischer Raketen. Zum Glück war Chruschtschow gerade in New York, denn in Wirklichkeit hatten die Programmierer des Systems nur vergessen, dass auch der Mond Radarstrahlen reflektiert.

9.11.1979 – Alarm bei NORAD: Das Frühwarnsystem meldet einen massiven Angriff sowjetischer Raketen, strategische Bomber starten. Glücklicherweise wird rechtzeitig bemerkt, dass die Eingabedaten des Systems gerade von einem Band mit Simulationsdaten kamen.

3.6.1980 – Alarm beim SAC: Sporadische Meldungen über sowjetische Raketen im Anflug. Die B52s lassen in Motoren an. Die Meldungen verschwinden wieder, nach drei Minuten wird der Alarm abgestellt. 6.6. – die Ereignisse wiederholen sich. Schließlich wurde der Grund gefunden: Ein Chip, der Nullen senden sollte, um den Leitungszustand zu prüfen, hatte versagt und schickte zufällige Zahlen.

Natürlich haben die meisten Programme bei weitem nicht die Möglichkeiten, Dinge kaputt zu machen, die die bei NORAD und ähnlichen Einrichtungen laufenden haben. In der Tat spielten bei den großen Industriekatastrophen des späten 20. Jahrhunderts von Bhopal bis Tschernobyl Computer in der Regel (und nach unserem Wissen) eine untergeordnete Rolle – aber dies wird sich ändern.

Auch so sind die Schäden durch Computerfehler bereits beträchtlich. Bereits 1984 – als am 26.7. in den USA der erste bekannte durch einen Industrieroboter verursachte Todesfall eintrat – schätzte man die Zahl der Roboteropfer in Japan auf fünf bis zwanzig.

Ein kleiner Fehler in einem Steuerungsprogramm für ein Bestrahlungsgerät kostete 1986 mindestens zwei Menschen das Leben – in einer selten auftretenden Situation „vergaß“ die Steuerung, die Stärke eines Elektronenstrahls zurückzuregeln. Sie merkte das sogar, doch die Fehlermeldungen waren so kryptisch und ähnelten „harmlosen“ Fehlermeldungen, die das System permanent ausspuckte, das die OperatorInnen sie ignorierten.

Weniger ernst – aber letztlich auch ein Problem einer nicht ausgegorenen Benutzerschnittstelle – war ein Absturz der Kurse für 10-jährige Französische Staatsanleihen am 23.7.1998: Der Grund war einfach, dass ein Broker in London seine Kaffetasse auf die „Instant Sell“-Taste gestellt hatte und diese 145 Signale abgegeben hatte.

In Flugzeugen gehen die Lichter aus, Autos bleiben einfach so stehen, die Universität weiß nicht mehr, was sie tut: Je mehr Prozesse von Programmen gesteuert werden, desto mehr kleine und große Katastrophen werden von Programmierfehlern ausgelöst. Was immer man programmiert, Fragen wie „Muss das sein?“ oder „Was passiert, wenns schief geht?“ sollte man stellen.

Mehr Risiken: risks digest[30].

Nicht nur Denk- oder Programmierfehler machen Computer gefährlich. In der Produktion sorgt wachsende Automatisierung dafür, dass es praktisch nur noch schechtbezahlte Putzkräfte und Security sowie eine Handvoll Ingenieure gibt – im Groben eine Abwertung von Jobs.

Eine geradezu klassische Variante davon sind Scannerkassen – diese wurden nicht etwa eingeführt, um den Job von KassiererInnen leichter zu machen. Teilweise war das Interesse sicher, die Warenhaltung zu erleichtern, indem der Rechner einen besseren Überblick über das Ein und Aus im Lager bekommt, teilweise vielleicht auch, die Fehlerquote zu reduzieren; der primäre Effekt ist aber, dass KassiererInnen austauschbarer und damit auch erpressbarer geworden sind, da Tippgeschwindigkeit und Überblick über die Preise im Laden nicht mehr gefragt sind. Musste man früher Personal wochenlang einarbeiten, muss man jetzt nur noch erklären, wie Barcodes über

[29] `http://www.cl.uni-heidelberg.de/kurs/ws02/prog2/gtktest.c`
[30] `http://catless.ncl.ac.uk/Risks`

den Scanner zu ziehen seien. Was die Konsequenzen einer solchen Entwicklung für Jobsicherheit und Lohnniveau sind, muss nicht erst ein aufwändiges Programm berechnen...

Wenn Sprachtechnologie mal funktionieren sollte, wird es eine ähnliche Entwicklung in Büros geben. Dazu kommen Datenschutzfragen. Hörende Computer können flächendeckende Telefonüberwachung machen, weit mehr als das Keyword Spotting, das gegenwärtig schon läuft. Data Mining in der Datenspur, die wir alle hinterlassen – von Stadtbücherei bis Reisebüro – wird viel effektiver werden.

Schließlich: Perfekte Sprachtechnologie setzt vermutlich „bewusste" Maschinen voraus. Darf man so eine Maschine noch abschalten?

```
main() {printf(&unix["\021%six\012\0"],
  (unix)["have"]+"fun"-0x60);}
        - David Korn, AT&T Bell Labs
```