1 Opakování pointerů

Charakterový ukazatel `char *p = array` můžeme používat k přístupu na prvky pole pomocí pointerové aritmetiky. `*p` odkazuje na 0. index, zatímco `*(p+1)` odkazuje na 1. index (jiný zápis pro `p[1]`). Například, `char str[5] = "abc"` vytvoří řetězec, který je ukončen znakem `'\0'`.

Více rozměrná pole můžeme deklarovat například jako `float matrix[3][2]`, což vytvoří matici s 3 řádky a 2 sloupci. Pro procházení touto maticí lze použít cykly:

`for (řádky)` a `for (sloupce)`.

Dynamickou alokaci matic lze provést pomocí ukazatelů. Alokujeme paměť pro matici takto:

`float *matrix;` a `matrix = malloc(r*s*sizeof(float));`, kde `sizeof` operátor vrací velikost datového typu.

Další možnost je dvouúrovňová alokace pro dynamickou matici:

`float **matrix;` a `matrix = malloc(r * sizeof(float *));`, kde následně pro každý řádek alokujeme paměť pro sloupce: `*(matrix + r) = malloc(s * sizeof(float));`.

2 Funkce

V kontextu pointerů můžeme předávat adresy proměnných do funkcí, abychom mohli měnit jejich hodnoty. Například:

`int x = 10;` a `int y = 11;` můžeme změnit pomocí funkce:

`void fn(int z, int *p)` tím, že nastavíme `z = 20;` a `*p = 30;`. Po zavolání `fn(x, &y);` zůstane `x = 10`, ale `y = 30` díky použití pointeru.

Další příklad s dynamickou alokací paměti:

`void fn(int **pa)` provede alokaci paměti pomocí `*pa = malloc(sizeof(int));` a následně nastaví hodnotu `**pa = 10;`.

V `main()` můžeme použít pointer k tomu, abychom zajistili, že alokovaná paměť nezmizí, a pointer v `main()` bude ukazovat na správně alokovanou paměť.

3 Soubory

Standardní vstup a výstup jsou v C rozděleny na tři kanály:

- `stdin` - standardní vstup
- `stdout` - standardní výstup
- `stderr` - výstup pro chyby

Směrování chybového výstupu v Linuxu:
Přesměrování chybového výstupu do souboru se provádí následovně:
`příkaz 2> soubor`. Například:
`ls neexistujici_soubor 2> error.log`.

Pro přesměrování jak výstupu, tak chybového výstupu do stejného souboru použijeme:
`příkaz > soubor 2>&1`. Například:
`ls neexistujici_soubor > output.log 2>&1`.
Pokud chceme oddělit výstup a chyby do různých souborů:
`příkaz > stdout.log 2> stderr.log`. Například:
`ls > output.log 2> error.log`.
Chceme-li přidat chybový výstup do existujícího souboru, použijeme append:
`příkaz 2>> soubor`.
Přesměrování výstupu na `/dev/null` (zahodit výstup) se provádí pomocí:
`příkaz 2> /dev/null`.

4 Řetězení příkazů (pipeline) v Linuxu

Řetězení příkazů se provádí pomocí znaku `|`.
Například příkaz:
`cat txt.txt | my_prog | grep "ahoj"`
znamená:

- `cat txt.txt` vypíše obsah souboru `txt.txt` do standardního výstupu.
- `| my_prog` přesměruje výstup z `cat txt.txt` do programu `my_prog`, který zpracovává text.
- `| grep "ahoj"` filtruje výstup z `my_prog` a vypisuje pouze řádky obsahující `"ahoj"`.

Optimalizace: Pokud `my_prog` umí číst soubor přímo, můžeme příkaz zjednodušit:
`my_prog txt.txt | grep "ahoj"`.

5 Použití getchar() a putchar()

Při čtení znaků z vstupu pomocí funkce `getchar()` můžeme hodnotu vrácenou funkcí přetypovat poté, co zkontrolujeme, zda se nejedná o `EOF` (End Of File). Následně můžeme znak předat funkci `putchar()`, která ho vypíše na výstup.

`getchar()` čte jeden znak ze standardního vstupu a vrací ho jako hodnotu typu `int`. Vrací buď hodnotu znaku, nebo `EOF`, což je speciální hodnota indikující konec vstupu nebo chybu při čtení. Z tohoto důvodu je návratový typ `int`, aby bylo možné odlišit znaky od `EOF`.

Příklad použití:

- Po zavolání `getchar()` je důležité zkontrolovat, zda vrácená hodnota není `EOF`.
- Pokud je hodnota platná (není `EOF`), můžeme ji přetypovat na `char` a následně předat funkci `putchar()`, která očekává právě `char`.

6 Práce s čísly

Pro práci s čísly ve vstupu používáme funkci `scanf`, například:
`scanf("%d", &x);`, kde proměnná `x` bude obsahovat načtenou hodnotu.

7 Práce se soubory

`int printf(const char *format, ...)` je funkce, jejíž správné použití zahrnuje například `printf("%s", str)`.

8 Formátovací značky

Konverze je povinný parametr (`d`, `f`, `s`, `c`, ...). Modifikátory, jako `hh` pro `short short`, umožňují upřesnit typ dat. Zarovnání - hodnoty můžeme zarovnat doleva nebo doprava. Pro výpis speciálních hodnot se používá `printf` s formátovacími značkami, jako `%f` nebo `%F`.

Při použití `scanf("%s", buffer)` hrozí nebezpečí čtení za hranici alokované paměti. Pro čtení celého řádku můžeme použít `int puts()`.

9 Binarni soubory

Práce s binárními soubory se principiálně neliší od práce se soubory textovými. Jazyk C nezná datový typ souboru a nemá syntaktické prostředky pro práci se soubory, proto si pomáháme knihovnami.

`FILE *` je struktura definovaná ve `stdio.h`, která popisuje soubor. `FILE *fopen` otevírá soubor zadaného jména a vrací `NULL`, pokud se to nepovede, což je nutné zkontrolovat. Mod je způsob otevření souboru, zadaný jako řetězec.

`FILE fclose(FILE *file)` uzavírá soubor a pokud se to nepovede, vrací `EOF`. Zápisy se provádějí do bufferu a po nějaké době se zapíší přímo do souboru. Předčtení souboru pomocí `abort()` může vést k poškození souboru!

Při práci se soubory také můžeme použít:

- `int getc(FILE *file)` - čte znak ze souboru.
- `int putc(int c, FILE *file)` - zapisuje znak do souboru.

Standardní vstupy a výstupy:

- `stdio` - `FILE *stdin;`
- `FILE *stdout;`
- `FILE *stderr;`

Můžeme je použít i pro standardní výstup pomocí `int fscanf` a `int fprintf`.

Pro čtení řádku s určením maximální délky můžeme použít `char *fgets`, což je lepší z hlediska bezpečnosti. Další užitečné funkce:

- `int ferror(FILE *file)` - zjistí, zda došlo k chybě při práci se souborem.
- `int errno` z knihovny `<errno.h>` obsahuje poslední chybu I/O operace.
- Funkce z `<string.h>` - `char *strerror(int errnum)` vrací popis chyby.

ftall(f) hodnota kurzoru fsec(f, podivat se na video o praci se souborem C