Einführung in die Programmierung

Vorlesungsteil 9
Zeiger, Zeichenketten und Dateiarbeit

PD Dr. Thomas Hinze

Brandenburgische Technische Universität Cottbus – Senftenberg
Institut für Informatik, Informations- und Medientechnik

Wintersemester 2015/2016

Brandenburgische
Technische Universität
Cottbus - Senftenberg

Zeichenketten sind im Alltag präsent

Zeichenketten sind das in Datenbanken am häufigsten genutzte Datenformat.

- Auch scheinbar numerische Daten wie *Telefonnummern* `(0355)69-0` oder *Hausnummern* `42a`
- Zeichenketten widerspiegeln die *menschliche Schriftsprache*, in der traditionell Daten archiviert wurden und werden

Zeichenketten sind vielseitig

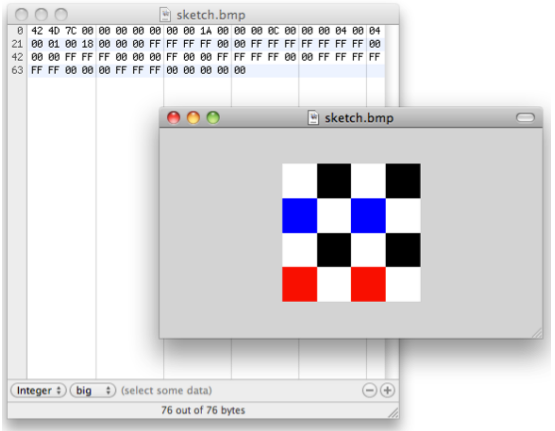
$$\sqrt{x^2 + y^2} \cdot \binom{a}{b}$$

```
$\sqrt{x^{2} + y^{2}} \cdot \left(a \atop b\right)$
```

Zeichenketten bieten eine universelle Datenstruktur.

- Beliebige Datenstrukturen wie *Zahlen*, *Bäume*, *Graphen* oder *Formeln* lassen sich verlustfrei und eindeutig durch wohldefinierte Abfolgen von Symbolen (Zeichen) darstellen
- Theorie *Formaler Sprachen* liefert algorithmische Werkzeuge zur Erzeugung und Analyse von Zeichenketten (z.B. Compiler einer Programmiersprache analysiert Quelltext und erzeugt Maschinencode)

Zeichenketten sequentialisieren Daten



Zeichenketten sind Grundlage für Dateien.

- *Datenströme*, z.B. aus Messungen, liefern eine Sequenz (Abfolge) aus Symbolen und mithin eine Zeichenkette
- Auch *Bilder* oder *Audiodaten* manifestieren sich in einer Sequenz aus Zeichen, die sich nach einer Bildungsvorschrift aus der Anordnung und Färbung von Pixeln oder aus Signalpegeln ergibt. Die resultierende Zeichensequenz ist dann als *Datei* speicherbar.

Dateien speichern Daten persistent

Eine Datei kann eine Zeichenkette langfristig computerlesbar archivieren.

- *Haltbarkeit* von Speichermedien für Dateien variiert je nach Nutzung und ist stets begrenzt
- **USB-Stick** . . . . . bis zu ca. **5** Jahren
- **Festplatte** . . . . . bis zu ca. **10** Jahren
- **CD / DVD** . . . . . mindestens **30** Jahre (Prognose)
- **Holografische Speicher** . . . . . $>$ **100** Jahre (Prognose)

Vorlesung Einführung in die Programmierung mit C

Was ist ein Zeiger in C? zeigerdemo1.c

Ein **Zeiger** (engl. *pointer*) ist in C eine *Variable*, die die *Adresse* (den Speicherort) eines getypten Datenobjekts enthält oder mit dem Wert `NULL` belegt ist.

Was ist ein Zeiger in C? zeigerdemo1.c

Ein **Zeiger** (engl. *pointer*) ist in C eine *Variable*, die die *Adresse* (den Speicherort) eines getypten Datenobjekts enthält oder mit dem Wert **`NULL`** belegt ist.

```c
#include <stdio.h>

int main(void)
{
  char w = 'A'; //getyptes Datenobjekt: char-Variable w

  char *p = NULL; //p ist ein Zeiger auf char-Werte

  p = &w; //p wird mit der Anfangsadresse von w belegt
          //p zeigt (verweist) jetzt auf den Inhalt von w

  printf("\n Zeiger p: Adresse %p, Inhalt: %c\n", p, *p);
  return 0;
}
```

```
Zeiger p: Adresse 0xbfa2e6db, Inhalt: A
```

Zeigereigenschaften

- Anlegen eines Zeigers in C:
  `<TypDatenobjekt> *<Zeigername>;`

Zeigereigenschaften

- Anlegen eines Zeigers in C:
  `<TypDatenobjekt> *<Zeigername>;`
- Ein Zeiger enthält stets die *Anfangsadresse* eines Datenobjekts, wenn dieses mehrere aufeinanderfolgende Adressen im Speicher belegt

Zeigereigenschaften

- Anlegen eines Zeigers in C:
  `<TypDatenobjekt> *<Zeigername>;`
- Ein Zeiger enthält stets die *Anfangsadresse* eines Datenobjekts, wenn dieses mehrere aufeinanderfolgende Adressen im Speicher belegt
- Die Größe des Adressblocks, die das Datenobjekt umfasst, ist durch seinen *Typ* festgelegt
  (z.B. `sizeof(char)` ist 1 Byte)

Zeigereigenschaften

- Anlegen eines Zeigers in C:
  `<TypDatenobjekt> *<Zeigername>;`
- Ein Zeiger enthält stets die *Anfangsadresse* eines Datenobjekts, wenn dieses mehrere aufeinanderfolgende Adressen im Speicher belegt
- Die Größe des Adressblocks, die das Datenobjekt umfasst, ist durch seinen *Typ* festgelegt
  (z.B. `sizeof(char)` ist 1 Byte)
- `NULL` ist eine in `stdio.h` definierte *Konstante*, die sich von einer regulären Adresse unterscheidet

Zeigereigenschaften

- Anlegen eines Zeigers in C:
  `<TypDatenobjekt> *<Zeigername>;`
- Ein Zeiger enthält stets die *Anfangsadresse* eines Datenobjekts, wenn dieses mehrere aufeinanderfolgende Adressen im Speicher belegt
- Die Größe des Adressblocks, die das Datenobjekt umfasst, ist durch seinen *Typ* festgelegt
  (z.B. `sizeof(char)` ist 1 Byte)
- `NULL` ist eine in `stdio.h` definierte *Konstante*, die sich von einer regulären Adresse unterscheidet
- Ein mit `NULL` belegter Zeiger heißt *Nullzeiger* (engl. *null pointer*) und wird zumeist zum Handling dynamischer Datenstrukturen genutzt, zum Beispiel, um das Ende einer Liste aus Datensätzen im Speicher zu markieren.

Arbeiten mit Zeigern

Wir benötigen dazu zwei wichtige Operatoren:

Referenzieren

Adressoperator `&`:
Sei `<Typ> a;` deklariert, dann liefert `&a` die Anfangsadresse von `a` im Speicher.
(Referenz: Verweis auf einen Speicherbereich)

Arbeiten mit Zeigern

Wir benötigen dazu zwei wichtige Operatoren:

Referenzieren

Adressoperator `&`:
Sei `<Typ> a;` deklariert, dann liefert `&a` die Anfangsadresse von `a` im Speicher.
(Referenz: Verweis auf einen Speicherbereich)

Dereferenzieren

Inhaltsoperator `*`:
Sei `<Typ> *p;` deklariert, wobei `p` eine Adresse ist. Dann liefert `*p` den Variablenwert der Bitkette ab Adresse `p`. Über den Typ `<Typ>` ist festgelegt, wieviele aufeinanderfolgende Bytes im Speicher ausgelesen werden und wie die Dekodierung der Bitkette in den Variablenwert erfolgt.

Zeiger geben Zugriff auf typverträgliche Variablen

zeigerdemo2.c

```c
#include <stdio.h>

int main(void)
{
  char x = 'A';
  char y = 'B';

  char *p = NULL;

  p = &x;
  printf("\n Zeiger p: Adresse %p, Inhalt: %c\n", p, *p);

  *p = 'E';
  printf("\n Zeiger p: Adresse %p, Inhalt: %c\n", p, *p);

  p = &y;
  printf("\n Zeiger p: Adresse %p, Inhalt: %c\n", p, *p);

  *p = 'G';
  printf("\n Zeiger p: Adresse %p, Inhalt: %c\n", p, *p);
  return 0;
}
```

```
Zeiger p: Adresse 0xbf9e57e9, Inhalt: A

Zeiger p: Adresse 0xbf9e57e9, Inhalt: E

Zeiger p: Adresse 0xbf9e57ea, Inhalt: B

Zeiger p: Adresse 0xbf9e57ea, Inhalt: G
```

Zeiger geben Zugriff auf typverträgliche Variablen

zeigerdemo3.c – Das gilt auch für Felder

```
Zeiger p: Adresse 0xbf88fd36, Inhalt: H

Zeiger p: Adresse 0xbf88fd37, Inhalt: a

Zeiger p: Adresse 0xbf88fd38, Inhalt: l

Zeiger p: Adresse 0xbf88fd38, Inhalt: m

Zeichenfeld: Hamlo
```

```c
#include <stdio.h>

int main(void)
{
  char z[6] = {'H', 'a', 'l', 'l', 'o', '\0'};

  char *p = NULL;

  p = z; //Anfangsadresse des Feldes z
  printf("\n Zeiger p: Adresse %p, Inhalt: %c\n", p, *p);

  p++;
  printf("\n Zeiger p: Adresse %p, Inhalt: %c\n", p, *p);

  p++;
  printf("\n Zeiger p: Adresse %p, Inhalt: %c\n", p, *p);

  *p = 'm'; //Alter Inhalt 'l' wird ueberschrieben
  printf("\n Zeiger p: Adresse %p, Inhalt: %c\n", p, *p);

  printf("\n Zeichenfeld: %s\n", z);
  return 0;
}
```

Operationen mit Zeigern

Sei ein Datenobjekt, zum Beispiel `int i = 5;` angelegt.

- **Deklarieren eines Zeigers** ..... `int *p;`

Operationen mit Zeigern

Sei ein Datenobjekt, zum Beispiel `int i = 5;` angelegt.

- **Deklarieren eines Zeigers** ..... `int *p;`
- **Zuweisen der Adresse eines Datenobjektes** ..... `p = &i;`

Operationen mit Zeigern

Sei ein Datenobjekt, zum Beispiel `int i = 5;` angelegt.

- **Deklarieren eines Zeigers** ..... `int *p;`
- **Zuweisen der Adresse eines Datenobjektes** ..... `p = &i;`
- **Zuweisen des Wertes `NULL`** ..... `p = NULL;`

Operationen mit Zeigern

Sei ein Datenobjekt, zum Beispiel `int i = 5;` angelegt.

- **Deklarieren eines Zeigers** ..... `int *p;`
- **Zuweisen der Adresse eines Datenobjektes** ..... `p = &i;`
- **Zuweisen des Wertes `NULL`** ..... `p = NULL;`
- **Dereferenzieren**, um das Datenobjekt an der Zeigeradresse auszulesen oder zu verändern ..... `*p = 8;`

Operationen mit Zeigern

Sei ein Datenobjekt, zum Beispiel `int i = 5;` angelegt.

- **Deklarieren eines Zeigers** ..... `int *p;`
- **Zuweisen der Adresse eines Datenobjektes** ..... `p = &i;`
- **Zuweisen des Wertes `NULL`** ..... `p = NULL;`
- **Dereferenzieren**, um das Datenobjekt an der Zeigeradresse auszulesen oder zu verändern ..... `*p = 8;`
- **Vergleichen** mit anderen Zeigern oder mit **`NULL`**
  ..... `if (p == NULL)`
  Vergleiche auf gleich (`==`) und ungleich (`!=`) zulässig.

Operationen mit Zeigern

Sei ein Datenobjekt, zum Beispiel `int i = 5;` angelegt.

- **Deklarieren eines Zeigers** ..... `int *p;`
- **Zuweisen der Adresse eines Datenobjektes** ..... `p = &i;`
- **Zuweisen des Wertes `NULL`** ..... `p = NULL;`
- **Dereferenzieren**, um das Datenobjekt an der Zeigeradresse auszulesen oder zu verändern ..... `*p = 8;`
- **Vergleichen** mit anderen Zeigern oder mit **`NULL`**
  ..... `if (p == NULL)`
  Vergleiche auf gleich (`==`) und ungleich (`!=`) zulässig.
- **Inkrementieren** eines Zeigers ..... `p++;`
  Innerhalb eines Feldes wird der Zeiger auf die Anfangsadresse des nächsten Feldelementes gesetzt.

Operationen mit Zeigern

Sei ein Datenobjekt, zum Beispiel `int i = 5;` angelegt.

- **Deklarieren eines Zeigers** ..... `int *p;`
- **Zuweisen der Adresse eines Datenobjektes** ..... `p = &i;`
- **Zuweisen des Wertes `NULL`** ..... `p = NULL;`
- **Dereferenzieren**, um das Datenobjekt an der Zeigeradresse auszulesen oder zu verändern ..... `*p = 8;`
- **Vergleichen** mit anderen Zeigern oder mit **`NULL`**
  ..... `if (p == NULL)`
  Vergleiche auf gleich (`==`) und ungleich (`!=`) zulässig.
- **Inkrementieren** eines Zeigers ..... `p++;`
  Innerhalb eines Feldes wird der Zeiger auf die Anfangsadresse des nächsten Feldelementes gesetzt.
- **Dekrementieren** eines Zeigers ..... `p--;`
  Innerhalb eines Feldes wird der Zeiger auf die Anfangsadresse des vorherigen Feldelementes gesetzt.

Zeichenketten (Strings)

```
"Welcome"
                    "Benventuti"
          "Bienvenida"
                "Zapraszamy"
 "Bineveniti"
                     "Isten hozott"
"Willkommen"
```

Eine Zeichenkette ist eine endliche Abfolge von Symbolen.

Zeichenkette in C

Eine **Zeichenkette** (engl. *string*) in C ist ein *Feld*, dessen Elemente (Zeichen) vom Typ `char` (character) sind. Jede Zeichenkette wird abgeschlossen mit dem Ende-Symbol `'\0'`.

```c
#include <stdio.h>

int main(void)
{
  char z[6] = {'H', 'a', 'l', 'l', 'o', '\0'};

  printf("%s\n", z);
  return 0;
}
```

```
Hallo
```

Zeichenkette in C

- Als Zeichen dürfen auch *Steuerzeichen* wie z.B. Zeilenumbruch `'\n'` oder Tabulator `'\t'` vorkommen

Zeichenkette in C

- Als Zeichen dürfen auch *Steuerzeichen* wie z.B. Zeilenumbruch `'\n'` oder Tabulator `'\t'` vorkommen
- Die Zeichenkette *endet beim ersten Vorkommen* des Ende-Symbols `'\0'`, weitere Ende-Symbole dahinter stören aber nicht

Zeichenkette in C

- Als Zeichen dürfen auch *Steuerzeichen* wie z.B. Zeilenumbruch `'\n'` oder Tabulator `'\t'` vorkommen
- Die Zeichenkette *endet beim ersten Vorkommen* des Ende-Symbols `'\0'`, weitere Ende-Symbole dahinter stören aber nicht
- Fehlt das Ende-Symbol, wird zur Ausgabe oder in Bibliotheksfunktionen der Speicher über das Feldende hinaus weiter ausgelesen oder beschrieben (Puffer-Überlauf), wodurch andere Daten im Speicher zerstört werden können.

Kurznotation für Zeichenketten-Literale

```c
#include <stdio.h>

int main(void)
{
  char z[6] = "Hallo";

  printf("%s\n", z);
  return 0;
}
```

```
Hallo
```

Anstelle der Aufzählung aller Feldelemente wie `{'H', 'a', 'l', 'l', 'o', '\0'}` kann man auch die Zeichen der Zeichenkette direkt hintereinanderschreiben und in Anführungszeichen einbetten, zum Beispiel `"Hallo"`. Das Ende-Symbol wird dann automatisch ergänzt und muss nicht mit hingeschrieben werden.

Zeichenketten am Monitor ausgeben

```c
#include <stdio.h>

int main(void)
{
  char h[6] = "Hallo";
  char w[6] = "Welt!";

  printf("%s %s\n", h, w);
  return 0;
}
```

```
Hallo Welt!
```

- Im **`printf`**-Formatstring markiert der Platzhalter **`%s`** eine Zeichenkette

Zeichenketten am Monitor ausgeben

```c
#include <stdio.h>

int main(void)
{
  char h[6] = "Hallo";
  char w[6] = "Welt!";

  printf("%s %s\n", h, w);
  return 0;
}
```

```
Hallo Welt!
```

- Im **`printf`**-Formatstring markiert der Platzhalter **`%s`** eine Zeichenkette
- Als *Variablenname* der Zeichenkette wird der *Feldname* angegeben

Zeichenketten am Monitor ausgeben

```c
#include <stdio.h>

int main(void)
{
  char h[6] = "Hallo";
  char w[6] = "Welt!";

  printf("%s %s\n", h, w);
  return 0;
}
```

```
Hallo Welt!
```

- Im **`printf`**-Formatstring markiert der Platzhalter **`%s`** eine Zeichenkette
- Als *Variablenname* der Zeichenkette wird der *Feldname* angegeben
- Ausgabe des Zeichenfeldes bis zum ersten Ende-Symbol

Zeichenketten am Monitor ausgeben

```c
#include <stdio.h>

int main(void)
{
  char h[6] = "Hallo";
  char w[6] = "Welt!";

  printf("%s %s\n", h, w);
  return 0;
}
```

```
Hallo Welt!
```

- Im **`printf`**-Formatstring markiert der Platzhalter **`%s`** eine Zeichenkette
- Als *Variablenname* der Zeichenkette wird der *Feldname* angegeben
- Ausgabe des Zeichenfeldes bis zum ersten Ende-Symbol
- *Rechtsbündig ausgeben:* z.B. **`%20s`** stellt der Zeichenkette in Ausgabe Leerzeichen voran, so dass insgesamt 20 Zeichen erscheinen

Zeichenkette von Tastatur eingeben mit `scanf`

```c
#include <stdio.h>

#define MAX_STRINGLAENGE 256

int main(void)
{
  char z[MAX_STRINGLAENGE]; //Feld anlegen und Speicherplatz dafuer ausfassen

  printf("Texteingabe: ");
  scanf("%s", z);
  printf("Ihre Eingabe bis zum ersten Leerzeichen: %s\n", z);
  return 0;
}
```

Zeichenkette von Tastatur eingeben mit `scanf`

```c
#include <stdio.h>

#define MAX_STRINGLAENGE 256

int main(void)
{
  char z[MAX_STRINGLAENGE]; //Feld anlegen und Speicherplatz dafuer ausfassen

  printf("Texteingabe: ");
  scanf("%s", z);
  printf("Ihre Eingabe bis zum ersten Leerzeichen: %s\n", z);
  return 0;
}
```

- Da ein Feld in C durch seine Anfangsadresse erfasst wird, beinhaltet die Variable **`z`** eine *Adresse*. Folglich muss bei Zeichenketten kein **`&`** vor den Variablennamen in **`scanf`** geschrieben werden wie bei den elementaren Datentypen

Zeichenkette von Tastatur eingeben mit `scanf`

```c
#include <stdio.h>

#define MAX_STRINGLAENGE 256

int main(void)
{
  char z[MAX_STRINGLAENGE]; //Feld anlegen und Speicherplatz dafuer ausfassen

  printf("Texteingabe: ");
  scanf("%s", z);
  printf("Ihre Eingabe bis zum ersten Leerzeichen: %s\n", z);
  return 0;
}
```

- Da ein Feld in C durch seine Anfangsadresse erfasst wird, beinhaltet die Variable **`z`** eine *Adresse*. Folglich muss bei Zeichenketten kein **`&`** vor den Variablennamen in **`scanf`** geschrieben werden wie bei den elementaren Datentypen
- **`scanf`** übernimmt die Eingabe bis zum ersten Leerzeichen.
  (Idee: einen Befehl eingeben)

Zeichenkette von Tastatur eingeben mit `scanf`

```c
#include <stdio.h>

#define MAX_STRINGLAENGE 256

int main(void)
{
  char z[MAX_STRINGLAENGE]; //Feld anlegen und Speicherplatz dafuer ausfassen

  printf("Texteingabe: ");
  scanf("%s", z);
  printf("Ihre Eingabe bis zum ersten Leerzeichen: %s\n", z);
  return 0;
}
```

- Da ein Feld in C durch seine Anfangsadresse erfasst wird, beinhaltet die Variable **`z`** eine *Adresse*. Folglich muss bei Zeichenketten kein **`&`** vor den Variablennamen in **`scanf`** geschrieben werden wie bei den elementaren Datentypen
- **`scanf`** übernimmt die Eingabe bis zum ersten Leerzeichen. (Idee: einen Befehl eingeben)
- **`scanf`** prüft leider nicht, ob die erfasste Eingabe in den reservierten Speicherbereich für die Zeichenkette passt und schreibt ggf. darüber hinaus.

Zeichenkette von Tastatur eingeben

```c
#include <stdio.h>
#define MAX_STRINGLAENGE 256

int main(void)
{
  char z[MAX_STRINGLAENGE]; //Feld anlegen und Speicherplatz dafuer ausfassen
  char c;
  int i = 0;

  printf("Texteingabe: ");
  do
  {
    c = getchar();
    z[i] = (char) c;
    i++;
  } while ((c != '\n') && (i < MAX_STRINGLAENGE));
  z[i] = '\0';

  printf("Ihre Eingabe: %s\n", z);
  return 0;
}
```

- Eine Schleife mit wiederholten **`getchar()`**-Anweisungen umgeht die Probleme mit **`scanf`** und erlaubt es, ein Zeichen festzulegen, mit dem die Eingabe endet (z.B. **`'\n'`** bei Drücken der Enter-Taste)
- Alternative Bibliotheksfunktionen wie **`fgets(...)`** ebenfalls nutzbar, dafür aber etliche Voreinstellungen zu treffen (Eingabequelle umlenken von Datei zu Tastatur)

Zahlenwerte in Zeichenketten konvertieren

```c
#include <stdio.h>
#define MAX_STRINGLAENGE 256

int main(void)
{
  char z[MAX_STRINGLAENGE];
  double m = 449.56;

  sprintf(z, "%lf", m);
  printf("Zeichenkette z: %s\n", z);
  return 0;
}
```

- Funktion **`sprintf`** in **`stdio.h`** verfügbar und arbeitet genauso wie **`printf`** mit dem einzigen Unterschied, dass die Ausgabe nicht am Monitor erfolgt, sondern in eine Zeichenkette
- Name der Zeichenkette (hier **`z`**) als *erstes Argument* an **`sprintf`** übergeben

Zahlenwerte in Zeichenketten konvertieren

```c
#include <stdio.h>
#define MAX_STRINGLAENGE 256

int main(void)
{
  char z[MAX_STRINGLAENGE];
  double m = 449.56;

  sprintf(z, "%lf", m);
  printf("Zeichenkette z: %s\n", z);
  return 0;
}
```

- Funktion **`sprintf`** in **`stdio.h`** verfügbar und arbeitet genauso wie **`printf`** mit dem einzigen Unterschied, dass die Ausgabe nicht am Monitor erfolgt, sondern in eine Zeichenkette
- Name der Zeichenkette (hier **`z`**) als *erstes Argument* an **`sprintf`** übergeben
- Mit **`sprintf`** leicht vorformatierte Zeichenketten erzeugbar, auch lassen sich damit mehrere Zeichenketten hintereinander in eine Zeichenkette zusammenfügen (*„konkatenieren“*)

Bibliothek von Zeichenkettenfunktionen

`string.h`

Standardbibliothek. Dort viele nützliche, aber auch manche gefährliche Funktion auf Zeichenketten verfügbar

strlen

ermittelt die *Anzahl Zeichen* in einer Zeichenkette (Stringlänge)

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

int main(void)
{
  char z[] = "((5+3)*(7-4)-2)/5";
  int laenge;

  laenge = (int) strlen(z);
  printf("Anzahl Zeichen: %d\n", laenge); // 17
  return 0;
}
```

- Das Ende-Zeichen `'\0'` wird nicht mitgezählt.
- `strlen` läuft den String ab, bis Ende-Zeichen erreicht

strcpy

kopiert eine Zeichenkette elementweise

```c
#include <stdio.h>
#include <string.h>
#define MAX_STRINGLAENGE 256

int main(void)
{
  char quellstring[MAX_STRINGLAENGE] = "Ready Steady Go";
  char z1[MAX_STRINGLAENGE];

  strcpy(z1, quellstring);
  printf("%s\n", z1);
  return 0;
}
```

- Syntax: **`strcpy(zielstring, quellstring);`**
- vorheriger Inhalt von **`zielstring`** wird überschrieben
- **`quellstring`** elementweise in Speicherbereich des **`zielstring`** kopiert einschließlich Ende-Zeichen
- **`strncpy(ziel,quelle,n);`** kopiert die ersten **`n`** Zeichen, aber setzt kein Ende-Zeichen

strcmp

vergleicht zwei Zeichenketten
bzgl. ihrer alphabetischen Anordnung (ASCII-Werte)

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

int main(void)
{
  char r[] = "rot";
  char b[] = "blau";
  char t[] = "rot";

  printf("Vergleich: blau < rot: %d\n", strcmp(b, r));
  printf("Vergleich: rot > blau: %d\n", strcmp(r, b));
  printf("Vergleich: rot == rot: %d\n", strcmp(r, t));
  return 0;
}
```

```
Vergleich: blau < rot: -1
Vergleich: rot > blau: 1
Vergleich: rot == rot: 0
```

- `strcmp(x,y);` liefert 0, wenn `x == y` (beide Strings gleich)
- `strcmp(x,y);` liefert -1, wenn `x < y` (x alphabetisch vor y)
- `strcmp(x,y);` liefert 1, wenn `x > y` (x alphabetisch hinter y)
- Alphabetische Ordnung gemäß ASCII-Werten der Zeichen

Alphabet durch ASCII-Werte gegeben

	32	0	48	@	64	P	80	`	96	p	112
!	33	1	49	A	65	Q	81	a	97	q	113
"	34	2	50	B	66	R	82	b	98	r	114
#	35	3	51	C	67	S	83	c	99	s	115
$	36	4	52	D	68	T	84	d	100	t	116
%	37	5	53	E	69	U	85	e	101	u	117
&	38	6	54	F	70	V	86	f	102	v	118
'	39	7	55	G	71	W	87	g	103	w	119
(	40	8	56	H	72	X	88	h	104	x	120
)	41	9	57	I	73	Y	89	i	105	y	121
*	42	:	58	J	74	Z	90	j	106	z	122
+	43	;	59	K	75	[	91	k	107	{	123
,	44	<	60	L	76	\	92	l	108	|	124
-	45	=	61	M	77	]	93	m	109	}	125
.	46	>	62	N	78	^	94	n	110	~	126
/	47	?	63	O	79	_	95	o	111	□	127

ASCII: American Standard Code for Information Interchange

Deutsche Landeshauptstädte alphabetisch sortieren

www.stepmap.de

Selectionsort auf Zeichenketten (selectionsort-strings.c)

```c
int main(void)
{
  char datenfeld[N][13] = {"Kiel", "Schwerin", "Hamburg", "Bremen",
                           "Berlin", "Potsdam", "Hannover", "Magdeburg",
                           "Duesseldorf", "Dresden", "Erfurt", "Wiesbaden",
                           "Mainz", "Saarbruecken", "Stuttgart", "Muenchen"};
                            //von Nord nach Sued
  int i;

  selectionsort(datenfeld); //in-place sortieren
  for(i = 0; i < N; i++)
  {
    printf("%s\n", datenfeld[i]); //Feldelemente ausgeben
  }
  printf("\n");
  return 0;
}
```

Anfangsanordnung der Städte von Nord nach Süd

Selectionsort auf Zeichenketten (selectionsort-strings.c)

```c
#include <stdio.h>
#include <string.h>
#define N  16  //Anzahl zu sortierender Werte

void selectionsort(char a[N][13])     //aufsteigend sortieren
{
    int i, k, min;
    char t[13];

    for(i = 0; i < N-1; i++)
    {
        min = i;
        for(k = i+1; k < N; k++)  //Kleinstes Element im noch
        {                         //zu sortierenden Teilfeld finden
            if(strcmp (a[k], a[min]) == -1)
            {
              min = k;
            }
        }
        if (i != min)
        {
          strcpy(t, a[min]);          //Vertausche kleinstes Element mit
          strcpy(a[min], a[i]);       //Anfangselement im noch zu
          strcpy(a[i], t);            //sortierenden Teilfeld
        }
    }
    return;
}
```

```
Berlin
Bremen
Dresden
Duesseldorf
Erfurt
Hamburg
Hannover
Kiel
Magdeburg
Mainz
Muenchen
Potsdam
Saarbruecken
Schwerin
Stuttgart
Wiesbaden
```

strstr

ermittelt das *erste Vorkommen* eines Teilstrings in einer Zeichenkette

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

int main(void)
{
  char z[] = "Fischers Fritze fischte frische Fische.";
  char *p;

  p = strstr(z, "Fritze"); //platziert Zeiger p auf das
                           //erste Vorkommen von Fritze in z

  printf("%s\n", p);
  return 0;
}
```

```
Fritze fischte frische Fische.
```

- `strstr` liefert einen Zeiger auf die Stelle in der Zeichenkette zurück, an welcher der Teilstring erstmalig beginnt
- Ist der gegebene Teilstring nicht in der Zeichenkette enthalten, wird `NULL` zurückgegeben

Die Funktionsbibliothek `ctype.h`

enthält nützliche Funktionen auf einzelnen Zeichen (characters)

`int tolower(int c)` ..... wandelt Groß- in Kleinbuchstabe um
`int toupper(int c)` ..... wandelt Klein- in Großbuchstabe um

Testfunktionen (liefern `1`, wenn Bedingung erfüllt, sonst `0`)

`int isalnum(int c)` . alphanumerisches Zeichen? (a-z, A-Z, 0-9)
`int isalpha(int c)` ..... Test auf Buchstabe (a-z, A-Z)
`int iscntrl(int c)` ..... Test auf Steuerzeichen (\n, ...)
`int isdigit(int c)` ..... Test auf Dezimalziffer (0-9)
`int islower(int c)` ..... Test auf Kleinbuchstabe (a-z)
`int isblank(int c)` ..... Test auf Leerzeichen
`int isupper(int c)` ..... Test auf Großbuchstabe (A-Z)
`int isxdigit(int c)` .. Test auf Hexadezimalziffer (0-9, a-f, A-F)

weitere Funktionen siehe Wikibook C-Programmierung

Arbeiten mit Dateien in C

Textdateien anlegen, schreiben und lesen mittels C-Programm

Eine Textdatei (er)öffnen und schließen

Damit eine *Textdatei* beschrieben oder ausgelesen werden kann, muss sie zuvor *angelegt* bzw. *geöffnet* werden. Es können mehrere Textdateien gleichzeitig geöffnet sein. Um sie unterscheiden zu können, wird beim Öffnen oder Anlegen einer Datei ein *Dateihandle* vergeben, das eine *Adresse* in einem speziellen Speicherbereich ist.

Sobald eine Datei nicht mehr zum Lesen oder Schreiben benötigt wird, *schließt* man sie, denn die Anzahl gleichzeitig offener Dateien ist durch das Betriebssystem begrenzt (oft auf 256 Dateien).

Eine Textdatei (er)öffnen und schließen

```c
#include <stdio.h>

int main(void)
{
  FILE *handle;

  handle = fopen("dateiname.txt", "w+");

  /* ... Lese- und Schreiboperationen in der Datei */

  fclose(handle);
  return 0;
}
```

- **`fopen`** gibt ein Dateihandle zurück, wenn das Öffnen oder Anlegen erfolgreich war. Der erste Parameter ist der Dateiname (frei wählbare Zeichenkette).
- Der zweite Parameter (z.B. **`w+`**) gibt, ebenfalls als Zeichenkette, den Modus zum Öffnen vor
- **`fclose`** schließt die Datei mit dem übergebenen Handle

Modi zum Dateiöffnen

`r`	Datei nur zum Lesen öffnen (*read*)
`w`	Datei nur zum Schreiben öffnen (*write*), alter Inhalt gelöscht
`a`	Daten am Dateiende anhängen (*append*)
`r+`	existierende Datei zum Lesen und Schreiben öffnen
`w+`	Datei zum Lesen und Schreiben öffnen, ggf. Datei anlegen
`a+`	Datei zum Lesen und Schreiben am Dateiende öffnen

In eine Textdatei schreiben mit `fprintf`

```c
#include <stdio.h>

int main(void)
{
  FILE *handle;

  handle = fopen("dateiname.txt", "w+"); //Datei anlegen bzw. oeffnen
  if (handle == NULL)
  {
    printf("Fehler beim Dateioeffnen.\n");
    return 1;
  }

  fprintf(handle, "Hallo Welt!\n"); //Text in die Datei schreiben
  fclose(handle);
  return 0;
}
```

- **`fprintf`** arbeitet genauso wie **`printf`**, nur dass es statt auf den Monitor in eine Datei (*file*) schreibt
- Dateihandle muss an **`fprintf`** übergeben werden, damit klar ist, in welche Datei geschrieben werden soll, wenn mehrere offen sind

Datei einlesen mit `fscanf` (z13.c)

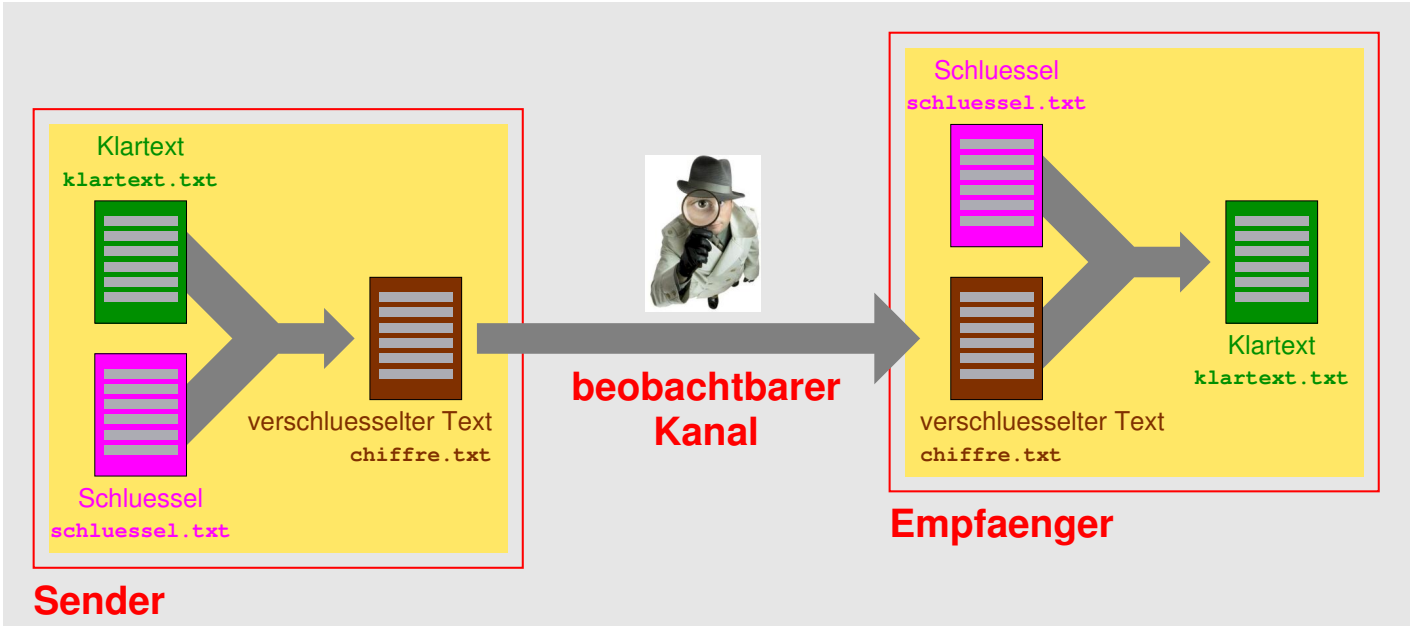
```c
#include <stdio.h>
#define MAX_STRINGLAENGE 65536

int main(void)
{
  FILE *handle;
  char textfeld[MAX_STRINGLAENGE];
  int i;

  for(i = 0; i < MAX_STRINGLAENGE; i++)
  {
    textfeld[i] = '\0'; //Textfeld mit Ende-Symbolen initialisieren
  }

  handle = fopen("dateiname.txt", "r"); //Datei zum Lesen oeffnen
  if (handle == NULL)
  {
    printf("Fehler beim Dateioeffnen.\n");
    return 1;
  }

  fscanf(handle, "%65535c", textfeld); //Dateiinhalt in textfeld kopieren
  fclose(handle);
  printf("%s\n", textfeld);
  return 0;
}
```

Größe einer Datei in Byte bestimmen (z14.c)

```c
#include <stdio.h>

#include <sys/stat.h>    //unter Linux
#include <sys/types.h>   //unter Linux
//#include <sys\stat.h>    //unter Windows

int main(void)
{
  struct stat attribute; //Zugriff auf Dateiattribute, darunter Dateilaenge
  int laenge;

  stat("dateiname.txt", &attribute); //Record der Dateiattribute auslesen
  laenge = (int) attribute.st_size;  //vordefinierter Name fuer Dateilaenge
  printf("Dateilaenge in Byte: %d\n", laenge);
  return 0;
}
```

- Record **`struct stat`** definiert einen Typ für einen Datensatz, der die kompletten Attribute einer Datei aufnehmen kann
- **`st_size`** ist eine Komponente darin, die die Dateilänge in Byte angibt
- weitere Komponenten stehen für weitere Attributwerte, z.B. die eingestellten Schreib- und Leserechte

Textverschlüsselung mit One-Time-Pad



- *einfaches* und bei richtiger Anwendung *beweisbar sicheres* Kryptoverfahren
- Schlüssel: zufällige Abfolge von Zeichen, genauso viele Zeichen wie Klartext. Gleicher Schlüssel für Sender und Empfänger
- Sender und Empfänger haben Schlüssel *auf Vorrat* ausgetauscht
- Schlüssel nur *einmal* („one time“) zum Ver- und Entschlüsseln verwendet

Klartext

Zu verschlüsselnder Text. Hier: 800 Zeichen in 10er-Blöcken

```
Lorem ipsu  m dolor si  t amet, co  nsetetur s  adipscing
elitr, sed  diam nonum  y eirmod t  empor invi  dunt ut la
bore et do  lore magna  aliquyam e  rat, sed d  iam volupt
ua. At ver  o eos et a  ccusam et   justo duo   dolores et
ea rebum.   Stet clita  kasd guber  gren, no s  ea takimat
a sanctus   est Lorem   ipsum dolo  r sit amet  . Lorem ip
sum dolor   sit amet,   consetetur  sadipscing  elitr, sed
diam nonu   my eirmod   tempor inv  idunt ut l  abore et d
olore magn  a aliquya   m erat, se  d diam vol  uptua.  At
vero eos e  t accusam   et justo d  uo dolores  et ea rebu
m. Stet cl  ita kasd g  ubergren,   no sea tak  imata sanc
tus est Lo  rem ipsum   dolor sit   amet. Lore  m ipsum do
lor sit am  et, conset  etur sadip  scing elit  r, sed dia
m nonumy e  irmod temp  or invidun  t ut labor  e et dolor
e magna al  iquyam era  t, sed dia  m voluptua  . At vero
eos et acc  usam et ju  sto duo do  lores et e  a rebum.
```

Klartext als *ASCII-Textdatei* aus *druckbaren Zeichen* (ASCII-Werte 32 bis 127)

Schlüsseltext

Zeichenkette aus zufälligen Zeichen. Hier: 800 Zeichen in 10er-Blöcken

```
*U%{YL0Wq"  □_$7>MyAx4  ?H/C69/9nS  p,'=1Y~Hnw  |.yv~59=H:
\fGtvObJ?*  |@zrq ji@O  m~(Z}oprf?  T0~cZz$]u-  Of2oQ>:~T'
|lOWi?G[%0  U/7ud\[K,U  QfS%P]'{nV  Au{E{QRp6.  KMyx"J^upl
EJl|f[S(PY  >KsFEV6{H}  zjHX_8E%96  ul=b\V'O6:  ^7H□Z'7Lq.
^'#J0Ah}_3  dmlw$m?$Gg  [f3Ls8\#yc6  sR"F7;3#3W  r{XYW># Sz
:^pQ-$1s[l  &~}yfkP%xs  xEo5!ADNeu  A@bg>dL]3□  -Xw!P+S pZ
&{PjI5'uBq  3CUW33V'M[  \-.|C\.LVP  G::jOm@0x1  Sj^f|q?MmX
_{.fKu _[I  aHAC!hY],'  JF3l(p:g:0  D-q\p^F/;g  FWkL8.GE,Y
\KMy;SWMn}  )<'|O},aMK  cy$VAK#:vT  e.Qj[m;=95  Ee0zTzaFOM
{'!'soxNf)  #B^h'uzWo+  jg@0l}goH}  7v_E\in□+L  gSBa*1fVMK
9&H!gix Xe  eASdlK?a,j  @.'Vz|B:)#  %"C.z(;{w'  Bg|nQ>'g8z
dz4v$XFI@D  \{?S"ZWZfk  $dS\^7W-{j  HAJm|@dU(;  X![nF?A"xy
0%tm5 ~6jk  2t+QXyX{43  izs*"@G[4@  {2wf)ZHz2H  sE oyTXtGW
%'m2s!snTj  Tq.:'%#pwI  >L$qK-^~ q  T[cEn□|r~%  n77JZnou9S
7=S(L'0mrB  ln4nlL[/M;  yu^,S2U YX  PFJ39F!%5d  dn)g4}APDp
n@d9Rr9,nB  R+&@TF'Z$u  _1e} &aYxy  K2b= zr%&3  ylZG□bt{Yx
```

Zufallszeichen *gleichverteilt* und *unabhängig voneinander*

Verschlüsseln

Es wird zeichenweise verschlüsselt. Sei `t[i]` das *Klartextzeichen* und `s[i]` das *Schlüsselzeichen* an der `i`-ten Position der jeweiligen Zeichenkette ($0 \leq$ `i` $<$ `strlen(t)` und $0 \leq$ `i` $<$ `strlen(s)`).
Dann ergibt sich das *Chiffre-Zeichen* `c[i]` wie folgt:

```
c[i] = ((t[i]-32) + (s[i]-32)) % 96 + 32
```

Verschlüsseln

Es wird zeichenweise verschlüsselt. Sei `t[i]` das *Klartextzeichen* und `s[i]` das *Schlüsselzeichen* an der `i`-ten Position der jeweiligen Zeichenkette (`0` $\leq$ `i` $<$ `strlen(t)` und `0` $\leq$ `i` $<$ `strlen(s)`).
Dann ergibt sich das *Chiffre-Zeichen* `c[i]` wie folgt:

```
c[i] = ((t[i]-32) + (s[i]-32)) % 96 + 32
```

Beispiel: `t[i] = 'L'` (ASCII-Wert 76) und `s[i] = '*'` (42)
`c[i] = 'V'` (86)

Verschlüsseln

Es wird zeichenweise verschlüsselt. Sei `t[i]` das *Klartextzeichen* und `s[i]` das *Schlüsselzeichen* an der `i`-ten Position der jeweiligen Zeichenkette ($0 \leq$ `i` $<$ `strlen(t)` und $0 \leq$ `i` $<$ `strlen(s)`).
Dann ergibt sich das *Chiffre-Zeichen* `c[i]` wie folgt:

```
c[i] = ((t[i]-32) + (s[i]-32)) % 96 + 32
```

Beispiel: `t[i] = 'L'` (ASCII-Wert 76) und `s[i] = '*'` (42)
`c[i] = 'V'` (86)

- `t[i]` und `s[i]` sind jeweils *ASCII-Werte* druckbarer Zeichen, also Zahlen zwischen 32 und 127

Verschlüsseln

Es wird zeichenweise verschlüsselt. Sei `t[i]` das *Klartextzeichen* und `s[i]` das *Schlüsselzeichen* an der `i`-ten Position der jeweiligen Zeichenkette ($0 \leq$ `i` $<$ `strlen(t)` und $0 \leq$ `i` $<$ `strlen(s)`).
Dann ergibt sich das *Chiffre-Zeichen* `c[i]` wie folgt:

```
c[i] = ((t[i]-32) + (s[i]-32)) % 96 + 32
```

Beispiel: `t[i] = 'L'` (ASCII-Wert 76) und `s[i] = '*'` (42)
`c[i] = 'V'` (86)

- `t[i]` und `s[i]` sind jeweils *ASCII-Werte* druckbarer Zeichen, also Zahlen zwischen 32 und 127
- Das resultierende Chiffre-Zeichen ist ebenfalls ein ASCII-Wert zwischen 32 und 127

Verschlüsseln

Es wird zeichenweise verschlüsselt. Sei `t[i]` das *Klartextzeichen* und `s[i]` das *Schlüsselzeichen* an der `i`-ten Position der jeweiligen Zeichenkette ($0 \leq$ `i` $<$ `strlen(t)` und $0 \leq$ `i` $<$ `strlen(s)`).
Dann ergibt sich das *Chiffre-Zeichen* `c[i]` wie folgt:

```
c[i] = ((t[i]-32) + (s[i]-32)) % 96 + 32
```

Beispiel: `t[i] = 'L'` (ASCII-Wert 76) und `s[i] = '*'` (42)
`c[i] = 'V'` (86)

- `t[i]` und `s[i]` sind jeweils *ASCII-Werte* druckbarer Zeichen, also Zahlen zwischen 32 und 127
- Das resultierende Chiffre-Zeichen ist ebenfalls ein ASCII-Wert zwischen 32 und 127
- Es gibt insgesamt 96 verschiedene druckbare Zeichen.

Verschlüsseln

Es wird zeichenweise verschlüsselt. Sei `t[i]` das *Klartextzeichen* und `s[i]` das *Schlüsselzeichen* an der `i`-ten Position der jeweiligen Zeichenkette ($0 \leq$ `i` $<$ `strlen(t)` und $0 \leq$ `i` $<$ `strlen(s)`).
Dann ergibt sich das *Chiffre-Zeichen* `c[i]` wie folgt:

```
c[i] = ((t[i]-32) + (s[i]-32)) % 96 + 32
```

Beispiel: `t[i] = 'L'` (ASCII-Wert 76) und `s[i] = '*'` (42)
`c[i] = 'V'` (86)

- `t[i]` und `s[i]` sind jeweils *ASCII-Werte* druckbarer Zeichen, also Zahlen zwischen 32 und 127
- Das resultierende Chiffre-Zeichen ist ebenfalls ein ASCII-Wert zwischen 32 und 127
- Es gibt insgesamt 96 verschiedene druckbare Zeichen.
- Aus einem beliebigen Klartextzeichen kann je nach Schlüsselzeichen jedes Chiffre-Zeichen werden.

Entschlüsseln

Es wird zeichenweise entschlüsselt. Sei `c[i]` das *Chiffre-Zeichen* und `s[i]` das *Schlüsselzeichen* an der `i`-ten Position der jeweiligen Zeichenkette ($0 \leq$ `i` $<$ `strlen(c)` und $0 \leq$ `i` $<$ `strlen(s)`).
Dann ergibt sich das *Klartextzeichen* `t[i]` wie folgt:

$$\texttt{t[i]} = \begin{cases} \texttt{(c[i] - s[i]) \% 96 + 128} & \text{falls } \texttt{s[i] > c[i]} \\ \texttt{c[i] - (s[i] - 32)} & \text{sonst} \end{cases}$$

Entschlüsseln

Es wird zeichenweise entschlüsselt. Sei `c[i]` das *Chiffre-Zeichen* und `s[i]` das *Schlüsselzeichen* an der `i`-ten Position der jeweiligen Zeichenkette ($0 \leq$ `i` $<$ `strlen(c)` und $0 \leq$ `i` $<$ `strlen(s)`). Dann ergibt sich das *Klartextzeichen* `t[i]` wie folgt:

$$\texttt{t[i]} = \begin{cases} \texttt{(c[i] - s[i]) \% 96 + 128} & \text{falls } \texttt{s[i] > c[i]} \\ \texttt{c[i] - (s[i] - 32)} & \text{sonst} \end{cases}$$

- Entschlüsseln ist die Umkehroperation zum Verschlüsseln.

Entschlüsseln

Es wird zeichenweise entschlüsselt. Sei `c[i]` das *Chiffre-Zeichen* und `s[i]` das *Schlüsselzeichen* an der `i`-ten Position der jeweiligen Zeichenkette (`0` $\leq$ `i` $<$ `strlen(c)` und `0` $\leq$ `i` $<$ `strlen(s)`). Dann ergibt sich das *Klartextzeichen* `t[i]` wie folgt:

$$\texttt{t[i]} = \begin{cases} \texttt{(c[i] - s[i]) \% 96 + 128} & \text{falls } \texttt{s[i] > c[i]} \\ \texttt{c[i] - (s[i] - 32)} & \text{sonst} \end{cases}$$

- Entschlüsseln ist die Umkehroperation zum Verschlüsseln.
- Es wird der gleiche Schlüssel sowohl zum Ver- als auch zum Entschlüsseln genutzt. Das heißt, die `s[i]` stimmen jeweils überein.

Entschlüsseln

Es wird zeichenweise entschlüsselt. Sei `c[i]` das *Chiffre-Zeichen* und `s[i]` das *Schlüsselzeichen* an der `i`-ten Position der jeweiligen Zeichenkette ($0 \leq$ `i` $<$ `strlen(c)` und $0 \leq$ `i` $<$ `strlen(s)`). Dann ergibt sich das *Klartextzeichen* `t[i]` wie folgt:

$$\texttt{t[i]} = \begin{cases} \texttt{(c[i] - s[i]) \% 96 + 128} & \text{falls } \texttt{s[i] > c[i]} \\ \texttt{c[i] - (s[i] - 32)} & \text{sonst} \end{cases}$$

- Entschlüsseln ist die Umkehroperation zum Verschlüsseln.
- Es wird der gleiche Schlüssel sowohl zum Ver- als auch zum Entschlüsseln genutzt. Das heißt, die `s[i]` stimmen jeweils überein.
- Durch das Mapping in den ASCII-Bereich der druckbaren Zeichen zwischen 32 und 127 erscheinen die Berechnungsvorschriften zum Ver- und Entschlüsseln etwas aufgebläht.

Programmdemo One-Time-Pad-Kryptotool

`one-time-pad-kryptotool.c` Quelltext auf Veranstaltungswebseite verfügbar

one-time-pad-krypto.c (~/Dokumente/Lehre-Einf-in-die-Prog-WiSe1415/vorlesungsteil09) - gedit

Öffnen Speichern Rückgängig

one-time-pad-krypto.c

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

#include <sys/stat.h>   /* Linux */
#include <sys/types.h>  /* Linux */
//#include <sys\stat.h> /* Windows */

#define MAX_DATEILAENGE 10000 //Byte
#define CHIFFRE_DATEINAME "chiffre.txt"

int verschluesseln(char *klartextdateiname, char *schluesseldateiname)
{
  FILE *handlet;
  FILE *handles;
  FILE *handlec;

  struct stat attribKlartext;
  struct stat attribSchluessel;

  char klartext[MAX_DATEILAENGE+1];
  char schluessel[MAX_DATEILAENGE+1];
  char chiffre[MAX_DATEILAENGE+1];
  int laengeKlartextdatei = 0;
  int laengeSchluessel = 0;
  char formatstring[10];
  int i;

  /* Dateilaengen Klartext und Schluessel ermitteln */
  stat(klartextdateiname, &attribKlartext);
  stat(schluesseldateiname, &attribSchluessel);

  laengeKlartextdatei = (int) attribKlartext.st_size;
  laengeSchluessel = (int) attribSchluessel.st_size;

  /* Zu lange Dateien begrenzen auf Maximallaenge */
  if (laengeKlartextdatei > MAX_DATEILAENGE)
  {
    laengeKlartextdatei = MAX_DATEILAENGE;
  }
  if (laengeSchluessel > MAX_DATEILAENGE)
  {
    laengeSchluessel = MAX_DATEILAENGE;
  }

  /* Schluessel mindestens so lang wie Klartext? */
  if ((laengeKlartextdatei < 1) || (laengeSchluessel < 1) || (laengeSchluessel < laengeKlartextdatei))
```

Vor dem Compilieren bitte die **`include`**-Anweisungen an das genutzte Betriebssystem anpassen

Wie lassen sich zufällige Zeichen erzeugen?

- In einer guten zufälligen Zeichenfolge sind die enthaltenen Zeichen annähernd gleichhäufig verteilt und (erscheinen) unabhängig voneinander. Beide Eigenschaften (Gleichverteilung und Autokorrelation) lassen sich statistisch bewerten.
- Einfache Idee: viele *Münzwürfe*. Jeder Münzwurf liefert ein Bit (z.B. Kopf $= 0$ und Zahl $= 1$)

Wie lassen sich zufällige Zeichen erzeugen?

- In einer guten zufälligen Zeichenfolge sind die enthaltenen Zeichen annähernd gleichhäufig verteilt und (erscheinen) unabhängig voneinander. Beide Eigenschaften (Gleichverteilung und Autokorrelation) lassen sich statistisch bewerten.
- Einfache Idee: viele *Münzwürfe*. Jeder Münzwurf liefert ein Bit (z.B. Kopf $= 0$ und Zahl $= 1$)
- Sieben aufeinanderfolgende Bit ergeben eine Binärzahl, deren dezimaler Wert zwischen 0 und 127 liegt.

Wie lassen sich zufällige Zeichen erzeugen?

- In einer guten zufälligen Zeichenfolge sind die enthaltenen Zeichen annähernd gleichhäufig verteilt und (erscheinen) unabhängig voneinander. Beide Eigenschaften (Gleichverteilung und Autokorrelation) lassen sich statistisch bewerten.
- Einfache Idee: viele *Münzwürfe*. Jeder Münzwurf liefert ein Bit (z.B. Kopf $= 0$ und Zahl $= 1$)
- Sieben aufeinanderfolgende Bit ergeben eine Binärzahl, deren dezimaler Wert zwischen 0 und 127 liegt.
- Ist der Wert zwischen 32 und 127, hat man den ASCII-Wert eines Zufallszeichens.
- Werte zwischen 0 und 31 werden ignoriert.

Wie lassen sich zufällige Zeichen erzeugen?

- In einer guten zufälligen Zeichenfolge sind die enthaltenen Zeichen annähernd gleichhäufig verteilt und (erscheinen) unabhängig voneinander. Beide Eigenschaften (Gleichverteilung und Autokorrelation) lassen sich statistisch bewerten.
- Einfache Idee: viele *Münzwürfe*. Jeder Münzwurf liefert ein Bit (z.B. Kopf $= 0$ und Zahl $= 1$)
- Sieben aufeinanderfolgende Bit ergeben eine Binärzahl, deren dezimaler Wert zwischen 0 und 127 liegt.
- Ist der Wert zwischen 32 und 127, hat man den ASCII-Wert eines Zufallszeichens.
- Werte zwischen 0 und 31 werden ignoriert.

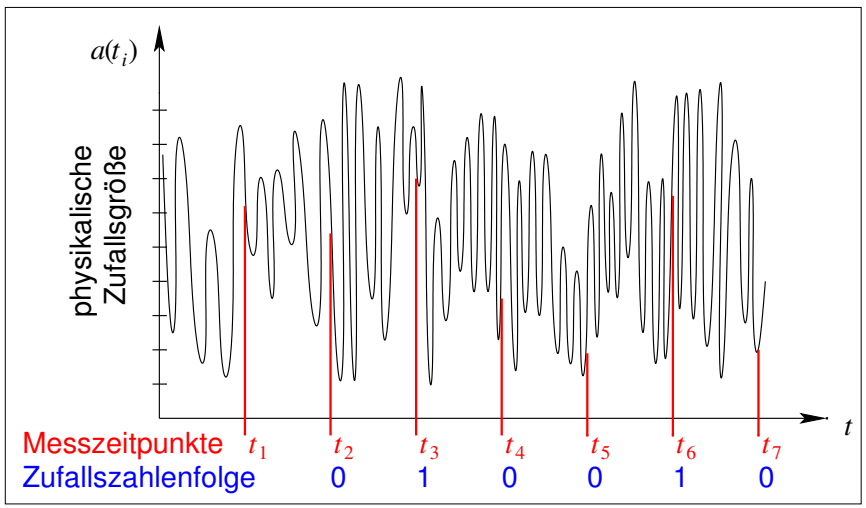
Problem: Diese Vorgehensweise ist sehr umständlich und sehr aufwendig.

Rauschgrößenmessung

z.B. thermisches Rauschen, atmosphärisches Rauschen, ...

- zeitquantisierte Erfassung einer zugrundeliegenden physikalischen Zufallsgröße und Transformation in Zufallszahlenfolge
- Transformation z.B.:

$$a(t_i) \geq a(t_{i-1}) \longrightarrow \text{Ausgabe } y(t_i) = 1$$
$$a(t_i) < a(t_{i-1}) \longrightarrow \text{Ausgabe } y(t_i) = 0$$



Rauschgrößenmessung

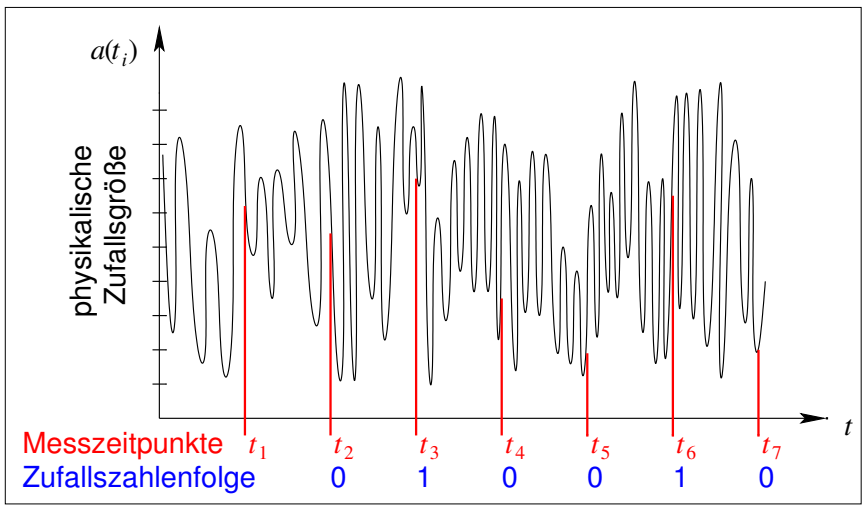
z.B. thermisches Rauschen, atmosphärisches Rauschen, ...

- zeitquantisierte Erfassung einer zugrundeliegenden physikalischen Zufallsgröße und Transformation in Zufallszahlenfolge
- Transformation z.B.:

$$a(t_i) \geq a(t_{i-1}) \longrightarrow \text{Ausgabe } y(t_i) = 1$$
$$a(t_i) < a(t_{i-1}) \longrightarrow \text{Ausgabe } y(t_i) = 0$$

Vorteile

– gute statistische Eigenschaften
– keine inhärente Reproduzierbarkeit



Rauschgrößenmessung

z.B. thermisches Rauschen, atmosphärisches Rauschen, ...

- zeitquantisierte Erfassung einer zugrundeliegenden physikalischen Zufallsgröße und Transformation in Zufallszahlenfolge
- Transformation z.B.:

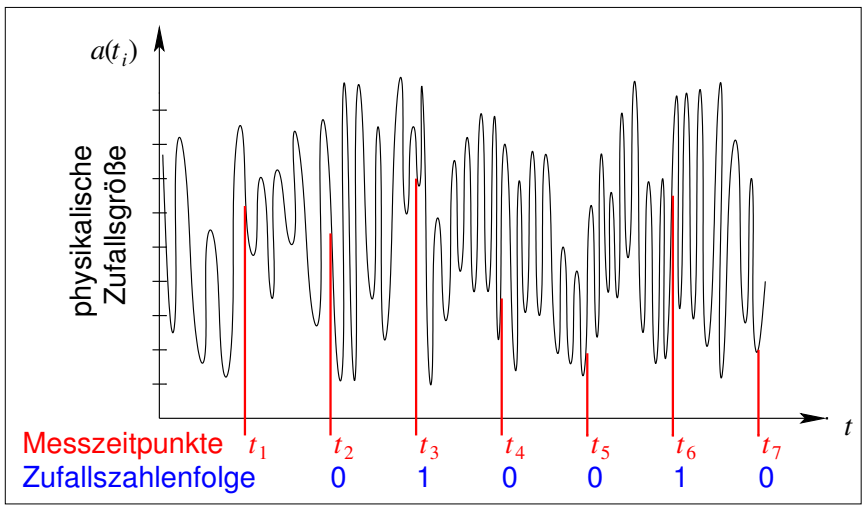
$$a(t_i) \geq a(t_{i-1}) \longrightarrow \text{Ausgabe } y(t_i) = 1$$
$$a(t_i) < a(t_{i-1}) \longrightarrow \text{Ausgabe } y(t_i) = 0$$

Vorteile

– gute statistische Eigenschaften
– keine inhärente Reproduzierbarkeit

Nachteile

– Auswirkungen von Messfehlern
– Verfügbarkeit und maximale Abtastrate abhängig von physikalischer Zufallsgröße



x^2 mod n Generator nach Blum/Shub

- **Prinzip**

  Parameter: $s, p, q \in \mathbb{N}$ mit p, q prim, $p \approx q$, $p, q \equiv 3 \mod 4$, $0 < s < pq$, $\mathrm{ggT}(s, pq) = 1$

  Startwert: $z_0 = s^2 \mod (pq)$

  Rekursion: $z_i = z_{i-1}^2 \mod (pq)$

  PZZ-Folge: $r_i = z_i \mod 2$ Es gilt: $r_i \in \{0, 1\}$

x^2 mod n Generator nach Blum/Shub

- **Prinzip**
  Parameter: $s, p, q \in \mathbb{N}$ mit p, q prim, $p \approx q$, $p, q \equiv 3 \mod 4$, $0 < s < pq$, $\mathrm{ggT}(s, pq) = 1$
  Startwert: $z_0 = s^2 \mod (pq)$
  Rekursion: $z_i = z_{i-1}^2 \mod (pq)$
  PZZ-Folge: $r_i = z_i \mod 2$ Es gilt: $r_i \in \{0, 1\}$
- **maximale Periodenlänge**
  pq

x^2 mod n Generator nach Blum/Shub

- **Prinzip**
  Parameter: $s, p, q \in \mathbb{N}$ mit p, q prim, $p \approx q$, $p, q \equiv 3 \mod 4$, $0 < s < pq$, $\mathrm{ggT}(s, pq) = 1$
  Startwert: $z_0 = s^2 \mod (pq)$
  Rekursion: $z_i = z_{i-1}^2 \mod (pq)$
  PZZ-Folge: $r_i = z_i \mod 2$ Es gilt: $r_i \in \{0, 1\}$
- **maximale Periodenlänge**
  pq

Vorteile

– perfekter Pseudozufallszahlengenerator
– leichte Implementierbarkeit
– leichte Wahl geeigneter Parameterbelegungen

$x^2 \bmod n$ Generator nach Blum/Shub

- **Prinzip**
  Parameter: $s, p, q \in \mathbb{N}$ mit p, q prim, $p \approx q$, $p, q \equiv 3 \mod 4$, $0 < s < pq$, $\mathrm{ggT}(s, pq) = 1$
  Startwert: $z_0 = s^2 \mod (pq)$
  Rekursion: $z_i = z_{i-1}^2 \mod (pq)$
  PZZ-Folge: $r_i = z_i \mod 2$ Es gilt: $r_i \in \{0, 1\}$
- **maximale Periodenlänge**
  pq

Vorteile

– perfekter Pseudozufallszahlengenerator
– leichte Implementierbarkeit
– leichte Wahl geeigneter Parameterbelegungen

Nachteile

– nur ein Bit pro Rekursionsschritt $\longrightarrow$ langsam
– kleine Periodenlänge