Vorlesung Informatik 1 (Wintersemester 2020/2021)

Kapitel 7: Benutzereingaben

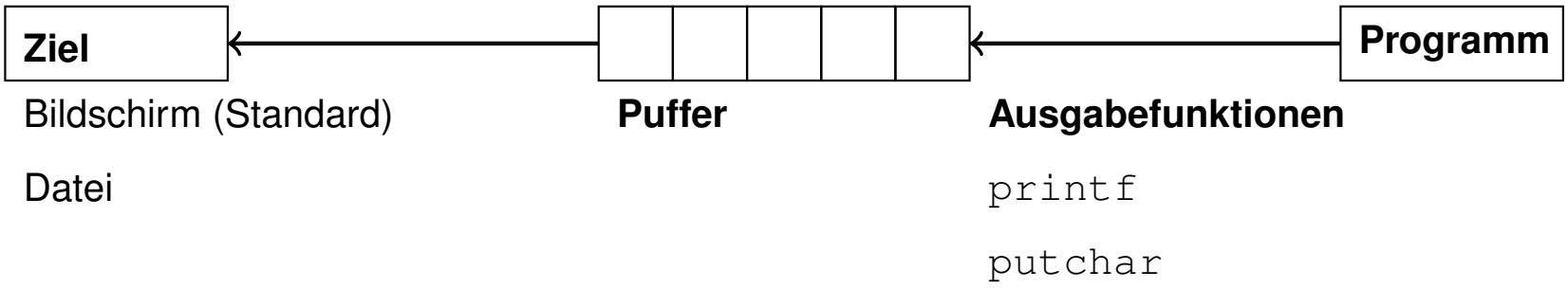
Martin Frieb
Johannes Metzger

Universität Augsburg
Fakultät für Angewandte Informatik

7. Dezember 2020

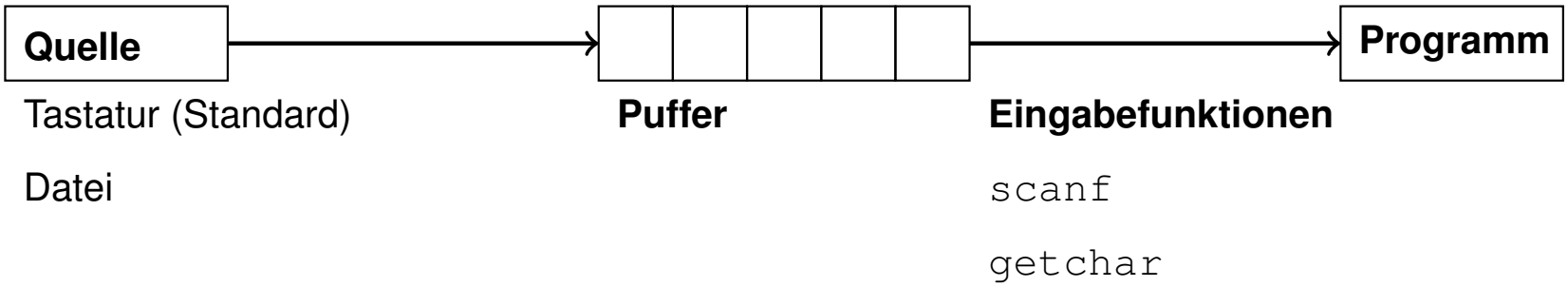
Universität Augsburg
Institut für Informatik

Standardausgabe



- Mit den Ausgabefunktionen `printf` und `putchar` werden die ausgegebenen Zeichen im Arbeitsspeicher in einem Puffer abgelegt
- Vom Puffer werden die Zeichen in der ausgegebenen Reihenfolge an das Ausgabeziel übertragen
- Bei der **Standardausgabe** ist das Ausgabeziel der Bildschirm
- Man kann die Standardausgabe auch in eine Textdatei umleiten (späteres Unterkapitel)

Standardeingabe



- Bei der **Standardeingabe** ist die Eingabequelle die Tastatur
- Von der Quelle werden die eingegebenen Zeichen in der eingegebenen Reihenfolge in einen Puffer im Arbeitsspeicher übertragen
- Mit den Eingabefunktionen `scanf` und `getchar` werden die Zeichen im Puffer in der eingegebenen Reihenfolge in Programmvariablen gespeichert
- Man kann die Standardeingabe umleiten, s.d. die Eingabe aus einer Textdatei erfolgt (späteres Unterkapitel)

Der Fehlerwert `EOF` (End of File)

Pufferfehler

- Bei der Ein- und Ausgabe von Zeichen kann es zu sog. **Pufferfehlern** kommen: Dateiende erreicht (Eingabe aus Datei), Festplatte voll (Ausgabe in Datei), ...
- Ein- und Ausgabefunktionen geben in solchen Fällen spezielle Fehlerwerte zurück
- **Ausnahme**: die Standardein- und -ausgabe erzeugt i.d.R. keine Pufferfehler (falls nicht umgeleitet)

Der Fehlerwert `EOF`

- Signalisiert Pufferfehler, aber keine Unterscheidung zwischen Dateiende und anderen Pufferfehlern
- Möglicher Rückgabewert verschiedener Ein- und Ausgabefunktionen
- Ist systemabhängig definiert als sog. Makro in `stdio.h` (Details zu Makros: späteres Kapitel)
- Wird wie eine symbolische Konstante benutzt

Die Funktion `getchar`

Die `getchar`-Funktion ist eine Bibliotheksfunktion aus `stdio.h`

`getchar`-Funktion

```
int getchar(void)
```

`int`	Typ des Rückgabewerts
`getchar`	Funktionsname
`void`	kein Eingabeparameter

- Mit `getchar` können einzelne Zeichen, die vom Benutzer über die Tastatur eingegeben wurden, eingelesen werden
- Rückgabewert: Der ASCII-Code des eingegebenen Zeichens oder `EOF` bei Auftreten eines Pufferfehlers

Benutzung von `getchar`

Beim Aufruf `getchar()` passiert Folgendes:

- Ist der Puffer leer, so **blockiert** das Programm (d.h. nachfolgende Anweisungen werden nicht ausgeführt), bis der Benutzer über die Tastatur (mindestens) ein Zeichen in der Kommandozeile eingibt und die Eingabe mit der Eingabetaste abschließt
- Im Puffer befinden sich dann die eingegebenen Zeichen in der Reihenfolge der Eingabe mit `'\n'` als letztem Zeichen
- Nach der Eingabe gibt `getchar()` das erste Zeichen im Puffer zurück.
- Weitere Zeichen **verbleiben im Puffer** (falls der Benutzer mehr als ein Zeichen eingegeben hat); diese können mit weiteren Aufrufen von Eingabefunktionen eingelesen werden

Beispiel 7.1

Gibt der Benutzer `"xyz\n"` ein, so gibt `getchar()` das Zeichen `'x'` zurück und `"yz\n"` verbleibt im Puffer.

Puffer leeren (ohne Pufferfehler)

Manchmal ist es nützlich, den Puffer zu **leeren** (d.h. alle Zeichen aus dem Puffer zu holen), **um weitere Eingaben nicht zu blockieren**.

```
1  void flush(void)
2  {
3      char c = getchar();
4      while (c != '\n') {
5          c = getchar();
6      }
7  }
```

- Zeile 3: Hole erstes Zeichen aus dem Puffer und speichere es in der Variable `c` - **Achtung**: Programm blockiert und wartet auf Eingabe, falls Puffer leer
- Zeile 4: Überprüfe, ob dieses Zeichen das letzte Zeichen `'\n'` ist (falls ja: Puffer ist leer)
- Zeile 5: Hole solange Zeichen aus dem Puffer bis zum letzten Zeichen `'\n'`
- *Anmerkung: es gibt **keine Bibliotheksfunktion**, die man dafür verwenden kann*
- *Anmerkung: dies ist eine vereinfachte Version **ohne Berücksichtigung von Pufferfehlern** (inwiefern?) - Verbesserung späteres Unterkapitel*

Puffer leeren (ohne Pufferfehler)

Manchmal ist es nützlich, den Puffer zu **leeren** (d.h. alle Zeichen aus dem Puffer zu holen), um weitere Eingaben nicht zu blockieren.

```
1  void flush(void)
2  {
3      char c = getchar();
4      while (c != '\n') {
5          c = getchar();
6      }
7  }
```

Kurzform:

```
1  void flush(void)
2  {
3      while (getchar() != '\n') { }
4  }
```

Die Funktion `scanf`

Die `scanf`-Funktion ist eine Bibliotheksfunktion aus `stdio.h`

`scanf`-Funktion

```
int scanf(const char * format, ...)
```

`int`	Typ des Rückgabewerts
`scanf`	Funktionsname
`format`	Eingabeparameter vom Typ `const char *` (konstante sog. **Formatzeichenkette**)
`...`	Es kann weitere Eingabeparameter geben

- Mit `scanf` können über Tastatur eingegebene Zeichenfolgen in Zahlen umgewandelt werden, die in Zahlvariablen gespeichert werden.
- Die Zeichenkette `format` enthält dazu Umwandlungsangaben.
- Als weitere Eingabeparameter werden die Adressen der Variablen übergeben.
- Eine Umwandlung muss nicht gelingen, denn dazu muss die Eingabe für die Umwandlungsangabe in der passenden Zahl-Schreibweise vorliegen.
- Rückgabewert: Anzahl der gelungenen Umwandlungen oder `EOF` bei Auftreten eines Pufferfehlers

Benutzung von `scanf` (ohne Pufferfehler)

Eine ganze Zahl einlesen

Beim Aufruf `scanf("%i", &n)` passiert folgendes (für eine `int`-Variable `n`):

- Ist der Puffer leer, so **blockiert** das Programm, bis der Benutzer über die Tastatur (mindestens) ein Zeichen in der Kommandozeile eingibt und die Eingabe mit der Eingabetaste abschließt
- Nach der Eingabe versucht `scanf("%i", &n)` ein **möglichst langes Anfangsstück** der Eingabe als ganze Zahl zu interpretieren, und zwar gemäß der Schreibweise von Ganzzahl-Konstanten; Dabei werden führende **Zwischenraumzeichen** (siehe `isspace` in `ctype.h`) überlesen
- Gibt es ein solches Anfangsstück, wird es in eine ganze Zahl umgewandelt, welche an der Adresse der Variable `n` gespeichert wird; Übrige (nicht umgewandelte) Zeichen **verbleiben im Puffer**; `scanf` hat den Rückgabewert `1`
- Gibt es **kein** solches Anfangsstück, **verbleiben alle Zeichen im Puffer**; `scanf` hat den Rückgabewert `0`

Beispiel 7.2

Gibt der Benutzer `"12.1cm\n"` ein, so speichert `scanf("%i", &n)` die Zahl `12` in `n` und gibt `1` zurück. `".1cm\n"` verbleibt im Puffer und **blockiert weitere Eingaben**.

Benutzung von `scanf` (ohne Pufferfehler)

Eine Dezimalzahl einlesen

Beim Aufruf `scanf("%lf", &x)` passiert folgendes (für eine `double`-Variable `x`):

- Ist der Puffer leer, so **blockiert** das Programm, bis der Benutzer über die Tastatur (mindestens) ein Zeichen in der Kommandozeile eingibt und die Eingabe mit der Eingabetaste abschließt
- Nach der Eingabe versucht `scanf("%lf", &x)` ein **möglichst langes Anfangsstück** der Eingabe als Dezimalzehl zu interpretieren, und zwar gemäß der Schreibweise von Dezimalzahl-Konstanten; Dabei werden führende **Zwischenraumzeichen** (siehe `isspace` in `ctype.h`) überlesen
- Gibt es ein solches Anfangsstück, wird es in eine Dezimalzahl umgewandelt, welche an der Adresse der Variable `x` gespeichert wird; Übrige (nicht umgewandelte) Zeichen **verbleiben im Puffer**; `scanf` hat den Rückgabewert `1`
- Gibt es **kein** solches Anfangsstück, **verbleiben alle Zeichen im Puffer**; `scanf` hat den Rückgabewert `0`

Beispiel 7.3

Gibt der Benutzer `"12.1cm\n"` ein, so speichert `scanf("%lf", &x)` die Zahl `12.1` in `x` und gibt `1` zurück. `"cm\n"` verbleibt im Puffer und **blockiert weitere Eingaben**.

Benutzung von `scanf` (ohne Pufferfehler)

Mehrere Zahlen einlesen

- Der Eingabeparameter `format` von `scanf` kann wie `printf` mehrere Umwandlungsangaben enthalten.
- Für jede Umwandlungsangabe muss die Adresse `&x` einer (vorher deklarierten) Variablen `x` übergeben werden, die einen zur Umwandlungsangabe passenden Typ hat
- Mehrere Umwandlungsangaben werden in `format` durch Trennzeichen getrennt; Diese Zeichen müssen genau wie vorgegeben vom Benutzer zwischen den Zahlen eingegeben werden.
- Erinnerung: `scanf` gibt Anzahl der gelungenen Wertumwandlungen zurück
- Umwandlungsangaben (Auswahl):
  `%lf`: Für Datentyp `double`
  `%i`: Für Datentyp `int`

Beispiel 7.4

Ein Datum einlesen mit `'.'` als Trennzeichen:
`scanf("%i.%i.%i", &day, &month, &year)`
Gültige Eingaben: `1.1.2020`, `-1.1000.2` (Rückgabe von `scanf`: `3`)
Ungültige Eingabe: `1-1-2020` (Rückgabe von `scanf`: `1`)

Benutzung des Adressoperators `&`

Frage

Wieso funktioniert `scanf("%i",x)` (für eine Zahlvariable `x`) nicht?

- **Call by Value**-Prinzip: Wert von `x` kann in `scanf` in diesem Fall nicht geändert werden
- **Call by Reference**-Prinzip: Übergibt man die **Adresse** `&x` von `x`, so kann der Wert von `x` in `scanf` geändert werden (wie bei Feldern, Details: späteres Kapitel)

Zusammenfassung

- Zeichen werden nach dem **FIFO-Prinzip** aus dem Puffer geholt (FIFO: **F**irst **I**n **F**irst **O**ut)
- **Zwischenraumzeichen** (bzw. **Whitespace-Zeichen**) zu Beginn werden überlesen
- Umwandlungsangaben sorgen für eine Umwandlung eingegebener Zeichen in Zahlwerte vom gewünschten Typ
- Trennzeichen in der Formatzeichenkette müssen an richtiger Stelle in der Eingabe vorkommen
- Es wird geprüft, ob ein Anfangsstück des Pufferinhalts zu einer Umwandlungsangabe passt: restliche Zeichen bleiben im Puffer stehen und blockieren weitere Eingaben

Was sind ungültige Eingaben?

- Programme erwarten Benutzereingaben in einer bestimmten Form und/oder Schreibweise, um diese korrekt interpretieren und verwenden zu können.
- Eine Benutzereingabe ist grundsätzlich **ungültig**, wenn sie nicht interpretierbar ist.
- In dieser Vorlesung betrachten wir nur solche Eingaben als interpretierbar, die **komplett** der erwarteten Form entsprechen, ohne dass restliche Zeichen im Eingabepuffer verbleiben

Beispiel 7.5 (Ungültige Eingaben)

Erwartet ein Programm als Eingabe eine ganze Zahl, so sind folgende Eingaben ungültig:

- `a` (nicht als ganze Zahl interpretierbar)
- `5a` (nicht komplett als ganze Zahl interpretierbar)
- `5.1` (nicht komplett als ganze Zahl interpretierbar)

Wie können Eingaben auf Gültigkeit überprüft werden?

In einfachen Fällen ist eine Eingabe gültig, ***falls sie einem bestimmten Datentyp entspricht*** *(z.B. dem Datentyp* `int`*)*:

- Eingabe mit `scanf` und zum Datentyp passender Umwandlungsangabe einlesen
- **Überprüfung 1**: *Gab es eine Umwandlung?*
  Überprüfe Rückgabewert von `scanf`: Muss `1` sein (für eine Umwandlungsangabe)
- **Überprüfung 2**: *Gibt es noch Zeichen im Puffer außer* `'\n'`*?*
  Überprüfe mit `getchar` das nächste Zeichen im Puffer: Dieses muss `'\n'` sein (da `'\n'` immer das letzte Zeichen im Puffer ist)

Beispiel 7.6 (Eine ganze Zahl einlesen)

```
1  int x, status;
2  status = scanf("%i", &x);
3  if (status == 0 || getchar() != '\n') {
4      /*Eingabe konnte gar nicht oder nicht komplett
5        umgewandelt werden*/
6  }
```

Wie können Eingaben auf Gültigkeit überprüft werden?

Manchmal ist eine Eingabe gültig, falls sie einem bestimmtem Datentyp entspricht ***und eine zusätzliche Eigenschaft erfüllt*** *(z.B. falls eine positive ganze Zahl als Eingabe erwartet wird)*:

- Eingabe mit `scanf` und zum Datentyp passender Umwandlungsangabe einlesen
- **Überprüfung 1**: *Gab es eine Umwandlung?*
- **Überprüfung 2**: *Gibt es noch Zeichen im Puffer außer* `'\n'`*?*
- **Überprüfung 3**: *Ist die zusätzliche Eigenschaft erfüllt?*
  Überprüfe die Zusatzeigenschaft mit einer passenden Bedingung.

Beispiel 7.7 (Eine positive ganze Zahl einlesen)

```
1  int x, status;
2  status = scanf("%i", &x);
3  if (status == 0 || getchar() != '\n' || (x <= 0) {
4    /*Eingabe konnte nicht (komplett) umgewandelt werden
5      oder ist nicht positiv*/
6  }
```

Wie können Eingaben auf Gültigkeit überprüft werden?

Es gibt Eingaben, die eine Schreibweise verlangen, ***die nicht direkt mit `scanf` und Bedingungen überprüft werden kann***:

- Eingabe **in einer Zeichenkette** speichern:
  Verschiedene Möglichkeiten (nächstes Unterkapitel)
- **Überprüfungen**: (Nächstes Unterkapitel)

Beispiel 7.8 (Komplexe Eingaben)

- Postleitzahlen (Länge: genau 5, besteht nur aus Ziffern)
- Namen (Länge: maximal 20, beginnt mit Großbuchstaben)

Wie soll man mit ungültigen Eingaben umgehen?

Die Überprüfung von Benutzereingaben auf Gültigkeit und den definierten Umgang mit ungültigen Eingaben nennt man **Fehlerbehandlung**

- Grundsätzlich darf ein Programm bei ungültigen Eingaben nicht aufgrund fehlender Überprüfungen undefiniert abbrechen oder fehlerhafte Ausgaben liefern, d.h. Programme benötigen eine Fehlerbehandlung ungültiger Eingaben
- Vielmehr soll es dem Benutzer für jede Art der Eingabe eine **sinnvolle Rückmeldung** geben.

Mögliche **Arten der Fehlerbehandlung** bei ungültigen Eingaben:

- Rückmeldung an den Benutzer über den Eingabefehler und Abbruch des Programms
- Rückmeldung an den Benutzer über den Eingabefehler und wiederholte Aufforderung zur Eingabe

Eigene Eingabefunktionen

Eine Eingabefunktion soll über ihren Rückgabewert

- die Eingabe zurückgeben
- und anzeigen, ob ein Fehler aufgetreten ist

Rückgabewerte von Eingabefunktionen

- Rückgabewerte, die Fehler anzeigen, **müssen unterscheidbar sein** von Rückgabewerten, die für eine Eingabe stehen.
- Bei Aufruf kann abhängig vom Rückgabewert eine geeignete Fehlerbehandlung (Abbruch, Wiederholung, Fehlermeldung) erfolgen

Eigene Eingabefunktionen

Beispiel 7.9 (Nicht-negative ganze Zahl einlesen)

```
1  int read_pos(void) {
2      int zahl;
3      if (scanf("%i", &zahl) != 1 || zahl < 0 ||
4          getchar() != '\n') {
5          flush();
6          return -1;
7      }
8      return zahl;
9  }
```

- Zeile 3: Lazy Evaluation beachten
- Leert bei ungültigen Eingaben den Eingabe-Puffer und gibt den **Fehlerwert** `-1` zurück
- Gibt bei gültigen Eingaben die eingelesene Zahl zurück
- Der Fehlerwert unterscheidet sich von allen Rückgaben für gültige Eingaben

Eigene Eingabefunktionen

Beispiel 7.10 (Nicht-negative ganze Zahl einlesen - `main`)

```
1  int main(void) {
2      int a;
3      do {
4          printf("Nicht-negative ganze Zahl eingeben:\n");
5          if ((a = read_pos()) == -1)
6              printf("Eingabe ungueltig\n");
7      } while (a == -1);
8      return 0;
9  }
```

- Fordert den Benutzer bei ungültigen Eingaben erneut zur Eingabe auf

Zeichenketten einlesen

Grundsätzliche Möglichkeiten Zeichenketten einzulesen

1. **Mit Feldern**:
   - Zuerst muss Speicherplatz durch Deklaration eines Feldes **fest reserviert werden** (Speicherplatz steht bei Compilierung des Quellcodes fest)
   - Dann kann in diesem Speicherbereich eine Benutzereingabe gespeichert werden; dabei muss nicht der gesamte reservierte Bereich ausgenutzt werden
   - Speicherbereich kann zur Programm-Laufzeit **nicht geändert werden**: Überprüfung notwendig, ob Eingabe zu lang
2. **Mit Zeigern**:
   - Speicherplatz kann zur Programm-Laufzeit **dynamisch verwaltet** (d.h. verkleinert und vergrößert) werden: Anpassung der Länge der Zeichenkette an die Eingabe möglich
   - wird erst später behandelt

Zeichenketten einlesen: Variante 1

```
char v[N]; /*N konstanter Ganzzahl-Ausdruck */
scanf("%s", v); /*v ist adresswertig */
```

Wirkung

- Speichert die komplette Eingabe des Benutzers in `v` und ersetzt dabei das abschließende `'\n'` der Eingabe durch `'\0'`
- **Unsicherer Aufruf**: Falls die Eingabe länger ist als der für `v` reservierte Speicherbereich, wird über diesen Bereich hinaus geschrieben

Fazit

- **Nicht benutzen**

Zeichenketten einlesen: Variante 1

Beispiel 7.11 (Pufferüberlauf bei Variante 1)

```
#include <stdio.h>

int main()
{
  char c = 'A';
  char test[2];
  printf("Maximal 1 Zeichen eingeben (sonst Speicherfehler):
      ");
  scanf("%s", test);
  printf("c: %c", c);
  return 0;
}
```

- Im Speicher werden `test` und `c` hintereinander abgelegt
- Überschreiben von `c` durch zu lange Eingabe (wird evtl. durch bestimmte Compiler verhindert)

Zeichenketten einlesen: Variante 2

```
char v[N]; /*N konstanter Ganzzahl-Ausdruck */
scanf("%Ms", v); /*M Ganzzahl-Konstante */
```

Wirkung

- Speichert die ersten `M` Zeichen der Eingabe des Benutzers in `v`
- **Sicherer Aufruf**: `M` kann so gewählt werden, dass nicht über den reservierten Speicherbereich hinaus geschrieben wird

Fazit:

- Für `M` kann **keine symbolische Konstante** eingesetzt werden.
- Will man später den Speicherbereich verkleinern oder vergrößern, muss `M` an vielen Stellen (in allen Aufrufen von `scanf`) aktualisiert werden
- Dadurch wird das Programm **schwer wartbar**.

Zeichenketten einlesen: Variante 3

Implementiere **eigene Einlesefunktion** (Zeile 2), die die Länge der Eingabe abhängig von einer **symbolischen Konstante** (Zeile 1) überprüft

```
1  #define MAX_STRING 10
2  int read_string(char in[]);
3  int main() {
4    char input[MAX_STRING];
5    if (!read_string(input)) {
6      printf("Eingabe zu lang");
7      return 1;
8    }
9    return 0;
10 }
```

- Zeile 4 **Zeichenkette anlegen**: Benutze symbolische Konstante für deren maximale Länge
- Zeile 5 **Aufruf Einlesefunktion**: liest Benutzereingabe in Zeichenkette `input` ein und gibt `0` zurück, falls diese zu lang ist

Zeichenketten einlesen: Variante 3

```
1  #define MAX_STRING 10
2  int read_string(char in[]);
3  int main() {
4    char input[MAX_STRING];
5    if (!read_string(input)) {
6      printf("Eingabe zu lang");
7      return 1;
8    }
9    return 0;
10 }
```

Wirkung

- Speichert die ersten `MAX_STRING` Zeichen der Eingabe des Benutzers in `input`
- **Sicherer Aufruf**: `MAX_STRING` wird zur Definition aller Zeichenketten und in der Einlesefunktion verwendet
- **Wartbarkeit**: Will man später den Speicherbereich verkleinern oder vergrößern, muss man nur `MAX_STRING` anpassen

Zeichenketten einlesen: Variante 3

```
1  int read_string(char in[])
2  {
3      int i = 0;
4      char c = getchar();
5      while (c != '\n' && i < MAX_STRING - 1) {
6          in[i++] = c;
7          c = getchar();
8      }
9      if (i == MAX_STRING - 1 && c != '\n') {
10         flush();
11         return 0;
12     }
13     in[i] = '\0';
14     return 1;
15 }
```

- Zeilen 5 - 8: Durch **wiederholten Aufruf von `getchar`** die Eingabe zeichenweise aus dem Puffer holen und in `in` ablegen, dabei maximal `MAX_STRING - 1` Zeichen
- Zeilen 9 - 12: `0` zurückgeben und Puffer leeren, falls Eingabe zu lang war
- Zeile 13: Zeichenkette in `in` mit `'\0'` abschließen!

Zeichenketten einlesen: Komplexe Eingabeformate

Beispiel 7.12 (Postleitzahl einlesen)

```
1  int read_plz(char plz[])
2  {
3      int i, c;
4      for (i = 0; i < MAX_PLZ - 1; ++i) {
5          c = getchar();
6          if (isdigit(c)) /*Eingabeformat ueberpruefen*/
7              plz[i] = c;
8          else {
9              if (c != '\n') /*Puffer leeren, wenn noch Reste
                   */
10                 flush();
11             return 0;
12         }
13     }
14     if (getchar() != '\n') { /*Eingabelaenge ueberpruefen*/
15         flush();
16         return 0;
17     }
18     plz[MAX_PLZ - 1] = '\0'; /*Zeichenkette abschliessen*/
19     return 1;
```

7. Benutzereingaben

Pufferfehler beobachten

- Bei der Benutzereingabe über Tastatur kommt es in der Regel nicht zu Puffer-Fehlern
- Nicht immer jedoch erfolgt die Eingabe über Tastatur: Es ist durch **Datenumleitung** möglich, **ohne Änderung des Quellcodes** Daten aus einer Textdatei einzulesen
- Bei Eingabe aus einer Textdatei **müssen Pufferfehler abgefangen werden**!

Behandlung von Pufferfehlern

- Bekanntlich werden Pufferfehler von Eingabefunktionen wie `scanf` oder `getchar` durch die Rückgabe des Werts `EOF` angezeigt.
- Tritt ein Pufferfehler auf, wird das Programm in `main` mit Rückgabe eines Fehlerwerts **ungleich 0** abgebrochen (da keine weitere Eingabe möglich ist)

Daten umleiten: Variante 1

Textdateien auf Kommandozeile ausgeben

Ist `text.txt` eine Textdatei, so lässt sich deren Inhalt wie folgt auf Kommandozeile ausgeben (anzeigen):

- Unix / Linux / Mac: `cat Text.txt`
- Windows: `type Text.txt`

Textdateien in Programme umleiten (**pipen**)

Ist `in.txt` eine Textdatei und `<Programm>` ein Maschinenprogramm, so lässt sich der Inhalt von `in.txt` bei Programmaufruf als Standardeingabe in `<Programm>` umleiten (anstelle der Tastatureingabe):

- Unix / Linux / Mac: `cat in.txt|<Programm>`
- Windows: `type in.txt|<Programm>`

Durch das `|`-Symbol wird die Ausgabe des Befehls `cat` bzw. `type` nicht auf den Monitor geschickt, sondern als Eingabe für den folgenden Befehl benutzt. Das Programm bekommt den Inhalt als simulierte Tastenanschläge. Man sagt, die Datei wird in das Programm **gepipet**.

Daten umleiten: Variante 2

Standardeingabe aus Textdateien

Ist `in.txt` eine Textdatei und `<Programm>` ein Maschinenprogramm, so lässt sich der Inhalt von `in.txt` bei Programmaufruf **systemunabhängig** als Standardeingabe in `<Programm>` umleiten (anstelle der Tastatureingabe):

```
<Programm> < in.txt
```

Standardausgabe in Textdateien

Ist `out.txt` eine Textdatei und `<Programm>` ein Maschinenprogramm, so lässt sich die Standardausgabe des Programms bei Programmaufruf **systemunabhängig** nach `out.txt` umleiten (anstelle der Bildschirmausgabe):

```
<Programm> > out.txt
```

Beide Umleitungsarten lassen sich auch kombinieren.

Pufferfehler abfangen

Ab jetzt sollen alle Eingabefunktionen Pufferfehler über ihren Rückgabewert anzeigen

Beispiel 7.13 (Verbesserte Version von `flush`)

```
1  int flush_buff(void) {
2      int c;
3      while ((c = getchar()) != '\n' && c != EOF) {}
4      return c != EOF;
5  }
```

- Zeile 3: Lazy Evaluation beachten
- Zeilen 3 + 4: Rückgabe von jedem `getchar`-Aufruf auf `EOF` überprüfen
- Gibt bei Pufferfehler `0` zurück, sonst `1`
- Fehlerbehandlung erfolgt beim Aufrufer

Pufferfehler abfangen

Beispiel 7.14 (Verbesserte Version von `read_pos`)

```
1  int read_pos_buff(void)
2  {
3      int zahl, status;
4      int c = '\0';
5      status = scanf("%i", &zahl);
6      if (status == EOF)
7          return BUFFER_ERROR;
8      if (status != 1 || zahl < 0 || (c = getchar()) != '\n')
           {
9          if (c == EOF || !flush_buff())
10             return BUFFER_ERROR;
11         return INVALID_INPUT;
12     }
13     return zahl;
14 }
```

- `BUFFER_ERROR`, `INVALID_INPUT`: eigene symbolische Konstanten
- Rückgabe von jedem `getchar`-, `scanf`- und `flush_buff`-Aufruf auf `EOF` überprüfen (z.B. Zeile 9: Puffer leeren **mit Überprüfung auf Pufferfehler**)