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

Martin Frieb

Universität Augsburg University

Wintersemester 2020/2021

Informatik 1

Kapitel 6 – Einfache C-Programme

Inhaltsverzeichnis

6.1 Vorwissen aus dem Vorkurs

Aus dem Vorkurs sollten bestimmte Themen schon bekannt sein, bzw. während dem Semester bis jetzt aufgeholt worden sein. Deswegen werden sie hier auch nicht weiter vertieft.
Das beinhaltet hauptsächlich Grundlagen der C-Programmierung:

Programme erstellen, compilieren, ausführen:

- Texteditor installieren und zur Erstellung von Quellcode nutzen
- `gcc`-Compiler installieren (mit C-Standardbibliothek)
- Kommandozeile bedienen
- `gcc` aufrufen (Compilerschalter `-o`, `-ansi`, `-pedantic`, `-Wall`, `-Wextra`)
- Programmierkonventionen

Datentypen und Konstanten

- ASCII-Tabelle und Datentyp `char` für ASCII-Zeichen, ASCII-Konstanten und Escapesequenzen
- Datentyp `int` für ganze Zahlen, Dezimalschreibweise für Konstanten
- Datentyp `double` für Dezimalzahlen, Festkomma- und Gleitkommaschreibweise für Konstanten, Rundungen
- Zeichenkettenkonstanten

Variablen und Rechenausdrücke

- Lokale Variablen vom Typ `char`, `int` oder `double` deklarieren
- Wertzuweisungen an lokale Variablen, mit Typumwandlungen
- Arithmetische Rechenausdrücke (Operatoren `+`, `-`, `*`, `/`, `%`, `++`, `--`) mit Klammerung, Auswertungsreihenfolge und Typumwandlungen
- Expliziter Typcast

Logische Ausdrücke

- Vergleichsausdrücke (Operatoren `<`, `<=`, `>`, `>=`, `==`, `!=`)
- Logische Ausdrücke (Operatoren `!`, `&&`, `||`)
- Wahrheitswerte in C-Programmen

Kontrollstrukturen

- Fallunterscheidungen (`if`, `else`, `else if`)
- Wiederholungen (`while`, `for`), Endlosschleifen / Terminierung, Verschachtelte Schleifen

Funktionen

- Funktionen deklarieren (Funktionsprototyp), definieren und aufrufen
- Funktionen mit und ohne Eingabeparameter / Rückgabewert
- `main`-Funktion ohne Kommandozeilenparameter

Felder

- Felder deklarieren, initialisieren und ausgeben
- Felder an Funktionen übergeben
- Elemente / Maximum / Minimum in unsortierten Feldern suchen

Standardbibliothek
Auch sollten bereits einige Standardbibliotheken und deren Nutzen bekannt sein, wie z.B.:

- `stdio.h`
  - Formatierte Ausgaben mit `printf` (Umwandlungsangaben `%i`, `%c`, `%e`, `%f`, `%s`)
  - Einzelne Zeichen ausgeben mit `putchar`
- `math.h`
  - Verschiedene mathematische Funktionen (`sin`, `cos`, `log`, `floor`, `ceil`, `abs`, `sqrt`, `exp`, ...)
- `stdlib.h`
  - Zufallszahlen erzeugen mit `srand` und `rand`
- `ctype.h`
  - Funktionen für ASCII-Zeichen (`isdigit`, `islower`, `isupper`, `tolower`, `toupper`, ...)
- `limits.h`, `float.h`
  - Konstanten für Wertebereichsgrenzen der primitiven Datentypen (`INT_MIN`, `INT_MAX`, ...)

6.2 Ergänzungen: Primitive Datentypen

6.2.1 Datentyp `float`

Der Datentyp `float` ist, ähnlich wie der Datentyp `double`, für Dezimalzahlen gedacht.

> Definition: 6.1 Der Datentyp `float`
>
> Der Datentyp `float` ist ein Datentyp für **Dezimalzahlen** mit folgenden Eigenschaften:
>
> - Speicherbedarf: 4 Byte (also kleinerer Wertebereich und größere Rundungsfehler als bei `double`)
> - Typumwandlungen: `float` - Werte werden in Bewertungen und Berechnungen immer in `double` umgewandelt
> - Formatierte Ausgabe mit `printf`: wie `double`

Ansonsten wird `float` überall wie `double` eingesetzt.

6.2.2 Modifikatoren

Man kann in vielen Programmiersprachen auch bestimmen, ob der Datentyp ein Vorzeichen haben soll oder nicht. Man unterscheidet hierbei zwischen `signed` (für Werte mit Vorzeichen) und `unsigned` (für Werte ohne Vorzeichen).

Der Vorteil hierbei wird vor allem in der hardwarenahen Programmierung ersichtlich. Betrachtet man beispielsweise einen Datentypen, der aus 4 Bits besteht. Nutzt man das vorderste Bit als Vorzeichen-Bit (also `signed`), so kann man ganze Zahlen im Bereich zwischen -8 und 7 darstellen. Wird das vorderste Bit stattdessen für die Zahl verwendet (also `unsigned`) ist man in der Lage, ganze Zahlen im Bereich zwischen 0 und 15 darzustellen. Das ist vor allem dann praktisch, wenn man (sehr) wenig Speicherplatz besitzt und keine negativen Zahlen auftreten.
Man sollte bloß bei Rechenoperationen beachten, ob man mit `signed`, `unsigned` oder beiden Modifikationen rechnet, damit das vorderste Bit entweder als Vorzeichen-Bit oder für die Zahl korrekt verwendet wird. Berechnungs-Ergebnisse, die zu größeren Zahlen führen, werden sonst fälschlicherweise negativ interpretiert.

Definition: 6.2 Datentyp-Modifikatoren: Vorzeichen

Den Grund-Datentypen `char`, `short`, `int` und `long` können die folgenden Modifikatoren **in der Variablen-Deklaration vorangestellt** werden:

- `signed`:
  - Versieht den Datentyp mit einem Vorzeichen (+, −)
  - Codierung: 2K-Codierung
  - Wertebereich symmetrisch zur `0`: in etwa gleich viele Bitmuster repräsentieren jeweils negative und positive Zahlen (Abfrage über Konstanten in `limits.h`)
  - Die Grund-Datentypen sind i.d.R. vorzeichenbehaftet, d.h. `signed int` entspricht `int`, und so weiter
- `unsigned`:
  - Datentyp ohne Vorzeichen
  - Codierung: Binärcodierung
  - Wertebereich enthält nur nicht-negative Zahlen: alle Bitmuster können für nicht-negative Zahlen verwendet werden, d.h. es sind größere positive Zahlen darstellbar – siehe Wertebereich Binärcodierung (Abfrage über Konstanten in `limits.h`)

Einige ergänzende Umwandlungsangaben (unvollständige Liste):

`%u`	`unsigned int`
`%lu`	`unsigned long`

Einige ergänzende Schreibweisen für Konstanten (unvollständige Liste)

- Konstanten vom Typ `unsigned int`: mit `U` beenden
  Beispiel: `1U`
- Konstanten vom Typ `unsigned long`: mit `UL` beenden
  Beispiel: `1UL`

Beispiel:

```
int main(void)
```

```
{
    /* Initialisiere ein int MIT Vorzeichen */
    int signed_n = 3;

    /* Initialisiere ein int OHNE Vorzeichen */
    unsigned int unsigned_n = 3;

    /* Die negative Zahl wird in der 2K-
       Codierung gespeichert, aber als Binär-
       Codierung gelesen und ausgegeben. (Auf
       einem Windows 64 Bit System wird der Wert
        4294967293 ausgegeben) */
    printf("%u", -3);

    /* Die positiven Zahlen bleiben unverändert
       (Ausgabe des Wertes 3) */
    printf("%u", 3);

    return 0;
}
```

6.2.3 Datentyp `size_t`

Wenn wir bislang den `sizeof`-Operator verwendet haben, ist uns vielleicht aufgefallen, dass wir dessen rückgabe casten mussten, damit der Compiler keine Warnung bringt. Dies liegt daran, dass er den Datentyp `size_t` zurück gibt, der dafür gedacht ist, die Größe/Länge von etwas anzugeben.

Definition: 6.3 Der Datentyp `size_t`

Der Datentyp `size_t` ist ein Datentyp mit folgenden Eigenschaften:

- Ist Rückgabetyp des `sizeof`-Operators
- Ist nach Standard `unsigned` und ganzzahlig (und ansonsten compiler-abhängig implementiert)
- Entspricht beim `gcc` dem Typ `unsigned long`.

6.2.4 Konstante Werte

Es gibt mehrere Möglichkeiten, konstante Werte zu benutzen:

- Konstanten zu jedem Datentyp: sind festgelegt durch ihre Schreibweise; machen ein Programm als sog. *magic numbers* schwer wartbar (**Wiederholung**)

- Mit `#define` definierte symbolische Namen für Konstanten
- Konstante Variablen und Parameter mit `const`

Symbolische Konstanten mit `#define`:

- `#define <ZEICHENFOLGE> <Wert>`
- `<ZEICHENFOLGE>` wird per Suchen-und-Ersetzen im gesamten Quelltext durch `<Wert>` ersetzt.
- `<ZEICHENFOLGE>` wird **immer** in Großbuchstaben hingeschrieben, um klar zu machen, dass es sich nicht um eine Variable handelt, sondern um eine symbolische Konstante, die vor dem Kompilieren durch einen konkreten Wert ersetzt wird.
- Im kompilierten Programm taucht die Bezeichnung `<ZEICHENFOLGE>` nirgendwo auf. Es sieht so aus, als hätte man überall den `<Wert>` direkt hingeschrieben.

Beispiel:

```c
/* mit dem #define wird überall PI durch
   3.141592 ersetzt */
#define PI 3.141592

int main(void)
{
    /* Kreisberechnung mit PI */
    int radius = 2;
    float umfang = 2 * PI * radius;
    float flaeche = PI * radius * radius;

    /* Ausgabe von PI und Angaben zum Kreis */
    printf("Pi: %f\n", PI);
    printf("Kreis:\n Radius: %i\n", radius);
    printf("Umfang: %f\n", umfang);
    printf("Flaeche: %f\n", flaeche);

    /* PI kann nicht verändert werden. Folgende
       Zeile führt zu einem Fehler: */
    PI = 5;

    return 0;
}
```

Konstante Variablen und Parameter mit `const`:

- `const <Typ> <Variable>;`
- Stellt man in der Variablendekalartion den Modifikator `const` voran, so kann der Variablenwert **nur einmal** in der Deklaration gesetzt werden.
- Stellt man einem Eingabeparameter einer Funktion den Modifikator `const` voran, so kann dessen Wert in der Funktion nicht geändert werden.

Beispiel:

```c
int main(void)
{
    /* Initialisiere eine Konstante*/
    const int n = 3;

    /* Man kann die Konstante ganz normal nutzen
        (Wie bspw. Addieren und ausgeben) */
    printf("%i", n + 1);

    /* ABER man kann die Zahl nicht verändern.
        Für die folgende Zeile wird der Compiler
            einen Fehler ausgeben: */
    n = n + 3;

    return 0;
}
```

Mehr Details und Vergleich: spätere Kapitel

6.3 Ergänzungen: Fallunterscheidungen

6.3.1 `?:`-Operator

Der *Fragezeichen-Doppelpunkt Operator* (Oder auch: `?:`-Operator) ist eine Art, wie man kurze Fallunterscheidungen schreiben kann, ähnlich zu `if-else`.

Definition: 6.4 Bedingter Ausdruck `?:`-Operator

Ist `<B>` eine Bedingung und sind `<A1>` und `<A2>` Ausdrücke vom selben Typ, so ist `<B> ? <A1> : <A2>` ein **bedingter Ausdruck** mit folgendem Wert:

- ist `<B>` wahr, so hat er den Wert von `<A1>`
- ist `<B>` falsch, so hat er den Wert von `<A2>`

Die Ausdrücke `<A1>` und `<A2>` werden also alternativ in Abhängigkeit von der Bedingung `<B>` ausgewertet.

Den `?:`-Operator kann man auch sehr gut anhand eines Beispieles erklären:

Beispiel:

```c
int foo();
```

```c
int main(void)
{
    int n;

    /* Wenn man beispielsweise folgende
       Wertzuweisung realisieren will, kann man
       das als eine if-else Abfrage realisieren.
        */
    if (3 > 2) {
        n = 3;
    } else {
        n = 2;
    }

    /* Gibt den Wert 3 aus */
    printf("%i", n);


    /* Oder man schreibt eine ?: Anweisung.
       Links vom ? befindet sich die abgefragte
       Bedingung: ob 3 > 2 ist. Wird diese
       Bedingung als wahr evaluiert, dann wird
       der Wert 3 zurück gegeben und in n
       gespeichert. Wird die Bedingung als
       falsch evaluiert, wird in n der Wert 2
       gespeichert.*/
    n = (3 > 2) ? 3 : 2;

    /* Das Konzept des ?: lässt sich nicht
       perfekt mithilfe einer if-else Abfage
       darstellen. Besser wäre die Darstellung
       als eigene Funktion: */
    n = foo();

    /* Alle Auswertungen geben den Wert 3 aus.
       */
    printf("%i", n);

}

int foo()
{
    if (3 > 2) {
```

```
            return 3;
        } else {
            return 2;
        }
    }
```

Beispiel:

Der bedingte Ausdruck
`(x > y) ? x : y`
hat als Wert das **Maximum** von `x` und `y`

Es muss aber beachtet werden, dass der `?:`-Operator nur „klug“ eingesetzt werden sollte. Das heißt, man sollte ihn nicht willkürlich nutzen, bloß weil man es kann. Es wird empfohlen, den Operator nur für sehr kurze Codestücke zu verwenden. Der Grund dafür ist, dass der Code sonst sehr schnell sehr unleserlich wird.

6.3.2 `switch-case` - Anweisung

Wenn man abhängig vom Wert einer Variable verschiedene Fälle unterscheiden möchte, kann das zu einer längeren *if-else-else if*-Verkettung führen:
`if (x==0) ... else if (x==1) ... else if (x==2) ... else ...`
Stattdessen empfiehlt sich ein `switch-case`-Block. Nach dem `switch` wird die Variable angegeben, deren Fälle unterschieden werden sollen. Jedes `case` deckt einen Fall ab. Zusätzlich gibt es einen Anweisungsblock `default`, der ausgeführt wird, falls der Wert der Variable mit keinem der `case`s übereinstimmt.

Definition: 6.6 switch-case - Anweisung

```
switch (<N>) { /* <N> ist ein ganzzahliger
   Ausdruck */
case <K_1>: /* <K_1> ist eine ganzzahlige
   Konstante */
        <Anweisungen_1>
case <K_2>: /* <K_2> ist eine ganzzahlige
   Konstante */
        <Anweisungen_2>
...
default:
        <Alternative_Anweisungen>
}
<C> /* Nachfolgende Anweisung */
```

- Für jede Zeile `case <K_i>` wird überprüft, ob `<N> == <K_i>` zutrifft
- Falls ja: Es werden alle nachfolgenden Anweisungen ausgeführt bis zu ersten `break` - Anweisung; danach wird mit `<C>` fortgesetzt
- Falls keine der Bedingungen `<N> == <K_i>` zutrifft, werden die alternativen Anweisungen nach `default` ausgeführt

(eine `switch-case`- Anweisung ist also eine spezielle Fallunterscheidung, die man auch durch `if`-, `else if`- und `else`-Anweisungen ausdrücken kann)

Beispiel:

```
int main(void)
{
    /* Initialisiere einen zufälligen Wert*/
    int n;

    srand(time(NULL));
    n = rand();

    /* Man kann beispielsweise folgende if-else
       Abfragen schreiben */
    if (n == 0) {
        printf("n␣ist␣0");
    } else if (n == 1) {
        printf("n␣ist␣1");
    } else {
        printf("n␣ist␣irgendwas␣anderes");
    }
```

```
    /* Da es sehr langwierig sein kann, so viele
        else if zu tippen, kann man die oberen
       Zeilen gut durch ein switch-case ersetzen
        */
    switch(n) {
        /* Fall 1: n == 0 */
        case 0:
            printf("n␣ist␣0");
            break;
        /* Fall 2: n == 0 */
        case 1:
            printf("n␣ist␣1");
            break;
        /* Fall 3: n ist keins von beiden. Das
           ist der else Fall */
        default:
            printf("n␣ist␣irgendwas␣anderes");
            break;
    }
    return 0;
}
```

Die break Anweisung ist sehr wichtig. Ohne diese Anweisung wüsste das Programm nicht, wo es stoppen würde.

Beispiel:

```
int main(void)
{
    int n = 0;

    /* switch-case ohne break. Da n = 0 wird der
        erste case ausgeführt und die
       Zeichenkette ausgegeben. Da allerdings
       kein break vorhanden ist, wird auch case
       1 und der default case ausgeführt. */
    switch(n) {
        case 0:
            printf("n␣ist␣0");
        case 1:
            printf("n␣ist␣1");
        default:
            printf("n␣ist␣irgendwas␣anderes");
    }
```

```
        return 0;
}
```

6.4 Ergänzungen: Rechenausdrücke

6.4.1 Wertzuweisungs-Ausdrücke

Wir kennen bereits `v++` als Kurzschreibweise für `v=v+1`. Nun lernen wir ein paar weitere Kurzschreibweisen kennen:

Definition: 6.7 Wertzuweisungs-Ausdrücke in C

Sei `v` eine Variable und `A` ein Ausdruck. Wir definieren folgende **Wertzuweisungs-Ausdrücke**:

- `v = A`:
  `v` wird der Wert von `A` zugewiesen
  Der Ausdruck selbst hat auch den Wert von `A`
- `v += A`: Entspricht `v = v + A`
- `v -= A`: Entspricht `v = v - A`
- `v *= A`: Entspricht `v = v * A`
- `v /= A`: Entspricht `v = v / A`
- `v %= A`: Entspricht `v = v % A`

Mehrere nicht geklammerte Wertzuweisungen in einem Ausdruck werden **von rechts nach links ausgewertet**.

Die folgenden zwei Wertzuweisungen werden allerdings nicht empfohlen, da dadurch die Lesbarkeit des Codes sinkt. Man *kann* somit Werte zuweisen, muss aber nicht.

Beispiel: Mehrere Wertzuweisung in einer Anweisung

Der Ausdruck `v = w = A` entspricht `v = (w = A)`:

- Zuerst wird `w` der Wert von `A` zugewiesen
- Dann wird `v` der Wert von `w = A` zugewiesen. Dieser Wert entspricht dem Wert von `A`.

Außerdem kann man Wertzuweisungs-Ausdrücke mit anderen Ausdrücken kombinieren:

Beispiel: Wertzuweisung kombiniert mit anderem Ausdruck

Der Ausdruck `(x = y % 2) != 0` weist `x` als Wert den Rest bei ganzzahliger Division von `y` durch `2` zu und vergleicht diesen Rest dann mit dem Wert `0` auf Ungleichheit.

```c
if ((x = y % 2) != 0) {
    /* Falls bspw. y == 5, wird der Wert 1
       ausgegeben */
    printf("%i", x);
}
```

6.4.2 Inkrement und Dekrement

Ob wir v++ oder ++v hinschreiben, führt auf den ersten Blick zum selben Ergebnis: v wird um 1 erhöht. Jedoch ist die Rückgabe des Ausdrucks eine andere: v++ gibt den *alten* Wert von v zurück, d.h. den Wert, den v hatte, *bevor* es erhöht wurde. ++v erhöht *zuerst* den Wert von v und gibt anschließend den *neuen* Wert von v zurück.

Definition: 6.10 Inkrement und Dekrement in C

Sei v eine zahlwertige Variable. Weitere **Wertzuweisungs-Ausdrücke**:

- ++v:
  Der Wert von v wird um 1 erhöht
  Der Ausdruck selbst hat den **neuen** Wert von v
- v++:
  Der Wert von v wird um 1 erhöht
  Der Ausdruck selbst hat den **alten** Wert von v
- --v:
  Der Wert von v wird um 1 verringert
  Der Ausdruck selbst hat den **neuen** Wert von v
- v--:
  Der Wert von v wird um 1 verringert
  Der Ausdruck selbst hat den **alten** Wert von v

Beispiel:

```c
int main(void)
{
    int n = 0;

    /* Gibt den Wert 0 aus*/
    printf("%i", n);

    /* Gibt den Wert 1 aus */
    printf("%i", ++n);
```

```
        /* Gibt zuerst den Wert 1, dann den Wert 2
           aus */
        printf("%i", n++);
        printf("%i", n);

        return 0;
    }
```

6.5 Ergänzungen: Logische Ausdrücke (Bedingungen)

6.5.1 Auswertung

Definition: 6.12 Auswertung von Bedingungen

- Der Wahrheitswert `wahr` wird durch die ganze Zahl `1` repräsentiert[a]
- Der Wahrheitswert `falsch` wird durch die ganze Zahl `0` repräsentiert
- Ein **zahlwertiger Ausdruck** `A` hat den Wahrheitswert `A != 0` (wird also genau dann als wahr interpretiert, falls er einen Wert ungleich `0` hat.[b]
- Die Bedingung `!A` hat den Wert `1` genau dann wenn `A` den Wahrheitswert `0` hat[c]
- Die Bedingung `A && B` hat den Wert `1` genau dann wenn `A` und `B` den Wahrheitswert `1` haben
- Die Bedingung `A || B` hat den Wert `0` genau dann wenn `A` und `B` den Wahrheitswert `0` haben

[a] Der C89-Standard schreibt fest, dass Werte, die sich von `0` unterscheiden, für `wahr` stehen. Wir gehen zur Vereinfachung davon aus, dass im Fall von `wahr` der Wert `1` angenommen wird.

[b] Bspw. der Wert `5` ist wahr.

[c] `!A` ist also die Negation des Wahrheitswertes

6.5.2 Lazy Evaluation

Wenn ein Ausdruck aus mehreren durch `&&` oder `||` verknüpften Teilen nicht mehr wahr werden kann, verzichtet C auf die Auswertung der restlichen Teile.

Definition: 6.13 Verzögerte Auswertung - lazy evaluation

Der 'und'-Operator `&&` und der 'oder'-Operator `||` werden von links nach rechts ausgewertet, wobei dabei ggf. die Auswertung nachfolgender Ausdrücke unterbleibt (Sprechweise 'und dann' / 'oder dann'):

- `A && B`:
  Die Auswertung von `B` unterbleibt, wenn `A` falsch ist
  (dann ist offenbar auch `A && B` falsch)
- `A || B`
  Die Auswertung von `B` unterbleibt, wenn `A` wahr ist
  (dann ist offenbar auch `A || B` wahr)

Die Lazy Evaluation kann verwendet werden, um undefinierte Situationen zu verhindern. Beispielsweise kann die Abfrage `if (n % m != 0)` zu Problemen führen, wenn `m` den Wert `0` annimmt, da dann durch 0 geteilt wird. Daher kombiniert man diese Abfrage mit einer Überprüfung `(m > 0)` via `&&`. Sollte `m` nicht größer als `0` sein, wird der zweite Teil nicht ausgewertet und somit auch nicht durch 0 geteilt:

Beispiel: Schutz vor undefinierten Situationen

```
(m > 0) && (n % m != 0)
```

Lazy Evaluation ist besonders auch für die Abarbeitungsdauer eines Programms interessant. Durch eine kluge Reihenfolge der Abfragen lassen sich eventuelle kostspielige und komplexe Abfragen oder Berechnung vermeiden, sodass ein bisschen Zeit eingespart werden kann.

6.5.3 Auswertungsreihenfolge

Gleichrangige Operatoren werden **von links nach rechts** ausgewertet.

Beispiel:

Die Reihenfolge spielt für den 'und'- und 'oder'-Operator jeweils keine Rolle

- Es gilt (da `&&` **assoziativ** ist):
  `(A && B) && C == A && (B && C)`
- Es gilt (da `||` **assoziativ** ist):
  `(A || B) || C == A || (B || C)`

Beispiel:

Auswertung von `0 < x < 1` (entspricht `(0 < x) < 1`):
Zuerst wird der Vergleich `0 < x` ausgewertet (mit Wert `0` oder `1`), und dann das Ergebnis mittels `<` verglichen mit `1`.

Zum Beispiel für `x=0.5` ergibt sich aus `(0 < 0.5)` der Wahrheitswert `wahr = 1`, danach wird `(1 < 1)` überprüft mit dem Ergebnis `falsch = 0`.

6.5.4 Wahrheitstafeln

Eine **Wahrheitstafel** stellt die Auswertung einer Bedingung in Abhängigkeit von den Werten der Operanden der benutzten logischen Operation dar; in Wahrheitstafeln werden die **mathematischen Operationszeichen verwendet: $\neg$ (logisches nicht), $\wedge$ (logisches und), $\vee$ (logisches oder)**

Beispiel:

A	$\neg$A
0	1
1	0

A	B	A $\wedge$ B
1	1	1
1	0	0
0	1	0
0	0	0

A	B	A $\vee$ B
1	1	1
1	0	1
0	1	1
0	0	0

Hierbei werden in den Zeilen alle möglichen Kombinationen der Werte der Operanden A und B aufgelistet und in der letzten Spalte die zu diesen Kombinationen gehörenden Werte der betrachteten Bedingung

Mit Wahrheitstafeln lassen sich beliebige komplexe Bedingungen auswerten. Teilbedingungen werden dazu in eigenen Spalten aufgelistet und ausgewertet.

Beispiel:

A	B	C	A $\wedge$ B	(A $\wedge$ B) $\vee$ C
1	1	1	1	1
1	1	0	1	1
1	0	1	0	1
0	1	1	0	1
1	0	0	0	0
0	1	0	0	0
0	0	1	0	1
0	0	0	0	0

6.6 Ergänzungen: Wiederholungen

6.6.1 do-Schleife

Wir kennen bereits die while-Schleife, bei der zuerst die Schleifenbedingung geprüft und dann die Schleife durchlaufen wird. Bei der do-while-Schleife ist das genau umgekehrt:

Definition: 6.19 do-while - Anweisung

```
<A> /* Vorherige Anweisung */
do { /* Beginn do-while-Block */
        <Wiederholte_Anweisungen>
} while (<Schleifenbedingung>) /* Ende do-while-
   Block */
<B> /* Nachfolgende Anweisung */
```

- Nach <A> werden <Wiederholte_Anweisungen> ausgeführt und danach wird <Schleifenbedingung> überprüft
- <Wiederholte_Anweisungen> werden immer wieder ausgeführt, solange <Schleifenbedingung> bei der Überprüfung als **wahr** ausgewertet wird
- Ist <Schleifenbedingung> **nicht wahr**, so **terminiert** die Schleife und es wird mit <B> fortgesetzt

```
<A> /* Vorherige Anweisung */
do { /* Beginn do-while-Block */
        <Wiederholte_Anweisungen>
} while (<Schleifenbedingung>) /* Ende do-while-Block */
<B> /* Nachfolgende Anweisung */
```

entspricht

```
<A>
<Wiederholte_Anweisungen> /*Erstmalige Ausführung*/
while (<Schleifenbedingung>) {
  <Wiederholte_Anweisungen>
}
<B>
```

while vs. do-while

- while: Überprüfen der Schleifenbedingung **vor der ersten** Ausführung des while-Blocks. Es kann sein, dass der while-Block nie ausgeführt wird
- do-while: Überprüfen der Schleifenbedingung **nach der ersten** Ausführung des while-Blocks. Der while-Block wird mindestens einmal ausgeführt

Beispiel:

```
/*Beispiel zur Nutzung von do-while- und while-
   Schleifen*/

int main{void} {
        int i = 0;
        int sum = 0;

        do {
                sum += i;
                i++;
        } while (i < 100)

        /*im Gegensatz zu der anderen Mö
           glichkeit*/

        i = 0;
        sum = 0;
        while(i < 100) {
                sum += i;
                i++;
        }

        return 0;
}
```

6.6.2 `break` und `continue`

`break`-Anweisung
Die `break`-Anweisung ist uns schon beim `switch-case` unter gekommen, um den `switch-case`-Block zu verlassen. Ebenso kann sie in einer Schleife eingesetzt werden, um den Schleifenblock zu verlassen.

Definition: 6.20 break - Anweisung

```
<A>
while (<Schleifenbedingung>) {
        <Wiederholte_Anweisungen_Teil_1>
        break; /* Programm wird mit <B>
           fortgesetzt */
        <Wiederholte_Anweisungen_Teil_2>
}
<B>
```

- Nach der Ausführung einer `break`-Anweisung wird das Programm nach dem Schleifen-Block fortgesetzt
- Gilt entsprechend für `do-while`- und `for`-Schleifen
- Verschachtelte Schleifen: `break`-Anweisungen betreffen nur den Schleifen-Block, in dem diese stehen, aber nicht umfassende äußere Blöcke

`continue`-Anweisung

Im Gegensatz zu `break` verlässt `continue` nicht den kompletten Schleifenblock, sondern nur die aktuelle Iteration. Das heißt, die Schleifenbedingung wird erneut geprüft und wenn sie weiter erfüllt ist, wird die nächste Iteration ausgeführt.

Definition: 6.21 continue - Anweisung

```
<A>
while (<Schleifenbedingung>) {
        <Wiederholte_Anweisungen_Teil_1>
        continue; /* Programm wird mit <
           Schleifenbedingung> fortgesetzt */
        <Wiederholte_Anweisungen_Teil_2>
}
<B>
```

- Anweisungen nach einer `continue`-Anweisung werden übersprungen und direkt der nächste Schleifendurchlauf mit Auswertung der Schleifenbedingung ausgelöst
- Gilt entsprechend für `do-while`- und `for`-Schleifen
- Verschachtelte Scheifen: `continue`-Anweisungen betreffen nur den Schleifen-Block, in dem diese stehen, aber nicht umfassende äußere Blöcke

Beispiel:

```c
/*Beispiel zur Nutzung der break- und continue-
   Anweisung*/

int main(void) {

        /*Bestimmte die erste durch 17 teilbare
           Zahl größer als 500 aus*/
        int i = 0;
        int prim = 1;
        while(i < 1000) {
                i++;
                if (i < 500)
                        /*Führe gleich nächsten
                           Schleifendurchlauf
                           aus*/
                        continue;
                if (i % 17 == 0) {
                        prim = i;
                        /*Es sind keine weiteren
                            Durchläufe mehr
                           notwendig, verlasse
                           also die Schleife*/
                        break;
                }
        }

        printf("%i", prim);
        return 0;
}
```

6.7 Ergänzungen: Felder

6.7.1 Felder im Speicher

Deklaration eines (lokalen) Feldes `w` vom Typ `T` mit `N` Komponenten (Wiederholung):
`T w[N];`

Beispiel:

Ein `int` Feld, das Platz für 10 verschiedene ganze Zahlen besitzt.

```c
int x[10];
```

Der Feldname:

- `w` ist eine **adresswertige Konstante**
- Der Wert von `w` ist die Adresse der ersten Speicherzelle des Speicherbereichs, in dem das Feld gespeichert ist.

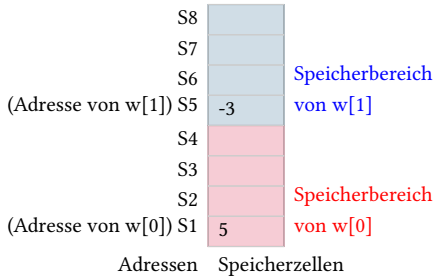
Die Feldkomponenten

- Der Speicherbedarf der Komponenten ist `N*sizeof(T)`. Dieser kann mit `sizeof(w)` abgefragt werden.
- Der reservierte Speicherbereich besteht aus aufeinanderfolgenden Speicherzellen im Stack, in denen die Komponenten nacheinander abgelegt werden.
- Der Speicherbereich von `w[i]` beginnt also bei der Adresse `w + i*sizeof(T)`.

Beispiel: `int`-Feld mit 2 Komponenten

```
int w[2];
w[0] = 5;
w[1] = -3;
```

- Speicherzelle = 1 Byte
- Der Wert von `w` ist `S1`
- `sizeof(w)`
  `= 2 * sizeof(int)`
  `= 2 * 4 = 8`
- `sizeof(w[0])`
  `= sizeof(w[1])`
  `= sizeof(int) = 4`
- Adresse von `w[1]`
  `= w + 1 * sizeof(int)`
  `= S1 + 4 = S5`



Für andere Datentypen haben die Komponenten einen anderen Speicherbedarf!

6.7.2 Erster Exkurs zu Adressen

Variablen oder Konstanten können auch Arbeitsspeicher-Adressen als Wert speichern. *Beispiel: Feldvariablen*

Adressen ausgeben:
Adressen können mit `printf` und der Umwandlungsangabe `%p` ausgegeben werden:

Beispiel: Ausgabe der Adresse eines Feldes

```
int v[10];
printf("Adresse von v: %p", v);
```

Auf Adressen zugreifen
Die Speicheradresse einer (beliebigen) Variable `x` vom Typ `T` erhält man mit dem Adressoperator `&`:

```
T x;
printf("%p", &x)
```

> Beispiel:
>
> ```
> int zahl;
> printf("Adresse von Zahl: %p", &zahl);
> ```

Adressen werden im späteren Verlauf der Vorlesung weiter vertieft.

6.7.3 Wichtige Eigenschaften

Felder und Wertzuweisungen - **Wiederholung**

- Eine Wertzuweisung direkt an ein Feld `w` (außer direkt in der Deklaration) ist nicht möglich (Compilerfehler, da `w` konstant)!
- Nur den Komponenten `w[i]` können Werte zugewiesen werden

Felder und Vergleiche - **Wiederholung**

- Ein Vergleich zweier Felder `v` und `w` der Form `v == w` vergleicht Adressen (Ergebnis ist für verschiedene Felder immer `0`)!
- Den Inhalt zweier Felder muss man über deren Komponenten mit `v[i] == w[i]` vergleichen.

Anpassung der Feldlänge zur Laufzeit?

- Die Anzahl der Komponenten ist (in C) durch eine Konstante im Quellcode festgelegt und kann zur Laufzeit nicht verändert werden! (**Wiederholung**)
- Wird auf nicht reservierten Speicherbereich zugegriffen, so erzeugt das nicht unbedingt Compiler-Warnungen/-Fehler oder Laufzeitfehler! (**Wiederholung**)
- **Folgerung 1**: Wähle diese Konstante groß genug für die Anwendung und lege sie einmalig mit `#define` fest (Vermeidung von *Magic Numbers* im Code)
- **Folgerung 2**: Benutze zur Laufzeit Felder, die maximal so lang sind, wie der durch die Konstante festgelegte Speicherbereich

Felder und Funktionen

- (**Wiederholung**) Funktion mit Feld als Eingabeparameter:
  `R array_function(T a[], int a_size, ...);`
  - Für `a` können Felder **unterschiedlicher Länge** übergeben werden
  - Für `a_size` wird **zusätzlich die Feldlänge** übergeben

- (**Wiederholung**) Aufruf der Funktion:
  `T w[N];`
  `array_function(w, N, ...);`
  - Für `a` wird der **Feldname** `w` übergeben (also eine Adresse)

Beispiel:

```c
/* Länge der Felder via #define festlegen */
#define LENGTH_ARRAY 5

/* Funktion, die Gleichheit zweier Felder prüft
   */
int is_equal(int v[], int w[], size_t length)
{
    int i = 0;
    while (i < length) {
        /* Eintrag unterschiedlich?
           => falsch zurückgeben */
        if (v[i] != w[i] return 0;
        i++;
    }
        /* wenn wir hier ankommen
          => alles stimmt überein */
        return 1;
}

int main(void)
{
    int v[LENGTH_ARRAY];
    int w[LENGTH_ARRAY];
    int i;

        /* Initialisierung */
    for (i = 0; i < LENGTH_ARRAY; i++) {
        v[i] = i;
        w[i] = i;
    }

        /* Vergleich und Rückgabe */
    if (is_equal(v, w, LENGTH_ARRAY))
        printf("Felder␣stimmen␣überein\n");
    else
        printf("Felder␣sind␣verschieden\n");

    return 0;
```

```
}
```

6.7.4 Call-by-Reference-Prinzip

Definition:

Wenn ein Feld an eine Funktion übergeben wird, erhält die Funktion **keine Kopie** des Feldes wie im Call-by-Value-Prinzip, **sondern das Original** übergeben. So kann der Inhalt des Feldes direkt gelesen und überschrieben werden.

- Beim Funktionsaufruf werden die originalen Exemplare (bspw. ein Feld w) als Eingabeparameter angegeben
- Zugriff auf Komponenten der Eingaben möglich durch w[i]
- Zusätzlich können die Komponenten `w[i]` gelesen **und überschrieben** werden (da die Adresse `w` übergeben wurde, aber nicht die Komponenten `w[i]`)
- weitere Details folgen in späteren Kapiteln

Im Gegensatz zu dem **Call-by-Value** ist also hier Vorsicht und Aufmerksamtkeit geboten, wenn mit Eingabeparametern dieser Art gearbeitet wird.

Beispiel:

```
/* Beispiel zum Call-by-Reference Prinzip */

void change_array(char w[]);

int main(void) {

        char v[6] = "Hallo";

        change_array(v);

        printf("%s", v);

        return 0;
}

void change_array(char[] w){
        int i = 0;
        while(w[i] != '\0') {
                w[i] = i + 65;
```

```
                    i++;
                }
            }
```

6.8 Zeichenketten

6.8.1 Was sind Zeichenketten?

Definition: 6.23 Zeichenkette

Eine **Zeichenkette** (ein **String**) ist ein Feld von (ASCII-)Zeichen, das im Speicher mit der sog. **binären Null** `'\0'` abgeschlossen wird.

Deklarationsmöglichkeiten:

- `char w[N];`
- `char w[N] = {<Zeichenkonstante>,...};`
- `char w[] = <konstante_Zeichenkette>;`

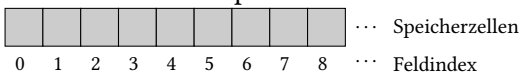
Beispiel:

```
char w[] = "Hallo";
```
entspricht
```
char w[6] = {'H','a','l','l','o','\0'};
```

Die binäre Null `'\0'` wird im weiteren Verlauf noch sehr wichtig. Ist beispielsweise nur die Zeichenkette gegeben, können wir alleine anhand der binären Null feststellen, wann die Zeichenkette aufhört.
Wenn bspw. für 10 Zeichen Speicherplatz reserviert wurde, aber das Wort nur 3 Buchstaben besitzt, bleiben somit immernoch Platz für 6 Zeichen (7 ohne die binäre Null). In diesen „leeren Speicherzellen“ befindet sich jedoch „Müll“. Unter „Müll“ versteht man den Inhalt der Speicherzelle vor der Reservierung. Das kann alles mögliche sein – mitunter auch andere Zeichenketten. Wäre die binäre Null nun nicht hier, wüsste man nicht wo die Zeichenkette aufhört und der „Müll“ beginnt.

6.8.2 Zeichenketten im Speicher

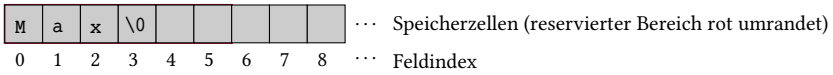
Zeichenketten im Speicher:



Beispiel: Zeichenkette "Hallo" im Speicher

Beispiel: Der reservierte Speicherbereich muss nicht komplett ausgenutzt werden

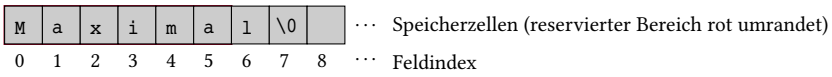
```
char w[6];
strcpy(w,"Max"); /*Kopierfunktion aus string.h */
```



Mit der Funktion `strcpy` wird der übergebene String `"Max"` an die Speicherstelle `w` kopiert.

ACHTUNG: C erlaubt, dass man über den reservierten Speicherbereich hinaus schreibt.

```
char w[6];
strcpy(w,"Maximal"); /*Kopierfunktion */
```



Hier sind nur 6 Speicherzellen reserviert, aber etwas Längeres wird dorthin kopiert. Dies ist unbedingt zu vermeiden: Da nachfolgende Variablen im Speicher überschrieben werden können, kann es zu Rechen- und Programmfehlern kommen!

6.8.3 Wichtige Eigenschaften

Zeichenkettenkonstanten:

Sei `"Konstante"` eine im Code verwendete Zeichenkettenkonstante:

- Die Konstante ist (wie ein Zeichenkettenname) **adresswertig**
- An dieser Adresse sind deren Buchstaben, abgeschlossen mit `'\0'`, abgelegt

Abschluss einer Zeichenkette mit `'\0'`

- Man kann die Zeichen einer Zeichenkette mit dem Feldindex `i` in einer Schleife durchlaufen: Abbruch beim `'\0'`-Zeichen
- Man kann höchstens `N-1` Zeichen in einer Zeichenkette `char w[N]` speichern, da das letzte Zeichen im Speicher immer `'\0'` sein muss

Da Zeichenketten Felder sind, haben sie sonst die gleichen Eigenschaften wie Felder.

Zeichenketten und Wertzuweisungen

- Eine Wertzuweisung an eine Zeichenkette `w` (außer direkt in der Deklaration) ist nicht möglich (Compilerfehler, da `w` konstant)!
- Nur den Buchstaben `w[i]` der Zeichenkette können Werte zugewiesen werden

Zeichenketten und Vergleiche

- Ein Vergleich zweier Zeichenketten `v` und `w` der Form `v == w` vergleicht Adressen (Ergebnis ist immmer 0)!
- Den Inhalt zweier Zeichenketten muss man über deren Buchstaben mit `v[i] == w[i]` vergleichen.
- Dafür gibt es Bibliotheksfunktionen (Details später)

Beispiel:

```
int main(void)
{
    int i = 0;
    char v[6] = "Hallo";
    char w[6] = "Holla";

    /* KEIN gültiger Vergleich. Es werden
       Speicheradressen verglichen und nicht die
        Zeichen selbst */
    if (v == w) {
        printf("Kein␣gültiger␣Vergleich!");
    }

    /* Das wiederum wäre ein gültiger Vergleich
       */
    while (i < 6) {
        if (v[i] != w[i]) {
            printf("v␣und␣w␣sind␣unterschiedlich
               ");
        }
                i++;
    }

        return 0;
}
```

Anpassung der Zeichenkettenlänge zur Laufzeit?

- Die Anzahl der Buchstaben ist (in C) durch eine Konstante im Quellcode festgelegt und kann zur Laufzeit nicht verändert werden!
- Wird auf nicht reservierten Speicherbereich zugegriffen, erzeugt das keinen Compiler-, sondern einen Laufzeitfehler!
- **Folgerung 1**: Wähle diese Konstante groß genug für die Anwendung und lege sie einmalig mit `#define` fest (Vermeidung von *Magic Numbers* im Code)
- **Folgerung 2**: Benutze zur Laufzeit Zeichenketten, die maximal so lang sind, wie der durch die Konstante festgelegte Speicherbereich

Zeichenketten und Funktionen

- Funktion mit Zeichenkette als Eingabeparameter:
  `R string_function(char w[], ...);`

  - Für `w` können Zeichenketten **unterschiedlicher Länge** übergeben werden
  - **Besonderheit**: Die Zeichenkettenlänge wird **nicht** zusätzlich übergeben (da das Ende der Zeichenkette durch `'\0'` markiert ist)
  - **Alternativ** kann der Eingabeparameter die Form `char * w` haben (siehe Bibliotheksfunktionen, wird später eingeführt)
- Aufruf der Funktion:
  `char s[N];`
  `string_function(s, ...);`
  - Für `w` wird der **Zeichenkettenname** `s` übergeben (also eine Adresse)
- **Call by Reference**-Prinzip:
  - Im Funktionsrumpf können die Buchstaben `s[i]` gelesen **und überschrieben** werden

6.8.4 Funktionen für Zeichenketten

Beispiel: Länge einer Zeichenkette berechnen

Der Aufruf `my_strlen("Hallo")` gibt 5 zurück

```
1  int my_strlen(char w[]) {
2      int i = 0;
3      while (w[i] != '\0')
4          ++i;
5      return i;
6  }
```

- Zeichenketten sind im Speicher immer mit `'\0'` abgeschlossen
- Die Länge der Zeichenkette entspricht dem Index `n` mit `w[n] == '\0'`
- Zeichenkette wird in `while`-Schleife so weit durchlaufen, bis Index von `'\0'` gefunden wurde. Dieser Index wird zurückgegeben.
- Entspricht der Bibliotheksfunktion `strlen` aus `string.h`

Das im Beispiel dargestellte Prinzip bzw. Verfahren zur Behandlung beliebiger Zeichenketten ist ein plausibler Ansatz zur Implementierung einer Vielzahl von verschiedenen Funktion auf Zeichenketten. Im Folgenden werden weitere Funktionen aus der `string.h` Bibliothek vorgestellt, jedoch nicht deren exakte Implementierung. Dies ist auch nicht nötig, da wir vorrangig das Ziel haben, Bibliotheksfunktionen zu verstehen und zu benutzen. Überlegen Sie daher selbst, wie eine mögliche Implementierung umgesetzt sein könnte.

Zeichenketten vergleichen:
Beim Vergleich von Zeichenketten gilt die lexikographische Ordnung, die wir schon in Kapitel 4 kennengelernt haben: Über die Sortierung entscheidet das erste unterschiedliche Zeichen nach einem gemeinsamen Anfangsstück (das leer sein kann). Die Zeichen sind dabei gemäß `ASCII`-Tabelle geordnet.

`int strcmp(const char * v, const char * w)`
vergleicht `v` und `w` bzgl. der lexikographischen Sortierung und hat folgenden Rückgabewert:

- `0`: falls der Inhalt beider Zeichenketten gleich ist
- negativ: falls der Inhalt von `v` kleiner als der von `w` ist
- positiv: sonst

Beispiel: Der Aufruf `strcmp("Dumm","Durst")` hat einen negativen Wert.

Beispiel:

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

int main(void)
{
    char v[5] = "test";
    char w[5] = "test";

    /* Gibt den Wert 0 aus */
    printf("%i", strcmp(v, w));

    /* Alternativ: (Gibt den Wert 0 aus) */
    printf("%i", strcmp("test", "test"));

    /* Gibt einen Wert ungleich 0 aus (im
       Windows 64Bit System: gibt -1 aus) */
    printf("%i", strcmp("anderes Wort", "test"))
       ;

    return 0;
}
```

Zeichenketten kopieren:

```
char * strcpy(char * v, const char * w)
```

- kopiert den Inhalt von `w` nach `v` inklusive der abschließenden binären Null
- gibt (die Adresse von) `v` zurück.
- `const` zeigt an, dass der Inhalt von `w` in `strcpy` nicht geändert werden kann

`strcpy` ist eine **unsichere** Funktion: Ist der reservierte Speicherbereich für `w` länger als für `v`, dann schreibt `strcpy` über den für `v` reservierten Speicherbereich hinaus

Beispiel:

Funktionierender Beispiel:

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

int main(void)
{
    char v[12] = "kurzer test";
    char w[6] = "apfel";

    strcpy(v, w);

    /* Gibt zweimal "apfel" aus */
    printf("%s %s", w, v);

    return 0;
}
```

Beispiel:

Fehlschlagendes Beispiel:

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

int main(void)
{
    char v[12] = "kurzer test";
    char w[6] = "apfel";

    strcpy(w, v); /* w und v vertauscht */

    /* Gibt "kurzer test" und " test" aus. Kann
       aber je nach System variieren. */
    printf("%s %s", w, v);
```

```
    return 0;
}
```

Zeichenketten kopieren: Sichere Variante:
`char * strncpy(char * v, constchar * w, int size)`

- `size` begrenzt die Anzahl der kopierten Zeichen und kann passend gewählt werden
- **Achtung**: Ist `size` kleiner als die Länge von `w`, so wird am Ende von `v` **keine** binäre Null gesetzt - das muss der Programmierer durch eine separate Anweisung durchführen!

Beispiel:

```
strncpy(v, w, sizeof(v) - 1);
v[sizeof(v) - 1] = '\0';
```

Beispiel:

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

int main(void)
{

    char v[12] = "kurzer test";
    char w[6] = "apfel";

    /* durch strlen bleibt \0 stehen */
    strncpy(w, v, strlen(w));

    /* Gibt "kurze" und "kurzer test" aus */
    printf("%s", w);
    printf("\n");
    printf("%s", v);

    return 0;
}
```

Zeichenketten aneinanderhängen:
`char * strcat(char * v, const char * w)`

- hängt den Inhalt von `w` an den Inhalt von `v` an inklusive der abschließenden binären Null
- gibt (die Adresse von) `v` zurück.

`strcat` ist eine **unsichere** Funktion: Ist der reservierte Speicherbereich für `v` nicht ausreichend, so schreibt `strcat` über diesen Speicherbereich hinaus

Beispiel:

Funktionierender Fall:

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

int main(void)
{
    char v[12] = "kuchen";
    char w[6] = "apfel";

    strcat(v, w);

    /* Gibt "kuchenapfel" aus. */
    printf("%s", v);

    return 0;
}
```

Beispiel:

Fehlschlagendes Beispiel:

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

int main(void)
{
    char v[12] = "kuchen";
    char w[6] = "apfel";

    strcat(w, v);

    /* Fehlerfall.
    Gibt "apfelkuchen" und "uchen" aus. */
    printf("%s", w);
    printf("\n");
    printf("%s", v);

    return 0;
}
```

Zeichenketten aneinanderhängen: Sichere Variante:
`char * strncat(char * v, const char * w, int size)`

- `size` begrenzt die Länge der angehängten Zeichenkette und kann passend gewählt werden

Beispiel:

```
strncat(v, w, sizeof(v)- strlen(v)- 1);
```

Beispiel:

Neu in diesem Beispiel: `w` besitzt eine Größe von 8 anstatt 6!

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

int main(void)
{
    char v[12] = "kuchen";
    char w[8] = "apfel";

    strncat(w, v, sizeof(w) - strlen(w) - 1);

    /* Gibt "apfelku" und "kuchen" aus */
    printf("%s", w);
    printf("\n");
    printf("%s", v);

    return 0;
}
```

Zahlen in Zeichenketten umwandeln
Die `stdio.h`-Bibliotheksfunktion
`int sprintf(char * v, const char * format, ...)`
erzeugt wie `printf` eine Zeichenkette, gibt diese aber nicht auf Kommandozeile aus, sondern schreibt diese nach `v`

`sprintf` ist eine **unsichere** Funktion
Ist der reservierte Speicherbereich für `v` nicht ausreichend, so schreibt `sprintf` über diesen Speicherbereich hinaus

Beispiel:

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

```c
int main(void)
{
    char w[6] = "apfel";

    sprintf(w, "1234");

    /* Gibt die Zeichenkette 1234 aus */
    printf("%s", w);

    return 0;
}
```

Zahlen in Zeichenketten umwandeln: Sichere Variante

`int snprintf(char * v, int size, const char * format, ...)`

`size` begrenzt die Anzahl der geschriebenen Zeichen und kann passend gewählt werden

Zeichenketten in Zahlen umwandeln

Die `stdlib.h`-**Bibliotheksfunktion**

`int atoi(char * v)`

wandelt das als ganze Zahl interpretierbare Anfangsstück von `v` in eine Zahl vom Typ `int` um; dabei werden Zwischenraumzeichen am Anfang ignoriert

- Wenn das Resultat zu groß werden würde, wird (je nach Vorzeichen) der größte bzw. kleinste darstellbare Wert geliefert

Beispiel:

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

int main(void)
{
    /* Gibt den Wert 0 aus */
    printf("%i", atoi("apfelkuchen"));

    /* Gibt den Wert 42 aus */
    n = atoi("42␣Birnen␣in␣2␣Taschen");

    /* Gibt den Wert 0 aus */
    n = atoi("Birnen␣in␣2␣Taschen");

    return 0;
}
```

6.9 Ergänzungen: `main`-Funktion

`main`-Funktion mit Kommandozeilenparametern

Möchte man ein C-Programm schreiben, dessen Ausführung von dabei übergebenen Kommandozeilenparametern abhängt, so benutzt man folgenden Prototyp:
`int main(int argc, char * argv[])`

- `argc`: Anzahl der übergebenen Kommandozeilenparameter plus 1. Da der Programmname immer als Kommandozeilenparameter mitgezählt wird, entsteht dadurch diese +1.
- `argv`: Feld der übergebenen Kommandozeilenparameter
- `argv[0]`: Programmname (Name der ausgeführten Datei)
- `argv[1]`: Erster Kommandozeilenparameter
- `argv[n]`: n-ter Kommandozeilenparameter
- Jeder Kommandozeilenparameter `argv[i]` ist eine Zeichenkette.
  `argv[i][j]` ist der j-te Buchstabe von `argv[i]`.

Beispiel:

```
int main(int argc, char * argv[]) {
        int anzahl;
        printf("\nAnzahl␣der␣Parameter:␣%i",
           argc - 1);
        printf("\nProgrammname:␣");
        anzahl = printf("%s",argv[0]);
        printf("\nDer␣Programmname␣hat␣%i␣
           Zeichen.", anzahl);
        return 0;
}
```

Ausgabe nach Übersetzung `gcc bsp02.c -o prg02` und Programmaufruf `prg02 param1 param2`:

```
Anzahl der Parameter: 2
Programmname: prg02
Der Programmname hat 30 Zeichen.
```

Beispiel:

Eine andere Darstellung, wie man eine Funktion in der Kommandozeile aufruft (im Beispiel: Windows Kommandozeile. Für Linux oder OS Betriebssysteme analog.)

```
PS C:\Users\Edmin\Documents> gcc .\test.c -Wall -Wextra -ansi -pedantic
PS C:\Users\Edmin\Documents> .\a.exe Übergabeparameter1 Apfelkuchen witziger Spruch

Anzahl der Parameter: 4
Programmname: C:\Users\Edmin\Documents\a.exe
Der Programmname hat 30 Zeichen.
PS C:\Users\Edmin\Documents>
```