Gepufferte Ein- und Ausgabe ist sinnvoll, um die Zahl der Systemaufrufe zu reduzieren. Standardmäßig wird in C als Lösung hierfür die *stdio* verwendet.

Frage: Ist *stdio* gut genug für alle Bedürfnisse?

Antwort: Nein, da sie keine Funktion anbietet, die analog zu dem Systemaufruf *read* sich darauf beschränkt, genau das zu liefern, was bereits ohne Zeitverzug vorliegt.

Ja, das ist in der Tat ein Problem beim Einlesen über das Netzwerk und auch das Einlesen, das mit regulären Ausdrücken gesteuert wird.

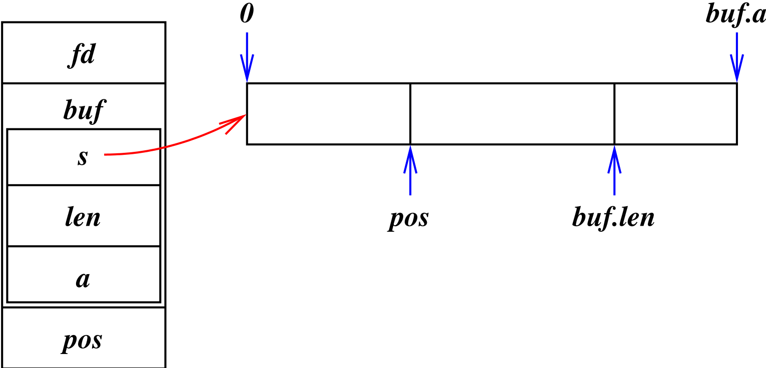
Was ist minimal notwendig für eine gepufferte Eingabe?

- Ein Puffer mit Informationen, wie groß dieser ist, wieweit dieser gefüllt wurde und bis wohin dieser gelesen worden ist.
- Eine Funktion, die einen leeren Puffer auffüllt mit genau einem Systemaufruf von *read*.
- Eine Funktion analog zu *read*, die, falls notwendig, den leeren Puffer auffüllt, und dann den Aufrufer nur aus dem gefüllten Puffer bedient.

- Entsprechend ist eine gepufferte Eingabe notwendig, bei der die Eingabe-Operationen aus einem Puffer versorgt werden, der, wenn er leer wird, mit Hilfe einer *read*()-Operation aufzufüllen ist.
- Die Datenstruktur für einen Eingabe-Puffer benötigt entsprechend einen Dateideskriptor, einen Puffer und einen Positionszeiger innerhalb des Puffers:

inbuf.h

```
typedef struct inbuf {
   int fd;
   stralloc buf;
   unsigned int pos;
} inbuf;
```


fd
buf
s
len
a
pos
0
buf.a
pos
buf.len

inbuf.h

```
#ifndef INBUF_H
#define INBUF_H

#include <stralloc.h>
#include <unistd.h>

typedef struct inbuf {
   int fd;
   stralloc buf;
   unsigned int pos;
} inbuf;

/* set size of input buffer */
int inbuf_alloc(inbuf* ibuf, unsigned int size);

/* works like read(2) but from ibuf */
ssize_t inbuf_read(inbuf* ibuf, void* buf, size_t size);

/* works like fgetc but from ibuf */
int inbuf_getchar(inbuf* ibuf);

/* move backward one position */
int inbuf_back(inbuf* ibuf);

/* release storage associated with ibuf */
void inbuf_free(inbuf* ibuf);

#endif
```

inbuf.c

```
/* set size of input buffer */
int inbuf_alloc(inbuf* ibuf, size_t size) {
   return stralloc_ready(&ibuf->buf, size);
}

/* works like read(2) but from ibuf */
ssize_t inbuf_read(inbuf* ibuf, void* buf, size_t size) {
   if (size == 0) return 0;
   if (ibuf->pos >= ibuf->buf.len) {
      if (ibuf->buf.a == 0 && !inbuf_alloc(ibuf, 512)) return -1;
      /* fill input buffer */
      ssize_t nbytes;
      do {
         errno = 0;
         nbytes = read(ibuf->fd, ibuf->buf.s, ibuf->buf.a);
      } while (nbytes < 0 && errno == EINTR);
      if (nbytes <= 0) return nbytes;
      ibuf->buf.len = nbytes;
      ibuf->pos = 0;
   }
   ssize_t nbytes = ibuf->buf.len - ibuf->pos;
   if (size < nbytes) nbytes = size;
   memcpy(buf, ibuf->buf.s + ibuf->pos, nbytes);
   ibuf->pos += nbytes;
   return nbytes;
}
```

inbuf.c

```
/* works like fgetc but from ibuf */
int inbuf_getchar(inbuf* ibuf) {
   char ch;
   ssize_t nbytes = inbuf_read(ibuf, &ch, sizeof ch);
   if (nbytes <= 0) return -1;
   return ch;
}

/* move backward one position */
int inbuf_back(inbuf* ibuf) {
   if (ibuf->pos == 0) return 0;
   ibuf->pos--;
   return 1;
}

/* release storage associated with ibuf */
void inbuf_free(inbuf* ibuf) {
   stralloc_free(&ibuf->buf);
}
```

- Reguläre Ausdrücke spezifizieren Sprachen, die mit endlichen Automaten syntaktisch anaylisiert werden können.
- Mit regulären Ausdrücken bzw. regulären Sprachen lassen sich die Grundsymbole einer Programmiersprache oder eines Eingabeformats spezifizieren wie z.B. das Format für eine Konstante in Gleitkommadarstellung.
- Reguläre Ausdrücke werden seit frühesten Zeiten in UNIX unterstützt: *ed*, *sed*, *grep*.
- Reguläre Ausdrücke sind im Rahmen des POSIX-Standards standardisiert: `http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap09.html`
- Reguläre Ausdrücke wurden von Perl deutlich erweitert und diese Erweiterungen werden dank des Erfolgs der PCRE-Bibliothek von vielen Werkzeugen unterstützt: `http://www.pcre.org`

Kleine Übersicht der Metazeichen:

„\“	nimmt dem folgenden nicht-alphanumerischen Zeichen die Sonderfunktion.
„^“	Anfang der Zeile.
„$“	Ende einer Zeile.
„.“	Beliebiges Zeichen außer dem Zeilentrenner.
„|“	Alternation. Beispiel: „foo|bar“ trifft sowohl für „foo“ als auch „bar“ zu.
„()“	Gruppierung. Beispiel: „(foo|bar)*“ trifft für „“, „foo“, „bar“, „foofoo“, „barfoo“, „barbar“ usw. zu.
„[]“	Zeichenbereiche. Beispiel: „[A-Za-z]“ (alle Klein- und Großbuchstaben).

- Dies entspricht dem Umfang von *egrep*.
- Im Gegensatz zu *vim* gelten die Klammern als Metasymbole.
- Es empfiehlt sich, alle nicht-alphanumerischen Zeichen, die sich selbst repräsentieren sollen, durch ein „\“ zu schützen.
- „\“ gefolgt von einem alphanumerischen Zeichen nimmt nicht dem alphanumerischen Zeichen seine (nicht vorhandene) Sonderbedeutung, sondern – ganz im Gegenteil – verleiht eine Sonderbedeutung (siehe noch folgende Übersicht).

Quantifikatoren beziehen sich auf das Zeichen, die Gruppierung oder den Zeichenbereich, der dem Quantifikator vorausgeht.

„*“	0 bis beliebig oft.
„?“	Optional: 0 oder 1 mal.
„+“	Mindestens einmal, ansonsten beliebig oft.
„$\{n\}$“	Genau n mal.
„$\{n,\}$“	Mindestens n mal.
„$\{n,m\}$“	Zwischen n und m mal.

- Normalerweise sind reguläre Ausdrücke „gierig“, das heisst sie versuchen, soviel Text wie möglich zu erfassen.
- Perl bzw. Perl-kompatible reguläre Ausdrücke unterstützen auch noch die Angabe eines „?“ hinter einem Quantifikator, um eine minimale Erfassung zu erreichen.

Perl bzw. Werkzeuge mit perl-kompatiblen regulären Ausdrücken unterstützen u.a. folgende Sonderzeichen (siehe *man perlre*):

„\t“	Tabulator.
„\n“	Zeilentrenner.
„\r“	Wagenrücklauf (CR).
„\f“	Papiervorschub (FF).
„\a“	Alarmglocke (BEL).
„\e“	Fluchtzeichen (ESC).
„\033“	Oktaldarstellung eines Zeichens.
„\x1b“	Hexdarstellung eines Zeichens.
„\cx“	Control-X.
„\N{*name*}“	Benanntes Zeichen, siehe *perldoc charnames*.
„\l“	Folgendes Zeichen in einen Kleinbuchstaben verwandeln.
„\u“	Folgendes Zeichen in einen Großbuchstaben verwandeln.
„\L“	Bis zu „\E“ alles in Kleinbuchstaben verwandeln.
„\U“	Bis zu „\E“ alles in Großbuchstaben verwandeln.
„\Q“	Bis zu „\E“ allen Metazeichen die Sonderbedeutung nehmen.

Perl und perl-kompatible Werkzeuge unterstützen diverse Kurzformen. Eine Auswahl:

„\w“	Erfasst alle alphanumerischen Zeichen einschließlich dem Unterstrich „_“.
„\W“	Erfasst alle Zeichen, die „\w“ nicht erfasst.
„\s“	Erfasst alle Leerzeichen.
„\S“	Erfasst alle Zeichen, die nicht zu den Leerzeichen gehören.
„\d“	Alle Ziffern.
„\D“	Alle Zeichen, die nicht zu den Ziffern gehören.

- Mit „\w“ wird noch kein Wort, sondern nur ein Zeichen eines Wortes erfasst. Jedoch lässt sich ersteres leicht mit „\w+“ erreichen.

```
#include <regex.h>

int regcomp(regex_t *restrict preg, const char *restrict pattern,
       int cflags);
size_t regerror(int errcode, const regex_t *restrict preg,
       char *restrict errbuf, size_t errbuf_size);
int regexec(const regex_t *restrict preg, const char *restrict string,
       size_t nmatch, regmatch_t pmatch[restrict], int eflags);
void regfree(regex_t *preg);
```

- Der POSIX-Standard unterstützt diese Funktionen.
- Mit *regcomp* wird ein regulärer Ausdruck übersetzt, mit *regexec* ausgeführt.
- *regexec* erwartet in *string* den vollständigen Text, auf den der reguläre Ausdruck bezogen wird.
- Es gibt hier keine teilweise Erfassung eines regulären Ausdrucks mit möglicher Fortsetzung.

mygrep5.c

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

- Reguläre Ausdrücke müssen zuerst mit *regcomp* in eine interne Datenstruktur (den endlichen Automaten) übersetzt werden.

mygrep5.c

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

- Anschließend kann mit *regexec* der Automat auf eine Vielzahl von Eingaben verwendet werden.

- Da die Erweiterungen von Perl für reguläre Ausdrücke sehr populär sind, wurde die entsprechende Bibliothek als selbständige Lösung herausfaktorisiert und weiter entwickelt, so dass sie auch für andere Anwendungen genutzt werden kann.
- Die PCRE-Bibliothek unterstützt partielle Erfassungen.
- `http://www.pcre.org`

Der prinzipielle Algorithmus:

1. Ein temporärer Puffer wird angelegt (als *stralloc*-Objekt),
2. Der Eingabe-Puffer wird gefüllt, falls er leer sein sollte.
3. Der gesamte Inhalt des Eingabe-Puffers wird an den temporären Puffer angehängt.
4. Der temporäre Puffer wird *pcre_exec* übergeben mit der Option *PCRE_PARTIAL_HARD*.
5. Im Erfolgsfalle lässt sich ermitteln, wieviel von der Eingabe von dem regulären Ausdruck erfasst wurde. Die Einlese-Position wird entsprechend weitergesetzt und das Einlesen ist beendet.
6. Wenn *pcre_exec* mit einem anderen Fehler als *PCRE_ERROR_PARTIAL* endet, wurde der reguläre Ausdruck in der Eingabe nicht erkannt. Entsprechend wird das Einlesen mit Fehler abgebrochen.
7. Beim Fehler *PCRE_ERROR_PARTIAL* wird der gesamte Eingabe-Puffer als gelesen markiert und die Verarbeitung mit Schritt 2 fortgesetzt.

Schnittstelle von *inbuf_scan*

```
#include <afblib/inbuf_scan.h>

int inbuf_scan(inbuf* ibuf, const char* regexp, ...);

typedef struct {
   const char* captured;
   size_t captured_len;
   int callout_number;
} inbuf_scan_callout_block;

typedef int (*inbuf_scan_callout_function)(inbuf_scan_callout_block*,
      void* callout_data);

int inbuf_scan_with_callouts(inbuf* ibuf, const char* regexp,
      inbuf_scan_callout_function callout, void* callout_data);
```

- „. . . “ steht für beliebig viele *stralloc*-Objekte, in denen die Eingabe abgelegt wird, die mit geklammerten Teilausdrücken im regulären Ausdruck erfasst worden sind.
- Zurückgeliefert wird −1 im Fehlerfall und die Zahl der gefüllten *stralloc*-Objekte im Erfolgsfall.

Lies eine Eingabezeile und entferne den Zeilentrenner:

```
stralloc line = {0};
int count = inbuf_scan(&ibuf, "(.*)\n", &line);
```

Analog, aber führende und abschließende Leerzeichen werden zusätzlich entfernt:

```
int count = inbuf_scan(&ibuf, "[ \t]*(.*?)\\s*?\n", &line);
```

Lies die letzte Zeile der Eingabe, überlies alles andere:

```
int count = inbuf_scan(&ibuf, "(?:.*\n)*(.*)\n", &line);
```

Nachteil bzw. Verbesserungsmöglichkeit:

- Inkrementelle Aufrufe von *pcre_exec* stehen leider noch nicht zur Verfügung. Entsprechend muss im temporären Puffer der gesamte bislang gelesene Text gesammelt werden. Dies sollte unnötig sein.
- Leider besteht *pcre_exec* auch bislang noch darauf, dass der gesamte zu untersuchende Text in einem zusammenhängenden Puffer steht.
- Wenn ein zusammenhängender Puffer nicht mehr Voraussetzung wäre, dann könnte es hilfreich sein, mit Pufferketten zu operieren. Auf Systemaufrufebene besteht durchaus Unterstützung von Pufferketten bei den Systemaufrufen *readv* und *writev*.
- Schließlich wäre auch ein Handler bei *pcre_exec* denkbar, der jedesmal aufgerufen wird, wenn mehr Eingabe benötigt wird.

Dann könnte das Umkopieren in einen sich sukzessive vergrößernden temporären Puffer entfallen und die Verabeitung wäre schneller mit Aufwand $O(n)$.