Programmieren in C

Prof. Dr. Peter Thiemann
Hannes Saffrich
Sommersemester 2021

Universität Freiburg
Institut für Informatik

Übungsblatt 3

Abgabe: Montag, 10.05.2021, 9:00 Uhr morgens

Wichtig. *Ab diesem Übungsblatt nehmen wir an, dass Sie verstanden haben wie* `Makefiles` *zum Kompilieren und Linken verwendet werden. Es gibt also* ***0 Punkte*** *falls* `make compile` *und* `make test` *nicht zumindest die von Ihnen bearbeiteten Aufgaben erfolgreich kompilieren und linken. Dadurch können die Tutoren ihre Zeit sinnvoller nutzen und Ihnen ausführlicheres Feedback geben. Es liegt also in Ihrer Verantwortung sicherzustellen, dass auf dem Buildserver wirklich alle relevanten Programme erfolgreich gebaut werden. Wenn Sie dabei Probleme haben melden Sie sich bitte rechtzeitig im Forum oder in den Tutoraten.*

In diesem Übungsblatt geht es um Strings, `char`s und Kommandozeilenargumente. Die zugehörigen Konzepte, insbesondere das gekonnte Jonglieren von ASCII-Codes, funktionieren auch in anderen Programmiersprachen so und sind daher allgemein relevant.

Hierzu zunächst zwei kleine Wiederholungen.

Exkurs: Zeichen

Ein `char` ist eine ganze Zahl, die in ein Byte Speicher passt - also wie ein `int`, nur kleiner:

```
char c = 23;
c = c - 42;
/* c has value -19 */
```

Wenn `char`s in Strings benutzt werden, stehen einige dieser Zahlen für darstellbare Zeichen, z.B. steht 65 für ein großes “A”, 66 für ein großes “B”, und so weiter.

Da es äußerst schmerzhaft wäre, Zeichen immer durch ihre Zahlencodes zu beschreiben, gibt es für `char`s eine alternative Schreibweise, bei der man das zu beschreibende Zeichen in single quotes schreibt:

```
char c = 65;
char d = 'A';
/* c and d both have value 65 and represent the character 'A'. */
d = d + 1;
/* d has value 66 and represents the character 'B'. */
```

Die vollständige Zuordnung von Zahlencodes zu Zeichen ist in der ASCII-Tabelle in Abbildung 1 zu sehen. Die meisten der ersten 32 Zahlen werden vom Terminal als bestimmte Steueroperationen interpretiert, die für uns momentan nicht relevant sind.

Exkurs: Strings

Strings werden in C als null-terminierte Arrays von `char`s repräsentiert. *Null-terminiert* bedeutet, dass nach dem letzten Zeichen eines Strings die Zahl `0` folgt, um zu signalisieren, dass der String zu Ende ist:

```
char s[4] = "foo";
char t[4];
t[0] = 'f';
t[1] = 'o';
t[2] = 'o';
t[3] = 0;    /* Important: 0 is not '0'! */
/* s and t are the same. */
```

Bei einem String muss man also nicht wissen wie lang das Array ist, da man einfach so lange durch den String laufen kann, bis der Wert `0` auftaucht. Das ist auch genau was die Funktion `strlen` aus `string.h` macht.

Wenn die Länge eines Strings erst zur Laufzeit bekannt ist, verwenden wir den Typ `char*`, dessen genaue Bedeutung wir erst in der nächsten Vorlesung kennen lernen werden.

Werte vom Typ `char*` verhalten sich genau so wie Arrays die `char`s enthalten:

```
#include <stdio.h>

/* argc is the number of command line arguments. */
/* argv contains the command line arguments, i.e. an array of strings. */
int main(int argc, char* argv[]) {
  if (argc > 1) {
    char* arg1 = argv[1];
    printf("arg1 = '%s'", arg1);
    char c1 = arg1[0];
    printf("c1   = '%c' and has code '%d'", c1, c1);
  }
  return 0;
}
```

Falls Sie an diesem Punkt verwirrt sein sollten, spielen Sie am besten zunächst etwas mit diesem Programm und ändern Sie es ab um Ihr Verständnis zu überprüfen.

```
$ ./program "foo"
arg1 = 'foo'
c1   = 'f' and has code 102

$ ./program ""
arg1 = ''
c1   = '' and has code 0

$ ./program
```

Decimal	Literal	Meaning
0	'\0'	NUL (null character)
1		SOH (start of heading)
2		STX (start of text)
3		ETX (end of text)
4		EOT (end of transmission)
5		ENQ (enquiry)
6		ACK (acknowledge)
7	'\a'	BEL (bell)
8	'\b'	BS (backspace)
9	'\t'	HT (horizontal tab)
10	'\n'	LF (new line)
11	'\v'	VT (vertical tab)
12	'\f'	FF (form feed)
13	'\r'	CR (carriage ret)
14		SO (shift out)
15		SI (shift in)
16		DLE (data link escape)
17		DC1 (device control 1)
18		DC2 (device control 2)
19		DC3 (device control 3)
20		DC4 (device control 4)
21		NAK (negative ack.)
22		SYN (synchronous idle)
23		ETB (end of trans. blk)
24		CAN (cancel)
25		EM (end of medium)
26		SUB (substitute)
27		ESC (escape)
28		FS (file separator)
29		GS (group separator)
30		RS (record separator)
31		US (unit separator)
32	' '	Space
33	'!'	
34	'"'	
35	'#'	
36	'$'	
37	'%'	
38	'&'	
39	'\''	Single quote '
40	'('	
41	')'	
42	'*'	
43	'+'	
44	','	
45	'-'	
46	'.'	
47	'/'	
48	'0'	
49	'1'	
50	'2'	
51	'3'	
52	'4'	
53	'5'	
54	'6'	
55	'7'	
56	'8'	
57	'9'	
58	':'	
59	';'	
60	'<'	
61	'='	
62	'>'	
63	'?'	
64	'@'	
65	'A'	
66	'B'	
67	'C'	
68	'D'	
69	'E'	
70	'F'	
71	'G'	
72	'H'	
73	'I'	
74	'J'	
75	'K'	
76	'L'	
77	'M'	
78	'N'	
79	'O'	
80	'P'	
81	'Q'	
82	'R'	
83	'S'	
84	'T'	
85	'U'	
86	'V'	
87	'W'	
88	'X'	
89	'Y'	
90	'Z'	
91	'['	
92	'\\'	Backslash \
93	']'	
94	'^'	
95	'_'	
96	'`'	
97	'a'	
98	'b'	
99	'c'	
100	'd'	
101	'e'	
102	'f'	
103	'g'	
104	'h'	
105	'i'	
106	'j'	
107	'k'	
108	'l'	
109	'm'	
110	'n'	
111	'o'	
112	'p'	
113	'q'	
114	'r'	
115	's'	
116	't'	
117	'u'	
118	'v'	
119	'w'	
120	'x'	
121	'y'	
122	'z'	
123	'{'	
124	'|'	
125	'}'	
126	'~'	
127		DEL

Abbildung 1: ASCII Table

Aufgabe 3.1 (Fingerübungen; 4 Punkte)

In dieser Aufgabe sollen Sie die Funktionen `strlen` und `strcmp` selbst implementieren und mit jeweils zwei Unittests auf Korrektheit überprüfen.

Da es sich hierbei nur um eine Fingerübung handelt, sollen die beiden Funktionen, die Unittests und die zugehörige `main`-Funktion alle in die selbe Datei `warmup.c` geschrieben werden. Sie brauchen also auch keine zugehörige Header-Datei.

(a) Definieren Sie eine Funktion `int my_strlen(char* s)`, die einen Null-terminierten String `s` als Argument nimmt und die Länge des Strings zurückgibt. Die Null-Terminierung zählt dabei nicht zur Länge, d.h. `my_strlen("foo")` soll `3` zurückgeben.

(b) Definieren Sie eine Funktion `int my_strcmp(char* s1, char* s2)`, die zwei Null-terminierten Strings `s` als Argument nimmt und `0` zurückgibt, wenn die Strings die gleichen Zeichen in der gleichen Reihenfolge enthalten. Ansonsten soll `1` zurückgegeben werden.[1]

```
$ ./warmup
warmup.c:42:test_my_strlen_1:PASS
warmup.c:43:test_my_strlen_2:PASS
warmup.c:44:test_my_strcmp_1:PASS
warmup.c:45:test_my_strcmp_2:PASS

-----------------------
4 Tests 0 Failures 0 Ignored
```

Hinweis. Eine Funktion, die ein Argument vom Typ `char*` nimmt, kann auch mit `char`-Arrays umgehen. Beispiel:

```
char s[4] = "foo";
int len = my_strlen(s);
```

[1] Die echte `strcmp` aus der Standardlibrary berechnet für ungleiche Strings eigentlich die lexikographische Ordnung. Unsere `my_strcmp` ist hier etwas einfacher.

Überblick

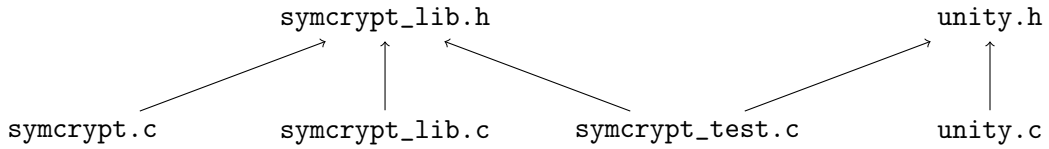
Im restlichen Übungsblatt sollen Sie eine einfache symmetrische Verschlüsselung implementieren. Nach erfolgreichem Bearbeiten des Übungsblatts haben Sie ein Programm `symcrypt`, das sich wie folgt verhält:

```
$ ./symcrypt
USAGE
  symcrypt [encrypt|decrypt] SECRET MESSAGE

  Encrypts or decrypts a MESSAGE using a shared SECRET.

$ ./symcrypt encrypt 'ilikecats' 'I am scared of broccoli!'
'3lKYeWEVfOQi[LcDgcMPYXOd'

$ ./symcrypt decrypt 'ilikecats' '3lKYeWEVfOQi[LcDgcMPYXOd'
'I am scared of broccoli!'

$ ./symcrypt decrypt 'catsarestupid' '3lKYeWEVfOQi[LcDgcMPYXOd'
'O+Ve$d_bqY` vh"Os"ZjecYs'
```

Man nennt diese Art von Verschlüsselung *symmetrisch*, da für das Verschlüsseln und Entschlüsseln einer Nachricht der selbe Schlüssel ("shared secret") benötigt wird.

Die symmetrischen Verschlüsselungen stehen im Gegensatz zu den *assymetrischen Verschlüsselungen*, wie z.B. den RSA Schlüsselpaaren mit denen Sie sich bei `git`-Servern per SSH authentifizieren können: dort wird eine Nachricht mit dem öffentlichen Schlüssel `~/.ssh/id_rsa.pub` verschlüsselt, aber zum Entschlüsseln wird ein anderer Schlüssel benötigt, nämlich der private Schlüssel `~/.ssh/id_rsa`.

Die Dateistruktur für unsere symmetrische Verschlüsselung soll dabei, analog zum vorherigen Übungsblatt, wie folgt aussehen:



`make test` soll dabei sowohl `symcrypt_test` als auch `warmup` aus der vorherigen Aufgabe ausführen.

Der Crypto-Algorithmus

Nehmen wir zunächst zur Vereinfachung an, dass der Schlüssel `secret` und die Nachricht `message` gleich lang sind:

```
message:   'I'  ' '  'a'  'm'  ' '  's'  'c'  'a'  'r'
           73   32   97   109  32   115  99   97   114

secret:    'i'  'l'  'i'  'k'  'e'  'c'  'a'  't'  's'
           105  108  105  107  101  99   97   116  115

encrypted: ?    ?    ?    ?    ?    ?    ?    ?    ?
```

Eine simple Möglichkeit `message` mit `secret` zu verschlüsseln, wäre einfach die einzelnen Zeichen zu addieren:

```
encrypted[i] = message[i] + secret[i]
```

Zum Entschlüsseln könnte man dann das `secret` wieder subtrahieren und würde wieder die ursprüngliche `message` erhalten:

```
decrypted[i] = encrypted[i] - secret[i]
```

Spielen wir das mal am Beispiel von zwei einzelnen Zeichen durch:

```
encrypted[0] = 'I' + 'i';  /* = 73 + 105 = 178 */
```

An dem Beispiel werden zwei Probleme sichtbar:

(a) `178` passt nicht in einen `char`, dessen Werte zwischen -126 und 127 liegen. Die Addition von zwei `char`s gibt in C zwar einen `int` zurück, der groß genug ist um 178 fassen zu können, aber wenn wir diesen `int` danach wieder dem `char` `encrypted[0]` zuweisen, dann stehen in diesem `char` keine sinnvollen Werte drin.[2]

(b) Wir müssen aufpassen, dass die Zeichen in `encrypted` in einem Bereich der ASCII-Tabelle liegen, der darstellbare Zeichen enthält, also keine Steueroperationen – schließlich wollen wir die verschlüsselte Nachricht danach wieder als Text ausgeben.

Um das zweite Problem zu lösen, beschränken wir uns auf `char`s, die größer gleich 32 und kleiner 127 sind. Das heißt wir fordern, dass dies für die Zeichen unserer Eingaben `secret` und `message` gilt, und stellen sicher dass dies für die Zeichen unserer Ausgaben `encrypted` und `decrypted` auch der Fall ist. Ein kurzer Blick auf die ASCII Tabelle in Abbildung 1 zeigt, dass wir mit diesem Bereich ausschließlich darstellbare Zeichen abdecken und auch fast alle darstellbaren Zeichen.

[2] Laut C-Standard darf der Compiler dann entscheiden was passieren soll. *facepalm*

Um das erste Problem zu Lösen verwenden wir beim Verschlüsseln den folgenden Trick:

- Vor dem Addieren, ziehen wir zunächst von beiden Zeichen 32 ab. Dadurch liegen die Zeichen zwischen `0` und `95 (= 127 - 32)`.
- Beim Addieren können Zahlen größer `127` entstehen. Daher speichern wir das Ergebnis der Addition in einem `int`.
- Ist die Zahl nach dem Addieren größer gleich `95`, dann subtrahieren wir zusätzlich `95`. Das hat zur Folge, dass wenn wir über den rechten Rand unseres darstellenbaren Bereich hinausschießen, wir einfach wieder vom linken Rand aus weitermachen.
- Anschließend addieren wir wieder `32` zu unserer Zahl, sodass diese im darstellbaren Bereich liegt.
- Da nun die Zahl wieder in einen `char` passt, können wir den `int` unserem Ausgabe-`char` zuweisen.

Beim Entschlüsseln kann man genau wie beim Verschlüsseln vorgehen, bis auf einen Unterschied: Da wir zwei Zeichen zwischen `0` und `95` voneinander subtrahieren statt addieren, müssen wir nicht überprüfen, ob das Zeichen nach dem Addieren größer gleich `95` ist und dann `95` abziehen, sondern wir müssen überprüfen, ob das Zeichen nach dem Subtrahieren kleiner `0` ist und in diesem Fall `95` hinzuaddieren. Es ist also genau anders herum wie beim Verschlüsseln – schließlich soll das Entschlüsseln ja gerade das Verschlüsseln rückgängig machen.

Ich empfehle an dieser Stelle das obige Beispiel mal auf Papier oder im Kopf durchzuspielen:

- beim Verschlüsseln von `'I'` mit `'i'` ergibt sich das Zeichen `'3'`; und
- beim Entschlüsseln von `'3'` mit `'i'` ergibt sich das Zeichen `'I'`.

Aufgabe 3.2 (Ver- und Entschlüsseln von einzelnen Zeichen; 4 Punkte)

(a) Definieren Sie in der `symcrypt_lib.c` eine Funktion

```
char encrypt_char(char m, char s)
```

die nach dem oben beschriebenen Algorithmus ein Zeichen `m` aus der Nachricht mit einem Zeichen `s` aus dem Secret verschlüsselt und das verschlüsselte Zeichen zurückgibt.

(b) Definieren Sie in der `symcrypt_lib.c` eine Funktion

```
char decrypt_char(char m, char s)
```

die nach dem oben beschriebenen Algorithmus ein Zeichen `m` aus einer verschlüsselten Nachricht mit einem Zeichen `s` aus dem Secret entschlüsselt und das entschlüsselte Zeichen zurückgibt.

(c) Schreiben Sie in der `symcrypt_test.c` zu beiden Funktionen jeweils mindestens 3 Unittests.

Aufgabe 3.3 (Ver- und Entschlüsseln von Strings; 3 Punkte)

(a) Definieren Sie in der `symcrypt_lib.c` eine Funktion

```
void encrypt(char* message, char* secret, char* encrypted)
```

die eine `message` mit einem `secret` verschlüsselt und die verschlüsselte Nachricht in `encrypted` schreibt. Sie können dabei annehmen, dass `encrypted` mindestens so groß ist wie `message`. Achten Sie darauf, dass `encrypted` am Ende null-terminiert sein muss.

(b) Definieren Sie in der `symcrypt_lib.c` eine Funktion

```
void decrypt(char* message, char* secret, char* decrypted)
```

die eine verschlüsselte Nachricht `message` mit einem `secret` entschlüsselt und die entschlüsselte Nachricht in `decrypted` schreibt. Sie können dabei annehmen, dass `decrypted` mindestens so groß ist wie `message`. Achten Sie darauf, dass `decrypted` am Ende null-terminiert sein muss.

(c) Schreiben Sie in der `symcrypt_test.c` zu beiden Funktionen jeweils mindestens 2 Unittests.

Beispiel:

```
char encrypted[5000];
encrypt("I am scared of broccoli!", "ilikecats", encrypted);
printf("'%s'\n", encrypted); /* prints '3lKYeWEVfOQi[LcDgcMPYXOd' */

char decrypted[5000];
decrypt("3lKYeWEVfOQi[LcDgcMPYXOd", "ilikecats", decrypted);
printf("'%s'\n", decrypted); /* prints 'I am scared of broccoli!' */
```

Wichtig. Wenn das `secret` weniger Zeichen enthält wie `message`, dann sollen die Zeichen aus dem `secret` wieder von vorne durchlaufen werden. Bei Eingaben von

```
secret:  ilikecats
message: I am scared of broccoli!
```

soll sich der Algorithmus also so verhalten als wären folgende Eingaben gemacht wurden:

```
secret:  ilikecatsilikecatsilikec
message: I am scared of broccoli!
```

Aufgabe 3.4 (Das `symcrypt`-Programm; 3 Punkte)

Schreiben Sie in `symcrypt.c` eine Funktion

```
int main(int argc, char* argv[])
```

welche die Funktionen, die Sie in der `symcrypt_lib.h` deklariert haben, verwendet um ein Program zu erhalten, was sich wie im Abschnitt *Überblick* beschrieben verhält:

- Wenn das Programm nicht mit drei Argumenten aufgerufen wird, oder das erste Argument nicht `"encrypt"` oder `"decrypt"` ist, soll die folgende Anleitung zum Benutzen des Programms ausgegeben werden und durch `return 1` ein Fehlschlagen des Programms signalisiert werden:

  ```
  $ ./symcrypt
  USAGE
    symcrypt [encrypt|decrypt] SECRET MESSAGE

    Encrypts or decrypts a MESSAGE using a shared SECRET.

  $ ./symcrypt magic ilikecats 'my secret is that i have no secrets.'
  USAGE
    symcrypt [encrypt|decrypt] SECRET MESSAGE

    Encrypts or decrypts a MESSAGE using a shared SECRET.
  ```

  Beachten Sie, dass das erste Argument in `argv[1]` steht, da in `argv[0]` der Programmname steht.

- Wenn es drei Argumente gibt und das erste Argument `"encrypt"` ist, dann soll der Schlüssel aus dem zweiten Argument verwendet werden um die Nachricht aus dem dritten Argument zu verschlüsseln und die verschlüsselte Nachricht in single quotes ausgegeben werden:

  ```
  $ ./symcrypt encrypt 'ilikecats' 'I am scared of broccoli!'
  '3lKYeWEVfOQi[LcDgcMPYXOd'
  ```

- Wenn es drei Argumente gibt und das erste Argument `"decrypt"` ist, dann soll der Schlüssel aus dem zweiten Argument verwendet werden um die verschlüsselte Nachricht aus dem dritten Argument zu entschlüsseln und die entschlüsselte Nachricht in single quotes ausgegeben werden:

  ```
  $ ./symcrypt decrypt 'ilikecats' '3lKYeWEVfOQi[LcDgcMPYXOd'
  'I am scared of broccoli!'
  ```

Sie können annehmen, dass die Nachricht und der Schlüssel maximal `4999` Zeichen lang sind und deshalb den `char`-Arrays, in denen Sie die verschlüsselte bzw. entschlüsselte Nachricht ablegen, einfach fest die Größe `5000` geben. Wie man Arrays von dynamischer Größe anlegt wird in der nächsten Vorlesung gezeigt.

Das Ergebnis soll in single quotes ausgegeben werden, da sowohl verschlüsselte als

auch entschlüsselte Nachrichten Leerzeichen enthalten können, die man sonst leicht übersehen kann.

Aufgabe 3.5 (Erfahrungen; 2 Punkte)

Notieren Sie Ihre Erfahrungen mit diesem Übungsblatt in der Datei `erfahrungen.txt` (benötigter Zeitaufwand, Probleme, Bezug zur Vorlesung, Interessantes, etc.).

Der Zeitaufwand *muss* dabei in der ersten Zeile und in exakt dem folgenden Format notiert werden, da wir sonst nicht automatisiert eine Statistik erheben können:

```
Zeitaufwand: 3:30

<...Andere Erfahrungen...>
```

Die Angabe `3:30` steht hier für 3 Stunden und 30 Minuten.