

F11 - tecken och strängar

Programmeringsteknik med C och Matlab, 7,5 hp

Niclas Börlin
niclas.borlin@cs.umu.se

Datavetenskap, Umeå universitet

2023-10-16 Mån

- Förstå relationen mellan **tecken** (**char**), fält av **char** och **strängar**

Tecken i C

Tecken (char) i C

- En teckenkonstant skrivs i C med apostrofer runt om

```
char beg    = 'A'; // 65
char end    = 'Z'; // 90
char start  = 'a'; // 97
char stop   = 'z'; // 122
char zero   = '0'; // 48
char nine   = '9'; // 57
char space  = ' '; // 32
char hash   = '#'; // 35
char dollar = '$'; // 36
```

- Kompilatorn översätter teckenkonstanterna till en **char** (vanligen en byte) med ASCII-värdet för respektive tecken

Styrkoder för tecken

- För en del "osynliga" tecken finns styrkoder (*escape sequences*)

```
char newline      = '\n'; // 10
char carriage_return = '\r'; // 13
char null         = '\0'; // 0
```

- OBS! Ovanstående teckensekvenser översätts av kompilatorn till **en char**, inte två!
 - '\n' översätts till en **char** som har värdet 10
- Några synliga tecken behöver också anges med styrkoder

```
char backslash    = '\\'; // 92
char single_quote = '\''; // 39
char double_quote = '\"'; // 34
```

- Det går också att initiera med heltal

```
char newline = 10;
```

Strängar i C

Strängar i C (1)

- C har ingen egen datatyp för strängar
- I stället representeras strängar som **fält av char**
- En sträng i C är **noll-terminerad**, dvs. den **avslutas** (termineras) med tecknet *null* (som har värdet 0)

Strängar i C (2)

- En strängkonstant

```
"a0bc"
```

i källkoden översätts av kompilatorn till en **sekvens** av tecken som också rymmer **termineringen**

- Kompilatorn reserverar alltså **fem** tecken för ovanstående sträng:

a	0	b	c	\0
---	---	---	---	----

- De heltalsvärden som faktiskt lagras är

97	48	98	99	0
----	----	----	----	---

- Notera skillnaden mellan **tecknet** '0' (ASCII 48) och värdet 0!

- Den **tomma** strängen

```
""
```

översätts alltså till **ett** tecken

\0

eller

0

Styrkoder i strängar

- ▶ Styrkoder översätts på samma sätt som för **char**
- ▶ Strängen

```
"Hello!\n"
```

kommer att översättas till

H	e	l	l	o	!	\n	\0
---	---	---	---	---	---	----	----

- ▶ De heltalsvärden som faktiskt lagras är

72	101	108	108	111	33	10	0
----	-----	-----	-----	-----	----	----	---

Deklaration av sträng som fält av char

- ▶ Vi kan deklarera en variabel som ett fält av **char**

```
char s[100];
```

- ▶ Variabeln *s* har 100 platser för element av typen **char**
- ▶ Ofta kallas en sådan variabel för en **buffert**

Initiering av strängvariabler som fält

- ▶ Vi kan initiera strängar som fält av **char**

```
char s[] = {'a', 98, 'c', '\0'};
```

eller som en "sträng"

```
char s[] = "abc";
```

- ▶ I det senare fallet reserverar compilatorn **automatiskt** utrymme för det avslutande \0-tecknet

s	a	b	c	\0
---	---	---	---	----

- ▶ Bägge deklarationerna ovan reserverar alltså minst 4 bytes

Längder på strängar (1)

- ▶ Vi har alltså **två** längder att hålla reda på:
 - ▶ En **fix** (maximal) längd som avgörs vid fältets deklaration
 - ▶ En **variabel** längd som bestäms av första \0-tecknet

- ▶ En deklaration utan storlek blir automatiskt rätt

```
char s1[] = "abc";
```

- ▶ Om bufferten är större än vad som behövs så kommer det att fungera...

```
char s3[5] = "abc";
```

... men om bufferten är för liten så kommer inte den avslutande nollan med!

```
char s2[3] = "abc"; // No termination null character!
```

Längder på strängar (2)

Följande kod

```
#include <stdio.h>

int main(void)
{
    char s1[] = "abc"; // Reserves 4 chars
    char s2[3] = "def"; // Reserves 3 chars
    char s3[5] = "xyz"; // Reserves 5 chars

    printf("sizeof(s1)=%ld\n", sizeof(s1));
    printf("s1 = '%c', '%c', '%c'\n", s1[0],
          s1[1], s1[2]);
    printf("s1 = \"%s\"\n", s1);

    printf("\nsizeof(s2)=%ld\n", sizeof(s2));
    printf("s2 = '%c', '%c', '%c'\n", s2[0],
          s2[1], s2[2]);
    printf("s2 = \"%s\"\n", s2); // DANGEROUS!

    printf("\nsizeof(s3)=%ld\n", sizeof(s3));
    printf("s3 = '%c', '%c', '%c'\n", s3[0],
          s3[1], s3[2]);
    printf("s3 = \"%s\"\n", s3);

    return 0;
}
```

gav följande utskrift för mig:

```
sizeof(s1)=4
s1 = 'a', 'b', 'c'
s1 = "abc"

sizeof(s2)=3
s2 = 'd', 'e', 'f'
s2 = "defabc"

sizeof(s3)=5
s3 = 'x', 'y', 'z'
s3 = "xyz"
```

Notera den andra utskriften av s2!

Pekare och strängar (1)

Vi kan också använda **char** *-variabler för att hantera strängar

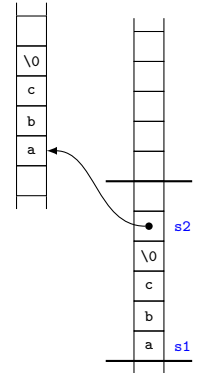
Deklarationerna

```
char s1[] = "abc";
char *s2 = "abc";
```

reserverar samma antal tecken (4) för strängen "abc"

I det andra fallet reserveras också plats för pekarvariabeln s2

- Pekarvariabeln initieras till adressen i minnet där kompilatorn lagrat strängen "abc"
- Denna plats är vanligtvis inte på stacken



Pekare och strängar (2)

Ska ni bara **läsa av** strängen spelar det ingen roll vilken ni väljer

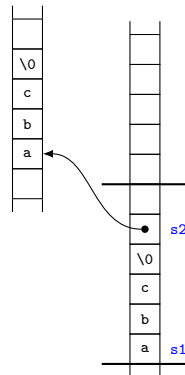
```
#include <stdio.h>

int main(void)
{
    char s1[] = "abc";
    char *s2 = "abc";

    printf("First char of s1 = '%c'\n", s1[0]);
    printf("First char of s2 = '%c'\n", s2[0]);
    printf("The whole s1 string = '%s'\n", s1);
    printf("The whole s2 string = '%s'\n", s2);

    return 0;
}
```

```
First char of s1 = 'a'
First char of s2 = 'a'
The whole s1 string = 'abc'
The whole s2 string = 'abc'
```



Pekare och strängar (3)

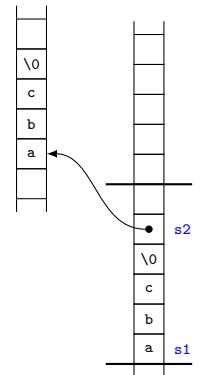
Vill ni däremot **ändra** i strängen så måste ni välja den första varianten

```
#include <stdio.h>

int main(void)
{
    char s1[] = "abc";
    char *s2 = "abc";

    s1[1] = 'x'; // Will change s1 to become "axc"
    s2[1] = 'x'; // Undefined, may crash

    return 0;
}
```



String copy — strcpy (1)

- Språket C saknar stöd för att **kopiera** strängar
 - Det går därför inte att skriva följande:

```
char buf[8];  
buf = "abc";
```

Kopiering av strängar

- Däremot finns det flera **biblioteksfunktioner** som stöd
- Funktionen strcpy med deklaration

```
strcpy(char *dest, const char *src);
```

kopierar strängen som börjar i `src` till bufferten som börjar i `dest`

- Kopieringen sker **inklusive** den avslutande nollan
- Det görs **ingen** kontroll av om bufferten är **tillräckligt stor**

Hjälp för funktioner

- Kommandot

```
man 3 strcpy
```

i linux ger

```
NAME
    strcpy, strncpy - copy a string
SYNOPSIS
    #include <string.h>
    char *strcpy(char *dest, const char *src);
    char *strncpy(char *dest, const char *src, size_t n);
DESCRIPTION
    The strcpy() function copies the string pointed to by src, including
    the terminating null byte ('\0'), to the buffer pointed to by dest.
    The strings may not overlap, and the destination string dest must be
    large enough to receive the copy. Beware of buffer overruns! (See
    BUGS.)

    The strncpy() function is similar, except that at most n bytes of src
    are copied. Warning: If there is no null byte among the first n bytes
    of src, the string placed in dest will not be null-terminated.

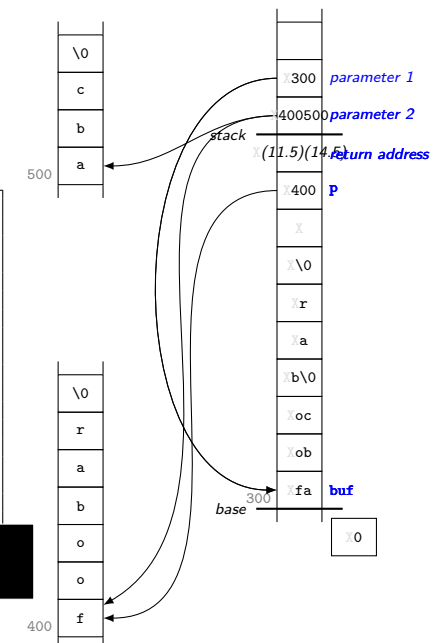
    If the length of src is less than n, strncpy() writes additional null
    bytes to dest to ensure that a total of n bytes are written.
RETURN VALUE
    The strcpy() and strncpy() functions return a pointer to the destina-
    tion string dest.
...
```

String copy — strcpy (2)

- Vi kör koden

```
code/string-copy.c
1  #include <stdio.h>
2  #include <string.h>
3
4  #define BUFSIZE 8
5
6  int main(void)
7  {
8      char buf[BUFSIZE];
9      char *p = "foobar";
10
11     strcpy(buf, p);
12     printf("buf = \"%s\"\n", buf);
13
14     strcpy(buf, "abc");
15     printf("buf = \"%s\"\n", buf);
16
17     return 0;
18 }
```

```
buf = "foobar"
buf = "abc"
```



String copy — strcpy (3)

- ▶ Funktionen strcpy vet inte **hur stort** dest-fältet är
 - ▶ Funktionen fortsätter att kopiera tecken tills den kopierat **termineringen** `\0`

```
char buf[3];
strcpy(buf, "foobar");
printf("%s\n", buf);
```

- ▶ Vad kommer printf att skriva ut?
- ▶ Vad kan gå fel?

String copy with limit — strncpy (1)

- ▶ Om vi vill begränsa hur många tecken som maximalt kopieras kan vi använda oss av funktionen strncpy

```
char *strncpy(char *dest, const char *src, size_t n);
```

- ▶ Typen `size_t` är en **unsigned** heltalsdatatyp avsedd att representera storlekar
- ▶ Ett anrop till strncpy kopierar `n` tecken
 - ▶ Är `src` kortare än `n` så fylls resterande (upp till `n`) tecken med `\0`
 - ▶ **WARNING**: om `src` inte innehåller `\0` bland de första `n` tecknen så kommer dest **ej bli terminerad** med `\0`

String copy with limit — strncpy (2)

- ▶ Vilket problem kan vi få i denna kod?
- ▶ Hur kan vi åtgärda problemet?
- ▶ Code 1:

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

#define BUFLen 10

int main(void)
{
    char buf[BUFLen];

    strncpy(buf, "Hello, World!\n",
            BUFLen);

    printf("%s\n", buf);

    return 0;
}
```

- ▶ Code 2:

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

#define BUFLen 10

int main(void)
{
    char buf[BUFLen];

    strncpy(buf, "Hello, World!\n",
            BUFLen);

    // Add explicit termination
    buf[BUFLen-1] = '\0';

    printf("%s\n", buf);

    return 0;
}
```

String length — strlen

- ▶ Om vi vill veta **längden** på en sträng finns funktionen strlen

```
size_t strlen(const char *s);
```

- ▶ Termineringen räknas **inte** in i strängens längd

```
printf("%d", strlen("hello"))
```

skriver ut 5

String concatenation — strcat (1)

- ▶ Att foga samman två strängar kallas att *konkatenera*
- ▶ Det kan göras med hjälp av funktionen `strcat`

```
NAME
    strcat, strncat - concatenate two strings
SYNOPSIS
    #include <string.h>
    char *strcat(char *dest, const char *src);
    char *strncat(char *dest, const char *src, size_t n);
DESCRIPTION
    The strcat() function appends the src string to the dest string, over-
    writing the terminating null byte ('\0') at the end of dest, and then
    adds a terminating null byte. The strings may not overlap, and the
    dest string must have enough space for the result. If dest is not
    large enough, program behavior is unpredictable; buffer overruns are a
    favorite avenue for attacking secure programs.

    The strncat() function is similar, except that
    * it will use at most n bytes from src; and
    * src does not need to be null-terminated if it contains n or more
      bytes.
    As with strcat(), the resulting string in dest is always null-termi-
    nated.
    If src contains n or more bytes, strncat() writes n+1 bytes to dest (n
    from src plus the terminating null byte). Therefore, the size of dest
    must be at least strlen(dest)+n+1.
...
RETURN VALUE
    The strcat() and strncat() functions return a pointer to the resulting
    string dest.
...
```

String concatenation — strcat (2)

- ▶ Funktionen `strcat`

```
char *strcat(char *dest, const char *src);
```

kopierar strängen `src` och lägger den i slutet på `dest`

- ▶ Exempel

```
char dest[16] = "Hello ";
char src[16] = "World!\n";
strcat(dest, src);
```

dest	H	e	l	l	o	␣	\0	o	r	l	d	!	\n	\0		
src	W	o	r	l	d	!	\n	\0								

String concatenation with limit — strncat

- ▶ Precis som `strcpy` kopierar `strcat` tills den hittar termineringen
 - ▶ Inget skydd för `dest`
- ▶ Vill vi hindra `dest` från att överfyllas måste vi använda `strncat`

```
char *strncat(char *dest, const char *src, size_t n);
```

- ▶ Funktionen `strncat` kopierar som mest `n` tecken från `src` och lägger därefter till `'\0'`
- ▶ Exempel

```
char dest[16] = "Hello ";
char src[16] = "World!\n";
strncat(dest, src, 5);
```

dest	H	e	l	l	o	␣	\0	o	r	l	d	\0				
src	W	o	r	l	d	!	\n	\0								

Jämförelser av strängar

Jämföra tecken i strängar

- ▶ När man arbetar med strängar är det vanligt att man vill jämföra deras **innehåll**
- ▶ Men följande kod

```
char string1[15] = "Hello";
char string2[15] = "Hello";
if (string1 == string2) {
    printf("Equal\n");
}
```

kommer inte att skriva ut något, eftersom villkoret i if-satsen jämför **minnesadresserna** för de båda strängarnas första element, inte strängarnas innehåll

Compare strings – strcmp och strncmp

- ▶ I stället kan vi använda oss av funktionerna strcmp och strncmp:

```
int strcmp(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
```

- ▶ Funktionerna strcmp och strncmp returnerar
 - ▶ ett **negativt** tal om s1 kommer **före** s2
 - ▶ ett **positivt** tal om s1 kommer **efter** s2
 - ▶ **noll** om de två strängarna är **lika**
- ▶ Tecknens **numeriska** värden jämförs
 - ▶ Tänk på att 'S' < 's' och 'Z' < 'a'

strncmp, exempel

```
char string1[15] = "Hello";
char string2[16] = "Hello";
if (strncmp(string1, string2, 5) == 0) {
    printf("Equal\n");
}
```

Inläsning av strängar

Get strings — fgets och gets

- ▶ Vill vi **läsa in** strängar kan vi använda funktionen fgets

```
char *fgets(char *str, int max_len, FILE *filep);
```

- ▶ Det finns också en funktion som heter gets

```
char *gets(char *str);
```

- ▶ Använd **INTE** gets!

- ▶ gets läser från stdin (normalt tangentbordet) tills den påträffar '\n' eller EOF (*end-of-file*)
- ▶ Risk för buffer overflow

- ▶ fgets är säkrare!

- ▶ Läser maximalt max_len-1 tecken
- ▶ Avslutar alltid med '\0', sparar '\n' bara om det ryms

In-/utmatning av ett tecken i taget — getchar och putchar

- ▶ Funktionen

```
int getchar(void);
```

läser **ett** tecken från stdin (normalt tangentbordet)

- ▶ Funktionen

```
int putchar(int c);
```

skriver ut tecknet c på stdout (normalt skärmen)

Från sträng till tal

- ▶ Funktionen scanf har vi använt många gånger
 - ▶ Vi kan läsa in till en sträng – '%s'
 - ▶ Vi kan läsa in till olika typer av tal – '%d', '%lf'

```
scanf("%d", &n);
```

- ▶ Funktionen sscanf fungerar som scanf men läser från en **buffert** istället för stdin

```
sscanf(buf, "%lf", &x);
```

- ▶ Används gärna i kombination med fgets:

```
char buf[BUFSIZE];  
double x;  
fgets(buf, BUFSIZE-1, stdin);  
sscanf(buf, "%lf", &x);
```

- ▶ Fördelen är att om något går fel i sscanf vid tolkningen av buf så ligger inte "skräpet" kvar vid nästa anrop till fgets

Från tal till sträng

- ▶ Funktionen printf har vi också använt många gånger

- ▶ Vi kan skriva ut olika värden
- ▶ Vi kan skriva ut en sträng – '%s'

```
printf("%s %s\n", "Hello", name);
```

- ▶ Funktionen sprintf fungerar som printf men skriver till en **buffert** istället för till stdout

```
sprintf(str, "The answer is %f", x);
```

Tolka tecken

- ▶ Ska vi tolka indata kan det vara bra att kunna **klassificera** enskilda tecken
 - ▶ Header-filen `ctype.h` innehåller funktioner för att kolla på tecken
 - ▶ Följande funktioner returnerar 0 vid falskt
 - ▶ `isalpha(c)` kollar om `c` är en bokstav
 - ▶ `isdigit(c)` kollar om `c` det är en siffra
 - ▶ `islower(c)` kollar om `c` är en liten bokstav
 - ▶ `isupper(c)` kollar om `c` är en stor bokstav
 - ▶ `isspace(c)` kollar om `c` är ett blanksteg (eller tab, newline, m.fl.)
 - ▶ `ispunct(c)` kollar om `c` är ett tecken som inte är ett kontrolltecken, ett blanksteg, en bokstav eller en siffra
- ▶ OBS! Ovanstående är garanterat att fungera för **engelska** tecken
- ▶ För övriga teckenuppsättningar finns idag "internationella" (*locale-aware*) varianter

Manipulera tecken

- ▶ Header-filen `ctype.h` innehåller även funktioner för att **manipulera** (engelska) tecken
 - ▶ `tolower(c)` ändrar `c` till liten bokstav
 - ▶ `toupper(c)` ändrar `c` till stor bokstav
- ▶ Exempel:

```
#include <stdio.h>
#include <stdbool.h>
#include <ctype.h>
int main(void)
{
    char c;
    do {
        printf("Do you want to continue? (Y/N)");
        c = tolower(getchar());
        if (c != 'y' && c != 'n') {
            printf("Please press 'y' or 'n'!");
        }
    } while (c != 'y' && c != 'n');
    return 0;
}
```

- ▶ Internationella varianter finns också

Många funktioner blir det...

- ▶ I dag har vi sett många funktioner
- ▶ Hur ska man ha koll på vilka som finns?
- ▶ Hur ska man ha koll på hur de ska användas?
- ▶ Hur ska man ha koll på vilken biblioteksfil som måste inkluderas?
- ▶ Kursbok, eller annan bok
- ▶ Webben
 - ▶ <http://en.cppreference.com/w/c>
 - ▶ Wikipedia (den engelska)
 - ▶ Sök på webben efter andra sidor
 - ▶ "man 3 funktionsnamn" (eller bara "man funktionsnamn") i ett terminalfönster
 - ▶ Välj den du gillar

Vanliga fel — strängar

- ▶ Kom ihåg att strängar är **fäلت av char**
- ▶ Kom ihåg att alla strängar ska **sluta med** (termineras av) `'\0'`
- ▶ Strängar som skapats i en lokal variabel i en funktion kan **inte returneras** (hur? kommer på senare kurser)
- ▶ Adress-operatorn `&` ska inte användas vid inläsning till strängar med hjälp av `scanf`
- ▶ **Se upp för buffer overflow!**
 - ▶ Använd de funktioner där man kan ange buffertlängder!
- ▶ Det går **inte** att jämföra och tilldela strängar mha `==` och `=` som vanliga enkla datatyper

- ▶ Skriv ett program som läser in en sträng, översätter den till rövarspråket samt skriver ut det översatta
- ▶ Rövarspråket skapas genom att alla konsonanter dubblas med ett 'o' mellan
- ▶ Vokaler och andra tecken förblir oförändrade

- ▶ Inlämning
- ▶ Uppgiften
- ▶ Strukturerad problemlösning