Vorlesung Informatik 1 (Wintersemester 2020/2021)

Kapitel 6: Einfache C-Programme

Martin Frieb
Johannes Metzger

Universität Augsburg
Fakultät für Angewandte Informatik

30. November 2020

6. Einfache C-Programme

6.1 Vorwissen

Themen aus dem Vorkurs

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

Themen aus dem Vorkurs

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

Themen aus dem Vorkurs

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

Themen aus dem Vorkurs

Standardbibliothek `stdio.h`

- Formatierte Ausgaben mit `printf` (Umwandlungsangaben `%i`, `%c`, `%e`, `%f`, `%s`)
- Einzelne Zeichen ausgeben mit `putchar`

Standardbibliothek `math.h`

- Verschiedene mathematische Funktionen (`sin`, `cos`, `log`, `floor`, `ceil`, `abs`, `sqrt`, `exp`, ...)

Standardbibliothek `stdlib.h`

- Zufallszahlen erzeugen mit `srand` und `rand`

Standardbibliothek `ctype.h`

- Funktionen für ASCII-Zeichen (`isdigit`, `islower`, `isupper`, `tolower`, `toupper`, ...)

Standardbibliotheken `limits.h`, `float.h`

- Konstanten für Wertebereichsgrenzen der primitiven Datentypen (`INT_MIN`, `INT_MAX`, ...)

Datentyp `float`

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`

Modifikatoren

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`)

Modifikatoren

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`

Datentyp `size_t`

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 compilerabhängig implementiert)
- Entspricht beim `gcc` dem Typ `unsigned long`.

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 (**Wiederholung**)
- Konstante Variablen und Parameter mit `const`

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.

Mehr Details und Vergleich: spätere Kapitel

? :-Operator

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.

Beispiel 6.5

Der bedingte Ausdruck

```
(x > y) ? x : y
```

hat als Wert das **Maximum** von `x` und `y`

`switch-case` - Anweisung

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)

Wertzuweisungs-Ausdrücke

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`

Wertzuweisungs-Ausdrücke

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

Beispiel 6.8 (Mehrere Wertzuweisungen 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 6.9 (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.

Inkrement und Dekrement

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 6.11

```
int n = 0 /*n hat den Wert 0*/
int m = ++n /*n und m haben den Wert 1*/
int k = n-- /*n hat den Wert 0, k hat den Wert 1*/
```

Auswertung

Definition 6.12 (Auswertung von Bedingungen)

- Der Wahrheitswert `wahr` wird durch die ganze Zahl `1` repräsentiert
- 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)
- Die Bedingung `!A` hat den Wert `1` genau dann wenn `A` den Wahrheitswert `0` hat
- 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

Lazy Evaluation

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)

Beispiel 6.14 (Schutz vor undefinierten Situationen)

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

Auswertungsreihenfolge

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

Beispiel 6.15

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 6.16

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`.

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 6.17

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

Wahrheitstafeln

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

Beispiel 6.18

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

do-Schleife

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

do-Schleife

```
<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

`break`-Anweisung

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

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

6. Einfache C-Programme

Felder im Speicher

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

Der Feldname

- `w` ist eine **adresswertige Konstante**
- der Wert von `w` ist die Adresse der ersten Speicherzelle des Speicherbereichs der Feldkomponenten

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)`

Felder im Speicher

Beispiel 6.22 (`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!

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("%p", v)
```

Auf Adressen zugreifen

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

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

Wichtige Eigenschaften

Felder und Wertzuweisungen - **Wiederholung**

- Eine Wertzuweisung 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.

Wichtige Eigenschaften

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

Wichtige Eigenschaften

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)
- **Call by Reference**-Prinzip:
  - Im Funktionsrumpf können die Komponenten `w[i]` gelesen **und überschrieben** werden (da die Adresse `w` übergeben wurde, aber nicht die Komponenten `w[i]`)
  - Details in späteren Kapiteln

6. Einfache C-Programme

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:

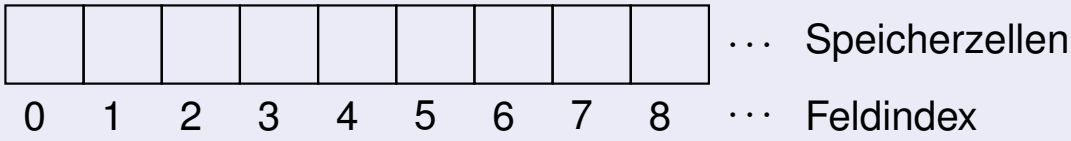
- `char w[N];`
- `char w[N] = {<Zeichenkonstante>,...};`
- `char w[] = <konstante_Zeichenkette>;`

Beispiel 6.24

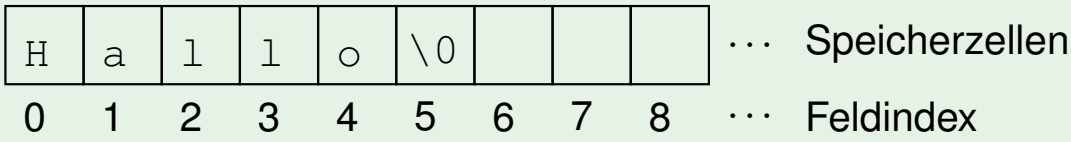
```
char w[] = "Hallo";
```
entspricht
```
char w[6] = {'H','a','l','l','o','\0'};
```

Zeichenketten im Speicher

Zeichenketten im Speicher



Beispiel 6.25 (Zeichenkette "Hallo" im Speicher)

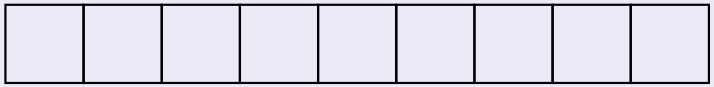


Beispiel 6.26 (Der reservierte Speicherbereich muss nicht komplett ausgenutzt werden)

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

Zeichenketten im Speicher

Zeichenketten im Speicher

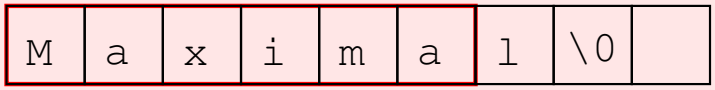


··· Speicherzellen

0 1 2 3 4 5 6 7 8 ··· Feldindex

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

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



··· Speicherzellen (reservierter Bereich rot umrandet)

0 1 2 3 4 5 6 7 8 ··· Feldindex

Unbedingt vermeiden: Da nachfolgende Variablen im Speicher überschrieben werden, kommt es zu Rechen- und Programmfehlern!

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

Wichtige Eigenschaften

Wie für Felder gilt:

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)

Wichtige Eigenschaften

Ebenso wie für Felder gilt:

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

Wichtige Eigenschaften

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

Funktionen für Zeichenketten

Beispiel 6.27 (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`

Funktionen für Zeichenketten

Zeichenketten vergleichen

Die `string.h`-Bibliotheksfunktion
`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

Lexikographische Sortierung (Details Kapitel 4)

Über die Sortierung entscheidet das erste unterschiedliche Zeichen nach einem gemeinsamen Anfangsstück (das leer sein kann). Zeichen sind dabei gemäß `ASCII`-Tabelle geordnet.
Beispiel: Der Aufruf `strcmp("Dumm","Durst")` hat negativen Wert.

Funktionen für Zeichenketten

Zeichenketten kopieren

Die `string.h`-Bibliotheksfunktion

```
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

Funktionen für Zeichenketten

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 6.28

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

Funktionen für Zeichenketten

Zeichenketten aneinanderhängen

Die `string.h`-Bibliotheksfunktion

```
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

Funktionen für Zeichenketten

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 6.29

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

Funktionen für Zeichenketten

Beispiel 6.30 (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 6.31 (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

Funktionen für Zeichenketten

Beispiel 6.32 (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
- Der Aufruf `atoi("16a")` liefert den Wert `16`

`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.
- `argv`: Array 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]`

`main`-Funktion mit Kommandozeilenparametern

Beispiel 6.33

```
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
param1
param2
```