Řetězce

tl;dr

Čtení a zápis textu v C:

```
char data[16];
printf("zadej text: "); scanf("%s", data);
printf("zadal jsi: %s", data);
```

Tento příklad vám stačí pro základní práci s řetězci, bez dalších podrobností se ale setkáte s celou řadou překvapení vedoucích k nečekaným chybám, které jsou co možná nejstručněji zmíněny níže. Je to rozvedeno na celých 5 stránek, pro které nepoužívejte rychločtení, ale opravdu si uvědomte obsah každé věty. Není to jen o práci s textem, popisuje se zde i základní práce s pamětí, která vám umožní proniknout do nízkoúrovňového programování. Ke zdůvodnění, proč se tím vůbec prokousávat, když vám stačí ty tři řádky zmíněné výše, viz poslední 2 stránky v kapitole Použití, kde je vysvětlena hranice mezi programy, které je lepší psát v javascriptu a mezi říší jazyka C.

Implementace řetězců

V předchozím díle jsme se setkali s řetězcovými literály (např. `"text"`). Ty jsou uloženy ve statické paměti, tj. jsou vytvořeny na začátku programu a uvolněny až na konci programu. Jsou také konstantní, tj. nelze je měnit. Tyto řetězce (*strings*) můžeme uložit do paměti typu `const char*`. Hvězdička na konci znamená ukazatel, což je typ proměnné, který obsahuje adresu. K hodnotě takové proměnné přistoupíme operátorem `*` (nezaměňujte tyto dva rozdílné významy). Shrňme si, jak je to s adresami a hodnotami hodnotových proměnných a ukazatelů:

	hodnota	adresa
`int x`	`x`	`&x`
`const char* y`	`*y`	`y`

Často potřebujeme v řetězci přepsat nějaké znaky. Nemůžeme ale zapsat např.

```
char* str = "login"; // varování, nekonstantní str ukazuje na konstantu
```

protože řetězcový literál je konstantní a proměnná `str` zde značí nekonstantní ukazatel. Některé architektury (např. Arduino) ukládají konstantní paměť do jiného segmentu než nekonstantní (např. do ROM). Pokus o změnu této paměti skončí běhovou chybou, proto většina kompilátorů při kompilaci vygeneruje varování.

Měnitelné řetězce lze uložit jako pole znaků. Lze tedy zapsat

```
char str[] = "login"; // v pořádku
```

Pokud proměnnou definujeme při deklaraci, není nutné psát do hranatých závorek velikost pole, určí se automaticky z pravé strany. **V tomto případě se nevytváří řetězcový literál**, je to jen zkrácený zápis následujícího

```
char str[] = {'l', 'o', 'g', 'i', 'n', '\0'};
```

Výraz ve složených závorkách se nazývá inicializátor, podrobněji ho probereme u struktur. Všechny řetězce v C jsou ukončeny znakem `'\0'`, což je znak NULL s ASCII indexem 0 (lze ho tedy zapsat i jako `0`. Nezaměňujte se znakem `'0'` (číslo 0), který má ASCII index 48. Díky tomu můžeme řetězec znak po znaku procházet bez jakýchkoliv dodatečných informací (např. bez znalosti jeho délky):

```
for(char* i=str; *i!=0; i++) printf("%c", *i);
```

Na začátku cyklu vytvoříme nekonstantní ukazatel `i` (typu `char*`), jehož hodnotu (`*i`) tiskneme. Poté zvětšíme adresu o 1 (**pole se vždy vytváří na souvislém bloku paměti**) a tento postup opakujeme tak dlouho, dokud je hodnota `i` různá od `0`. Tímto způsobem bychom mohli zjistit délku řetězce, případně též voláním funkce `strlen(řetězec)`.

Je možné také psát

```
for(const char* i=str; *i!=0; i++) printf("%c", *i);
```

Zde je označena jako konstantní paměť, na kterou ukazatel ukazuje, ukazatel je nekonstantní, proto ho můžeme měnit příkazem `i++`. Návod, jak se v této konstantnosti vyznat, se skládá ze dvou kroků:

1. pokud je `const` na začátku, dáme ho na druhé místo (`const char*` je tedy totéž, co `char const*`), je tomu tak proto, že jiné jazyky vyžadují `const` jako první.
2. výraz čteme zprava doleva, tedy
    - `char const*` je ukazatel (`*`) na konstantní (`const`) paměť typu char
    - `char* const` je konstantní ukazatel na nekonstantní paměť char
    - `char const* const` je konstantní ukazatel na konstantní paměť char

Je jedno, z jaké strany hvězdičky dáme mezeru, následující výrazy jsou ekvivalentní:

- `char* x`
- `char *x`
- `char*x`
- `char * x`

`char* x` zdůrazňuje, že proměnná `x` je typu ukazatel na char. `char *x` zdůrazňuje, že ukazatel `x` ukazuje na paměť typu char. Pozor pouze na následující:

- `char* x,y;` je totéž, co `char* x; char y;`
- `char *x,*y` je totéž, co `char* x; char* y;`

Následující ukázky kódu upozorňují na rozdíl mezi řetězcovým literálem a polem znaků:

ukázka 1

```
const char* getHello() {
  const char* result = "hello";
  // result[0] = 'H'; // chyba, result ukazuje na konstantní paměť
  return result; // v pořádku, result ukazuje na statickou paměť
}
const char* x = getHello();
puts(x); // v pořádku
```

ukázka 2

```
char* getHello() {
  char result[] = "hello";
  result[0] = 'H'; // v pořádku
  return result; // vracíme odkaz na paměť, která bude ihned uvolněna!
}
const char* x = getHello(); // x ukazuje na uvolněnou paměť
puts(x); // může fungovat, mohou se vypsat divné znaky, může spadnout
```

ukázka 3

```
char* getHello() {
  static char result[] = "hello";
  result[0] = 'H';
  return result;
}
const char* x = getHello(); // v pořádku, getHello vrací statickou paměť
puts(x); // Hello
char* y = getHello();
y[0] = 'B';
puts(x); // Bello (x a y ukazují na tutéž paměť)
```

Všimněte si v ukázce č. 3, že přestože jsme `x` označili jako ukazatel na konstantní paměť, tato paměť byla změněna, protože `y` je ukazatel na nekonstantní paměť, která ukazuje na totéž místo.

Načítání řetězců

Funkci `scanf` (*scan* formatted) lze použít i k načítání řetězců pomocí symbolu `%s`, vyzkoušejte si:

```
int x;
char str[16];
scanf("%d %s", &x, str); // zadejte např. 15 jablek
while(getchar()!='\n');  // vyprázdnění vstupního bufferu
printf("%s %d", str, x); // vypíše jablek 15
```

`str` je referenční proměnná (odkaz), obsahuje adresu, a proto při načítání před ní neuvádíme `&`. Uvedený příklad má následující skryté chyby:

- Pokud uživatel zadá víc než 15 znaků, zapíše se výsledek do nealokované paměti. To lze ošetřit číslem za `%`: `scanf("%15s", str)` - načte max 15 znaků (poslední se doplní ukončovací 0).
- `%s` čte znaky až do první mezery. To lze ošetřit výčtovým typem v hranatých závorkách, např. `scanf("%[ab ]", str)` - načítá do pouze znaky a, b a mezeru. Lépe

lze použít negativní výčet specifikovaný prvním znakem stříšky (*circumflex*) `^`:
`scanf("%15[^\n]", str)` - načte max. 15 znaků včetně mezer (čte všechny znaky krom konce řádky (který se při načítání z konzole stejně ve vstupu neobjeví).

Kopírování řetězců

Při použití operátoru `=` na řetězce se kopíruje pouze odkaz, ne jejich obsah - tzv. mělká kopie (*shallow copy*). Pro hluboké kopírování (*deep copy*) je nutno do nového místa zkopírovat vše znak po znaku. To lze také funkcí `strcpy` z knihovny `string.h`:

```
const char* src = "hello";
char* dest1 = src; // shallow copy
// dest1[0] = 'H'; // chyba: dest ukazuje na src a ta je konstantní

// char dest2[6] = src; // chyba: takto pole inicializovat nelze
char dest2[6];
for(char* i=src, *j=dest2; *i!=0; ++i, ++j) *j = *i; // deep copy
*j = 0; // nezapomeňme na ukončovací nulu

for(char* i=src, *j=dest2; *i!=0 ;) *j++ = *i++; // deep copy ;)

#include <string.h>
strcpy(dest2, src); // také deep copy, srozumitelný zápis
```

Srovnejte kopírování do `dest1` s ukázkou č. 3 výše. Zde je `src` vytvořena jako konstantní, v ukázce jsme ji vytvořili jako nekonstantní (konstantní byl pouze ukazatel).

Porovnávání řetězců

Použitím operátoru `==` se porovnává adresa, ne hodnota řetězců. Pro porovnávání hodnoty je třeba porovnat znak po znaku nebo použít funkci `strcmp` (*string compare*), která vrací nulu, pokud se řetězce rovnají, pozor na řetězcové literály.

ukázka 1

```
char str1[] = "hello";
char str2[] = "hello";
printf("%d", str1==str2); // vždy 0
```

`str1` a `str2` se nacházejí na různém paměťovém místě

ukázka 2

```
char str1[] = "hello";
char str2[] = "hello";
printf("%d", strcmp(str1,str2)==0 ); // 1
```

Zde je hluboké porovnávání, **obsahy** řetězců se rovnají.

ukázka 3

```
const char* str1 = "hello";
const char* str2 = "hello";
printf("%d", str1==str2); // může být 1!
```

V tomto případě kompilátor může (nemusí!) vytvořit konstantu "hello" na jednom paměťovém místě a `str1` i `str2` na ně odkázat.

Prohledávání řetězců

Pro nalezení znaku v řetězci můžeme použít funkci `strchr(řetězec,znak)` (*string search character*), která vrací ukazatel na nalezený znak. Všechny pozice znaku ve slově najdeme:

```
const char* str = "yabadabadoo";
for(const char* i=strchr(str,'a'); i!=0; i=strchr(i+1,'a')) {
  printf("podslovo %s na pozici %d\n", i, i-str);
}

podslovo abadabadoo na pozici 1
podslovo adabadoo na pozici 3
podslovo abadoo na pozici 5
podslovo adoo na pozici 7
```

Obdobně lze použít funkci `strrchr` (*string search reverse character*) pro prohledávání od konce a funkci `strstr` pro hledání podřetězců v řetězci.

Spojování a rozdělování řetězců

Potřebujeme-li řetězec zkrátit, stačí na příslušnou pozici vložit ASCII 0:

```
char str[] = "hello world"; // 12 bytů v paměti
str[5] = 0;
puts(str); // hello
```

Za koncem řetězce máme ještě vyhrazeno 5 bytů paměti (plus jeden navíc na ukončovací nulu), můžeme tedy připojit další část (pomocí cyklu nebo funkcí `strcat` *string concatenate*):

```
strcat(str, " baby");
puts(str); // hello baby
```

Při navazování uživatelského vstupu je třeba mít na paměti, že jeho délka nemusí být známá. Lze pak využít funkci `strncat`, která kopíruje pouze n znaků a nepřipojuje ukončovací nulu:

```
char str[12] = "hello "; // rezervováno dalších 5 znaků
printf("login: ");
char login[16];
scanf("%15s", login); while(getchar()!='\n'); // login dlouhý až 15 znaků
strncat(str, login, 5); // připojíme nejvýše 5 znaků z loginu
str[11] = 0; // standardní ukončení řetězce
puts(str);
```

Obdobně bychom při spojování řetězců neznámé délky měli použít funkci `strncpy` místo `strcpy`. Pro rozdělování řetězců lze použít funkci `strtok(str,znaky)` (*string tokenize*), která najde výskyt některého znaku v řetězci `znaky` a nahradí ho `\0`. Pokud je `str` 0, prohledává se o 1 byte vpravo od posledního nahrazení. Funkce vrací ukazatel na další část řetězce a 0 po skončení řetězce:

```
char str[] = "one two three four";
for(char* i=strtok(str," "); i!=0; i=strtok(0," ")) puts(i);

one
two
three
four
```

Použití

Ve srovnání s vyššími jazyky (tj. takovými, které jsou vzdálenější hardwaru) jako Javascript či PHP je práce s řetězci v C náročnější (musíme myslet na jejich bytovou reprezentaci v paměti). Na druhou stranu program v C je mnohem hospodárnější:

- pokud ve vyšších jazycích spojujete dva řetězce, vyhradí se nová paměť, kam se oba řetězce nakopírují
- pokud dělíme řetězec, vyhradí se nová paměť pro všechny podřetězce

Pokud tedy pracujete na architekturách, kde paměť není problém (např. na PC), ušetříte si práci, použijete-li vyšší jazyk. Ale i zde vám při nevhodném kódu může paměť dojít, např.

```
<script>
var a = "a";
for(var i=0; i<100000; ++i) {
  a+= "a";
  document.write(a); // znemožnění optimalizace na pozadí
}
</script>
```

Tento kód vyhradí 1 byte pro znak "a", poté vyhradí 2 byty pro řetězec "aa", kam řetězce spojí, poté vyhradí 3 byty atd. Celkem tedy potřebujeme řádově 1 + 2 + 3 … + 99999 + 100000 bytů paměti, což je 100000 * 99999 / 2, tedy minimálně 5 GB paměti (další paměť potřebuje prohlížeč - interpret pro svůj běh, zkuste si spustit správce úloh, pak spusťte tento skript v prohlížeči a sledujte, jak paměť roste. Občas se možná její větší blok uvolní díky virtuální paměti a garbage collectoru, o tom více ve čtvrťáku.). Zkuste si také stopnout, jak dlouho vašemu nabušenému procesoru trvá vykonat tento jednoduchý program, a přitom sto tisíc není až tak velké číslo.

Zkuste si podobný program v C:

```
void main() {
  char str[100001] = {}; // vyplní celý blok paměti \0
  for(int i=0; i<100000; ++i) {
    str[i] = 'a';
    puts(str);
  }
}
```

Program také poběží dlouho díky faktu, že příkazová řádka Windows není na tak dlouhý výstup dělaná. Potřebujeme však pouze 100 kB paměti, která za běhu už dále neroste. Aby byla demonstrace přesvědčivější, použijeme pro optimalizaci funkce jádra OS[1]:

```
#include <windows.h>

void main() {
  HANDLE stdout = GetStdHandle(STD_OUTPUT_HANDLE);
  COORD start = {1,1}; // novou hodnotu vypíšeme vždy od začátku konzole
  DWORD out;
```

[1] nemusíte umět - zde jen pro účely demonstrace bez dalších podrobností (existuje řada dalších optimalizací)

```
  char str[100001] = {};
  for(int i=1; i<100000; ++i) {
    str[i] = 'a';
    WriteConsoleOutputCharacter(stdout,str,i,start,&out);
  }
}
```

Na mém stroji trval výpočet 44 sekund za využití jednoho jádra procesoru. Javascript puštěný v Opeře za využití všech 4 jader za stejnou dobu spadl poté, co spotřeboval veškerou dostupnou paměť (cca 7 GB).

Na problém tohoto typu narazíte, pokud budete např. chtít javascriptem vygenerovat nějakou rozsáhlou tabulku. I v javascriptu toto můžete optimalizovat tím, že nepoužijete řetězce, ale předem alokované pole znaků. Zde už ale musíte dohledávat v technických manuálech jak a kde je javascript optimalizován, jelikož jeho interpret je pravděpodobně napsaný v C++ a méně se nadřete, pokud rovnou použijete jazyk C.