Chair of Embedded Systems
Department of Computer Science
University of Augsburg, Germany

Martin Frieb

Universität Augsburg University

Wintersemester 2020/2021

Informatik 1

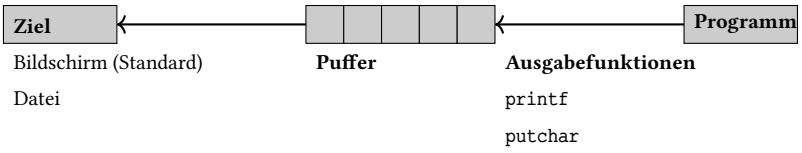
Kapitel 7 – Benutzereingaben

Inhaltsverzeichnis

Version 20201202-1720

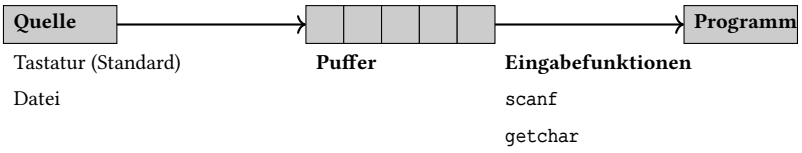
7.1 Standardein- und -ausgabe

Standardausgabe



- Bei der **Standardausgabe** ist das Ausgabeziel der Bildschirm.
- 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.
- Man kann die Standardausgabe auch in eine Textdatei umleiten (siehe Kapitel 7.6).

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 so umleiten, dass die Eingabe aus einer Textdatei erfolgt (siehe Kapitel 7.6).

Der Fehlerwert `EOF` (End of File)

Bei der Ein- oder Ausgabe von Zeichen kann es zu *Pufferfehlern* kommen. Das kann passieren, wenn beispielsweise beim Lesen einer Datei das Dateiende erreicht wurde, oder beim Schreiben einer Datei die Festplatte voll ist. Wenn ein solcher Pufferfehler vorkommt, wird ein spezieller Fehlerwert zurück gegeben. Dieser Wert ist systemabhängig definiert, als symbolische Konstante in `stdio.h`. Ausnahme bilden die Standard-Ein- und Ausgaben. Sie erzeugen in der Regel keine Pufferfehler.

Dieser spezielle Fehlerwert wird auch `EOF` genannt, kurz für *End of File*. Dieser Wert signalisiert einen aufgetretenen Pufferfehler, jedoch wird nicht zwischen verschiedenen Pufferfehlern unterschieden. D.h. es kommt beispielsweise nicht zur Unterscheidung zwischen Dateiende oder voller Festplatte.
Die Standard Ein- und Ausgabefunktionen aus der `stdio.h` können im Fehlerfall EOF zurück geben. Somit kann man sich den Fehlerwert `EOF` zunutze machen, um auf erfolgreiche Ein- und Ausgaben testen. Mehr dazu wird im weiteren Verlauf des Kapitels noch erläutert und anhand von Beispielen erklärt.

7.2 Eingabe einzelner Zeichen

7.2.1 Die Funktion `getchar`

Die `getchar`-Funktion ist eine Bibliotheksfunktion aus `stdio.h`. Sie ist definiert als
`int getchar(void)`

Die `getchar` Funktion ist „eine ganz normale Funktion“, wie wir sie bis jetzt kennengelernt haben. Das `int` am Anfang gibt an, dass die Funktion eine ganze Zahl zurück gibt. Auch verlangt sie keine Eingabeparameter, wie das `void` angibt.

Mithilfe von `getchar` können einzelne Zeichen, die vom Benutzer über die Tastatur eingegeben wurden, eingelesen werden. Somit wird eine Möglichkeit geschaffen, wie eine Person (also auch wir) mit dem geschriebenen Programm interagieren kann. Als Rückgabe liefert `getchar` den 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:

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

Beispiel:

Im Folgenden wird ein kurzes Beispiel gezeigt, wie man `getchar` nutzen kann. Führt der Benutzer bzw. die Benutzerin das Programm aus, „passiert erst einmal nichts“. Wir müssen nun etwas in die Kommandozeile tippen und mit Enter bestätigen. Anschließend wird der eingegebene Inhalt ausgegeben.

```
#include <stdio.h>

int main() {
    /* Hiermit wird das erste Zeichen aus dem
       Puffer genommen. Befindet sich kein
       Zeichen im Puffer, blockiert das Programm
        und wartet auf eine Eingabe. Befinden
```

```
            sich Zeichen im Puffer, wird das erste
            Zeichen aus dem Puffer genommen, in die
            Variable c gespeichert und aus dem Puffer
             gelöscht. */
        char c = getchar();

        /* Hier findet eine überprüfung statt, ob
            nur Enter ('\n' ist der "Code für Enter")
             gedrückt wurde oder Buchstaben und
            anschließend Enter gedrückt wurde. Falls
            (mehrere) Buchstaben eingegeben wurden,
            kommt man in die while-Schleife rein. */
        while (c != '\n') {
            /* Es wird zuerst das eingelesene
                Zeichen ausgegeben. */
            printf("%c", c);
            /* Anschließend nimmt man sich das
                nächste Zeichen aus dem Puffer heraus
                und speichert es in c hinein. Es
                kommt danach wieder zur Abfrage nach
                der Enter Taste. */
            c = getchar();
        }
        return 0;
    }
```

7.2.2 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**.

Hierfür definieren wir uns für die weitere Vorlesung eine Funktion namens `flush`. Sie hat die einzige Aufgabe, den Puffer zu leeren und die dort noch enthaltenen Eingaben zu verwerfen. Das geschieht mit einem wiederholten Aufruf von `getchar`, und zwar solange, bis sich kein Zeichen mehr im Puffer befindet.

Im Laufe der Vorlesung lernen wir jedoch noch weitere Abfragen und Fehlerarten kennen. Deswegen wird sie im Laufe der Vorlesung erweitert.

Die `flush` Funktion sieht nun folgendermaßen aus:

```
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** (d.h. es wird **nicht** auf `EOF` geprüft) – Verbesserung folgt im Kapitel 7.6*

Eine alternative Kurzform der Funktion wäre:

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

Beispiel:

Wie benutzt man nun diese `flush` Funktion? Das ist an sich ganz einfach. Es ist eine ganz normale Funktion, die wir aufrufen können. Hierfür können wir das obige Beispiel nochmal betrachten. Wir modifizieren es nun ein bisschen. Wir wollen jetzt, anstatt alle eingegeben Zeichen auszugeben, nur noch die maximal ersten zwei Zeichen eingeben.

Hierfür nutzen wir eine Zählvariable `i` und `if`-Abfragen.

```
#include <stdio.h>

void flush(void)
{
    while (getchar() != '\n') {}
}

int main() {
    int i = 0;
    char c = getchar();

    while (c != '\n') {
        /* Funktionsweise wie oben beschrieben.
           */
        printf("%c", c);
        c = getchar();

        /* Wir zählen nun das i hoch. Da wir
```

```
            maximal zwei Zeichen ausgeben wollen,
             überprüfen wir i. Sollte i > 1 sein,
             wurden bereits zwei Zeichen
            ausgegeben. Deswegen rufen wir flush
            auf, um den (eventuell) befüllten
            Puffer zu leeren. Zu guter Letzt wird
             mit dem break die Schleife
            abgebrochen. */
        i++;
        if (i > 1) {
            flush();
            break;
        }
    }
    return 0;
}
```

7.3 Eingabe von Zahlen

7.3.1 Exkurs: Call-by-Value und Call-by-Reference

Wir hatten *Call-by-Value* schon im Kapitel 2 und *Call-by-Reference* im Kapitel 6 kennengelernt. Bei Call-by-Value werden nur Kopien von Variablenwerten übergeben. Das heißt, man kann die Werte innerhalb der aufgerufenen Funktion beliebig verändern, ohne dass dies Auswirkungen auf die Originale hätte.

Beim Call-by-Reference Prinzip werden keine Kopien der Variablen übergeben, sondern deren Adressen. Somit lassen sich deren Werte direkt verändern. Das haben wir bereits bei den Feldern in Kapitel 6.7 angewendet: Felder können wir innerhalb von Funktionen verändern, an die sie übergeben wurden. Um eine einfache Variable (die kein Feld ist), via Call-by-Reference an eine Funktion zu übergeben, müssen wir ein `&` davor schreiben: `funktion(&variable)`.[1] Dies brauchen wir bei der Funktion `scanf`, die wir nun kennenlernen.

7.3.2 Die Funktion `scanf`

Uns ist schon die `printf`-Funktion aus der Bibliothek `stdio.h` bekannt, mit der man Variablen und deren Werte, ASCI-Zeichen usw. ausgeben kann. Dafür haben wir in der Ausgabe mithilfe von Platzhaltern, die Stellen spezifiziert wo wir Variablen oder ähnliches eines spezifischen Typs ausgeben möchten, bspw. `%i` für die Ausgabe einers Integers.

[1] Innerhalb der Funktion muss man mit der Variable dann auch etwas anders umgehen – das lernen wir im Kapitel 9.

Analog dazu gibt es die scanf-Funktion, eine weitere Bibliotheksfunktion aus stdio.h. Diese ist definiert als

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

Mit scanf können über Tastatur eingegebene Zeichenfolgen in Zahlen umgewandelt werden, die in Zahlvariablen gespeichert werden. Hierbei ist es sowohl möglich, nur eine Zahl einzulesen, aber auch mehrere Zahlen hintereinander einzulesen. Um die Eingabe zu speichern, muss dahinter die Adresse einer Variable angegeben werden, die zum Speichern verwendet wird (als &variable). Bei mehreren Eingaben muss für jede eingelesene Zahl eine eigene Variable angegeben werden. Die Funktion gibt einen Zahlenwert zurück. Dieser zurückgegebene Wert gibt entweder einen Fehlerwert an, oder die Anzahl der erfolgreich eingelesenen Zahlen.

Beispiel:

Im Folgenden wird ein kurzes Beispiel gezeigt. Wie jedoch genau die scanf Funktion und deren Eingabewerte funktionieren, wird im weiteren Verlauf des Skriptes erklärt. Zum besseren Verständnis (und als kleines Testbeispiel zum ausprobieren) wird jedoch schonmal ein kurzes Beispiel vorgegriffen.

```
#include <stdio.h>

int main(void)
{
    int x, y, status;
    printf("Geben Sie zwei ganze Zahlen ein:\n")
       ;

    /* Es wird auf die Eingabe von zwei Zahlen
       gewartet. Die erste Zahl wird in x, die
       zweite in y gespeichert. Im Erfolgsfall
       wird status auf 2 gesetzt. */
    status = scanf("%i %i", &x, &y);
    /* Sollte die Eingabe nicht erfolgreich
       gewesen sein (bspw. wenn nur eine Zahhl
       eingegeben wurde) oder noch andere Zahlen
        oder Buchstaben sich im Puffer befinden,
        dann wird eine Fehlermeldung ausgegeben
       und das Programm beendet. */
    if (status != 2 || getchar() != '\n') {
        printf("Eingabe ungueltig\n");
        return 1;
    }

    printf("Es wurden erfolgreich zwei Zahlen
       eingegeben: %i und %i", x, y);
    return 0;
```

```
}
```

Man beachte: Die Eingabe der beiden Zahlen muss genau so erfolgen wie das im `scanf` angegeben wurde, d.h. zwei ganze Zahlen mit einem Leerzeichen dazwischen.

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 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`) ignoriert.
- 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` (und zeigt damit an, dass eine Zahl erfolgreich eingelesen werden konnte).
- Gibt es **kein** solches Anfangsstück, **verbleiben alle Zeichen im Puffer**; `scanf` hat den Rückgabewert `0` (da keine Eingabe zu einer Zahl umgewandelt werden konnte).

Beispiel:

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**.

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 Dezimalzahl zu interpretieren, und zwar gemäß der Schreibweise von Dezimalzahl-Konstanten; Dabei werden führende **Zwischenraumzeichen** (siehe `isspace` in `ctype.h`) ignoriert.
- 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` (und zeigt damit an, dass eine Zahl erfolgreich eingelesen werden konnte).
- Gibt es **kein** solches Anfangsstück, **verbleiben alle Zeichen im Puffer**; `scanf` hat den Rückgabewert `0` (da keine Eingabe zu einer Zahl umgewandelt werden konnte).

> Beispiel:
>
> 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**.

Mehrere Zahlen einlesen:

- Der Eingabeparameter `format` von `scanf` kann wie `printf` mehrere Umwandlungsangaben enthalten.
- Für jede Umwandlungsangabe muss die Adresse `&x` einer (vorher deklarierten) Variable `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 die Anzahl der gelungenen Wertumwandlungen zurück.
- Umwandlungsangaben (Auswahl):
  `%lf`: Für Datentyp `double`
  `%i`: Für Datentyp `int`

> Beispiel:
>
> 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)

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.

7.4 Ungültige Eingaben und Fehlerbehandlung

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 (komplett) interpretierbar ist. Darunter fallen beispielsweise eingegebene Buchstaben, falls Zahlen erwartet werden.

Ein anderes Beispiel für eine ungültige Eingabe sind nicht komplett interpretierbare Werte. Beispielsweise fordert das Programm nur Zahlen als Eingabe, jedoch gibt eine Person die Zeichenkette „123abc“ein. Allerdings ist „abc“ keine Zahl, und somit ist dieser Teil der Eingabe nicht interpretierbar – also ist „123abc“ keine gültige Eingabe.

Man beachte: In dieser Vorlesung betrachten wir nur solche Eingaben als interpretierbar, die **komplett** der erwarteten Form entsprechen, ohne dass restliche Zeichen im Eingabepuffer verbleiben.

> **Beispiel: 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)
>
> Ein anderes Beispiel für einen Formatfehler lässt sich gut anhand vom obigen `scanf` zeigen: `scanf("\%i.\%i.\%i", \&day, \&month, \&year)`. Ein gültige Eingabe wäre bspw. 1.2.2019, ein ungültiges Eingabeformat wäre 1-2-2019.

7.4.1 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: 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 }
> ```

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: 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  }
```

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

- Eingabe **in einer Zeichenkette** speichern:
  Verschiedene Möglichkeiten (siehe Kapitel 7.5)
- **Überprüfungen**: (siehe Kapitel 7.5)

Beispiel: Komplexe Eingaben

- Postleitzahlen (Länge: genau 5, besteht nur aus Ziffern)
  Die Postleitzahl muss als Zeichenkette gespeichert werden, da sie auch führende Nullen besitzen kann. Dresden beispielsweise besitzt die Postleitzahl 01067. Ein `int` speichert jedoch keine führende Nullen, weshalb `int n = 01067` als `n = 1067` gespeichert wird – was keiner Postleitzahl entspricht.
- Namen (Länge: maximal 20, beginnt mit einem Großbuchstaben).

7.4.2 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.

7.4.3 Eigene Eingabefunktionen

Eine Eingabefunktion soll über ihren Rückgabewert

1. die Eingabe zurückgeben
2. 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: Nicht-negative ganze Zahl einlesen

```
int read_pos(void)
{
    int zahl;
    /* Hier muss die Lazy Evaluation Reihenfolge
        beachtet werden. Zuerst wird eine Zahl
       mit scanf eingelesen und überprüft, ob
       auch genau eine Zahl eingegeben wurde.
       Dann wird geprüft, ob sie kleiner 0 ist
       und abschließend ob ein Enter folgt oder
       nicht. */
    if (scanf("%i", &zahl) != 1 || zahl < 0 ||
       getchar() != '\n') {
```

```
            /* Bei ungültiger Eingabe muss der
               Puffer geleert werden und der
               Fehlerwert -1 wird zurück gegeben */
            flush();
            return -1;
        }
        /* Bei gültiger Eingabe wird die eingelesene
            Zahl zurück gegeben. */
        return zahl;
    }
```

Beispiel: 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

7.5 Eingabe komplexer Zeichenketten

Zeichenketten einlesen
Grundsätzliche Möglichkeiten, um 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 ist!
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 im Kapitel 9 behandelt.

Zeichenketten mit Feldern 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.

Bewertung:

- **Nicht benutzen**

Beispiel: Pufferüberlauf bei Variante 1

```
#include <stdio.h>
int main()
{
    /* Im Speicher werden test und c
       hintereinander abgelegt. Bildlich in
       Speicherzellen gesprochen folgt direkt
       auf test das char c. */
    char c = 'A';
    char test[2];
    printf("Maximal 1 Zeichen eingeben (sonst
       Speicherfehler)");

    /* Mit einer zu langen Eingabe wird nun c
       überschrieben (wird evtl. durch bestimmte
           Compiler verhindert) */
    scanf("%s", test);
    printf("c: %c", c);
    return 0;
}
```

Man beachte:
Wenn wir in `scanf` ein Feld übergeben, können wir das `&` weglassen!

Zeichenketten mit Feldern 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

Bewertung:

- 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 mit Feldern einlesen: Variante 3

Zu guter Letzt wird eine Variante vorgestellt, die im weiteren Verlauf der Vorlesung verwendet wird. Diese Variante ist sicher und leicht wartbar:

```
int read_string(char in[])
```

Bevor wir uns der Implementierung von `read_string` zuwenden, werfen wir einen kurzen Blick auf die Benutzung dieser Funktion. Sie ist als **eigene Einlesefunktion** (Zeile 2) Implementiert, 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.

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.

Die Funktion `read_string` selbst sieht wie folgt aus:

```
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 verarbeiten.
- Zeilen 9 - 12: `0` zurückgeben und Puffer leeren, falls Eingabe zu lang war.
- Zeile 13: Zeichenkette in `in` mit `'\0'` abschließen!

Bewertung:

- Einlesen in eigene Funktion gekapselt.
- Dadurch gut wartbar und mehrfach verwendbar.
- Sicher verwendbar, da das Beschreiben von nicht-reserviertem Speicher verhindert wird.

Zeichenketten einlesen: Komplexe Eingabeformate

Die Funktion `read_string` bietet außerdem eine gute Grundlage für komplexere Eingabeformate. Dafür müssen nur kleinere Veränderungen an der Funktion durchgeführt werden. Ein Beispiel wäre das Einlesen einer Postleitzahl. Dafür muss beachtet werden, dass die eingegebene Zahl genau 5 Stellen besitzt.

Beispiel: 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;
20  }
```

7.6 Pufferfehler

Pufferfehler beobachten

- Bei der Benutzereingabe über die 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, soll das Programm in `main` mit Rückgabe eines Fehlerwerts **ungleich** `0` abgebrochen werden, 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 der 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
Da wir nun die Ein- und Ausgabe mit Dateien kennen und entsprechende Pufferfehler beachten müssen, sollen ab jetzt alle Eingabefunktionen Pufferfehler über ihren Rückgabewert anzeigen:

Beispiel: 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

Beispiel: 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
- Entsprechende Rückmeldung an den Aufrufer
- Zeilen 6-7: Wenn `scanf` den Fehlerstatus `EOF` zurück liefert, Einlesen abbrechen und Pufferfehler an Aufrufer weitergeben.
- Zeile 9: Puffer leeren mit Überprüfung auf Pufferfehler
  Wenn `flush_buff` einen Pufferfehler zurückgibt, wird dieser an den Aufrufer weitergegeben.

Beispiel: `main`-Funktion, die `flush_buff` und `read_pos` verwendet

```
1   int main(void) {
2       int zahl;
3       do {
4           printf("Nicht-negative ganze Zahl
                eingeben:\n");
5           zahl = read_pos_buff();
6           if (zahl == INVALID_INPUT)
7               printf("Eingabe ungueltig\n");
8           else if (zahl == BUFFER_ERROR) {
9               printf("Pufferfehler\n");
10              return 1;
11          }
12      } while (zahl < 0);
13      printf("Es wurde %i eingegeben\n", zahl);
14      return 0;
15  }
```

- Wenn Sie eine Datei in das Programm pipen, in der eine Zahl, aber danach keine neue Zeile gespeichert ist, kommt es zu einem Pufferfehler.
- Sie können diesen auch mit Strg+Z (Windows) bzw. Strg+D (andere

Betriebssysteme) direkt eingeben, wenn Sie eine Zahl eingeben sollen.