Screencast

Strängar

Översikt

– I C är strängar implementerade som arrayer av tecken, vars sista tecken är `'\0'`, det så-kallade ”nulltecknet” som anger att strängen är slut. Man säger att C:s strängar är ”nullterminerade”.

– Det normala sättet att deklarera en ”strängvariabel” är att deklarera en pekare till en **`char`**. Minns att pekare och arrayer i stort sätt är utbytbara i C, så detta betyder inte att strängens längd är *ett* tecken, utan att pekaren pekar på det första tecknet.

– Strängliteraler skrivs `"så här!"`, d.v.s. inom citationstecken eller dubbla anföringstecken, och får automatiskt ett nulltecken instoppat i slutet. `"`-tecknen ingår förstås inte som en del av strängen.

Nedanstående kod deklarerar tre strängvariabler.

```
char *person1 = "Dawkins";
char *person2 = "Hitchens";
char person3[] = "Bingo Rimer";
```

Notera skillnaden mellan Bingo och de två andra. Bingos variabel är deklarerad som en array medan de andra är pekare. Resultatet av denna kod är att Bingo hamnar på stacken medan Dawkins och Hitchens hamnar på heapen. Vi lämnar som en övning att fundera ut vad detta betyder tills vidare.

Fråga: om ett tecken ryms i en byte, hur många bytes tar då strängen `"1234567890"`?

– **SVAR:** 11 – de 10 tecknen `'1'` till `'0'` samt `'\0'`, nulltecknet.

– En teckenliteral skrivs `'A'`, `' '` etc. Notera skillnaden mellan `"A"` (en sträng) och `'A'` (ett tecken). Varför kan man inte tilldela mellan strängar och tecken?

– Nulltecknet är ett specialtecken; ytterligare specialtecken kan får med hjälp av ”escaping”, t.ex. `\n` (newline) och `\t` (gå till nästa tabulatorstopp). Prova t.ex.:

```
1     puts("----------------------------");
2     printf("Hej");
3     printf("Hej\t");
4     printf("Hej\n");
5     printf("King\0Kong");
6     puts("----------------------------");
```

– En stor mängd funktioner för att manipulera strängar finns i C-biblioteket `string.h` som inkluderas som vanligt:

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

Bekanta dig med detta bibliotek på egen hand! Exempel på ofta använda funktioner är t.ex. `strlen` som räknar ut längden på en sträng (skriv ett litet program som verifierar att mitt svar på frågan ovan är korrekt), `strcpy` som kopierar innehållet i en sträng till en annan, etc.

– Strängars innehåll kopieras inte vid tilldelning. Det enda som händer vid tilldelningen nedan är att `s1` och `s2` sätts att peka på samma sträng.

```
char *s1 = "Foo, bar";
char *s2 = s1;
```

Efter detta är `s1` och `s2` alias och alla förändringar av `s2` blir synliga i `s1` och tvärtom.

– För att kopiera en sträng till en annan använder vi istället funktionen `strcpy`:

```
char *s1 = "Foo, bar";
char s2[strlen(s1)+1];
strcpy(s2, s1);
```

Notera att vi måste se till att det finns plats för de kopierade tecknen (glöm inte nulltecknet!) i målsträngen. Det är bättre i regel att använda `malloc`:

```
char *s1 = "Foo, bar";
char *s2 = malloc(strlen(s1)+1);
strcpy(s2, s1);
...
free(s2);
```

Notera att vi också måste lämna tillbaka minnet som krävdes av den nya strängen med hjälp av `free`.

– Jämförelser mellan strängar görs med funktionen `strcmp` som finns i `string.h`. Ponera följande kod (finns bland utdelat material):

```
 1 char *p1 = "Captain Beefheart";
 2 char *p2 = p1;
 3 char p3[] = "Captain Beefheart";
 4
 5 printf("p1 == p2?\t\t%s\n", p1 == p2 ? "yes" : "no");
 6 printf("p1 == p3?\t\t%s\n", p1 == p3 ? "yes" : "no");
 7 printf("p2 == p3?\t\t%s\n", p2 == p3 ? "yes" : "no");
 8 printf("strcmp(p1, p2)?\t%s\n", strcmp(p1, p2) ? "no" : "yes");
 9 printf("strcmp(p1, p3)?\t%s\n", strcmp(p1, p3) ? "no" : "yes");
10 printf("strcmp(p2, p3)?\t%s\n", strcmp(p2, p3) ? "no" : "yes");
```

Jämförelserna på rad 5–7 kontrollerar om *adresserna* i variablerna `p1`, `p2` och `p3` är samma, d.v.s. om de pekar till samma plats i minnet.

Jämförelserna på rad 8–10 jämför istället *innehållet i strängarna*. Strcmp returnerar 0 om strängarna är lika.

Fråga: Vad skrivs ut när man kör programmet?

1. yes, no, no, yes, yes, yes
2. yes, yes, yes, yes, yes, yes
3. yes, no, yes, yes, yes, yes

– **SVAR**: Alternativ (1). Alla jämförelser av strängens innehåll blir naturligtvis sanna (det är ju samma innehåll i båda strängarna). Variablernas adresser är samma i `p1` och `p2`, men inte i `p3` som innehåller en egen kopia av texten.

  Notera att tilldelningen `p2 = p1` på rad 2 *inte kopierar strängen* utan endast sätter pekaren `p2` att peka på samma minnesadress (och därmed samma sträng) som `p1`.

  Pröva att byta ut `char p3[] = "Captain Beefheart";` mot `char *p3 = "Captain Beefheart";` och köra programmet igen! Resultatet blir nog förvånande, och beror på att C-kompilatorn optimerar programmet och bara inkluderar en enda kopia av `"Captain Beefheart"`! Detta torde illustrera att man måste vara aktsam när man jämför strängar med `==` i C!

– Funktionen `strcmp` undersöker egentligen relationen mellan två strängar i lexikografisk ordning. Ponera ett anrop `strcmp(s1,s2)`. Dessa är de möjliga returvärdena:

< 0	`s1` kommer *före* `s2` i lexikografisk ordning
0	`s1` och `s2` är lika
> 0	`s1` kommer *efter* `s2` i lexikografisk ordning

Sammanfattning

- Strängars *innehåll* koperias **ej** med tilldelningsoperatorn `=` och jämföras **ej** med jämförelseoperatorerna `==`, `<`, `>` etc.

  Istället används funktionerna `strcpy` och `strcmp`.

- Minne för en sträng måste explicit allokeras, antingen
  - Via en literal som skapar en konstant, t.ex. `"Foo"`
  - På stacken, t.ex. `char s[128]`
  - På heapen, t.ex. `char *s = malloc(128);`

- Strängar är nullterminerade

– Tänkbara implementationer av nämnda funktioner:

```
1   int strlen(char *str) {
2     int length = 0;
3     while (str[length]) ++length;
4     return length;
5   }
6
7   char *strcpy(char *dst, char *src) {
8     char *cursor = dst;
9     do {
10      *cursor++ = *src;
11    } while (*src++);
12    return dst;
13  }
14
15  int strcmp(char *s1, char *s2) {
16    for (; *s1 && *s2 && *s1 == *s2; ++s1,++s2) ;
17    return *s1 - *s2;
18  }
```