Jazyk C: Opakování 1

Cvičení z předmětů Operační systémy 1 a 2 jsou zaměřena na pochopení a procvičení základních služeb poskytovaných operačním systémem. Pro zvládnutí těchto cvičení je naprosto zásadní zvládnutí programování v jazyce C. Proto v úvodním cvičení připomeneme základní aspekty tohoto jazyka, se kterými budeme potřebovat pracovat.[1]

1 Datové typy

Jazyk C nabízí širokou škálu datových typů, ať už skalárních (obsahují jednu hodnotu) tak složených. Určitou nevýhodou je, že různé překladače jazyka C mohou nabízet datové typy odlišných vlastností.[2]

1.1 Skalární datové typy

Pro reprezentaci celých čísel máme pět znaménkových datových typů `char`, `short`, `int`, `long`, `long long`, které mají ještě svou neznaménkovou variantu uvozenou klíčovým slovem `unsigned`. Velikosti a rozsahy jednotlivých datových typů, jak je používají nejběžnější překladače (GCC, MSVC na platformě x86 a AMD64) ukazuje Tabulka 1.

typ	velikost (bit)	min. hodnota	max. hodnota
`char`	8	-128	127
`unsigned char`	8	0	255
`short`	16	-32.768	32.767
`unsigned short`	16	0	65.535
`int`	32	-2.147.483.648	2.147.483.647
`unsigned int`	32	0	4.294.967.295
`long`	*	*	*
`unsigned long`	*	*	*
`long long`	64	-9.223.372.036.854.775.808	9.223.372.036.854.775.807
`unsigned long long`	64	0	18.446.744.073.709.551.616

Tabulka 1: Celočíselné datové typy a rozsahy

Datové typy `int` a `long` mají v případě 32bitového překladače stejnou velikost, tj. 32 bitů. U 64bitových překladačů *v unixech* se používá `int` o velikosti 32 bitů a `long` o velikosti 64 bitů, kdežto na platformě Windows je velikost `int` a `long` stejná, tj. 32 bitů.

[1] Tento text je pouze přehledový a nečiní si ambice popsat celý jazyk C.

[2] Díky tomu je možné v jazyce C programovat na různých obvyklých i méně obvyklých platformách.

Název datového typu `char` naznačuje, že se jedná o typ sloužící k uložení znaků,[3] jedná se však o zcela obecný celočíselný typ, který může obsahovat nejen znaky ale obecně celá čísla. Následující deklarace a přiřazení jsou možné a ekvivalentí:

```
char foo = 'A';
char foo = 65;
```

K reperezentaci čísel s plovoucí řádovou čárkou slouží datové typy `float`, `double` a `long double` o velikostech 32, 64 a 80 bitů. Čísla s plovoucí řádovou čárkou jsou obvykle zpracovávána jinak, proto se jim budeme věnovat v samostatném cvičení.

1.2 Ukazatele a pole

Specifickým případem skalárních datových typů jsou ukazatele. Hodnota ukazatele ukazuje na místo v paměti, kde je uložena hodnota daného typu. Následující příklad ilustruje jejich použití.

```
int a  = 42;        // proměnná typu int
int *p = &a;        // ukazatel na hodnotu typu int
                    // operátor & (reference) je použit k získání ukazatele (adresy) proměnné a
printf("%i\n", *p); // přečtení hodnoty dané ukazatelem p
                    // operátor * (dereference) získá hodnotu
*p = 123;           // zde je operátor dereference použit ke změně hodnoty dané ukazatelem p
```

Ukazatele mají několik důležitých rolí (i) umožňují předávat argumenty odkazem, (ii) umožňují práci s poli, (iii) umožňují práci s dynamicky alokovanou pamětí (objekty).

1.2.1 Předávání argumentů odkazem

Předávání argumentů odkazem je vhodné v situacích, kdy potřebujeme uložit výsledek do připravené paměti nebo chceme vrátit hodnotu přes argument (např. pokud máme více návratových hodnot), jak ukazuje následující příklad.

```
void add(int a, int b, int *x) {
  *x = a + b;
}

int z;
add(10, 20, &z);
```

1.2.2 Pole a ukazatele

Datový typ pole představuje celými čísly indexovanou kolekci hodnot stejného typu. Pole mohou být deklarována buď s pevnou, nebo nespecifikovanou velikostí.

[3] Nejčastěji z ASCII tabulky.

```
int a[3]; // pole celých čísel o třech prvcích
int b[];  // pole celých čísel s nespecifikovanou velikostí
```

Pole jako hodnota si nenese informaci o své velikosti a jazyk nekontroluje, zda nepřistupujeme za hranici pole. Proto se následující kód sice provede, ale jeho provedení povede k nespecifikovanému chování, které se může projevit nežádoucím chováním ihned, později nebo vůbec.

```
int a[3];
int a[10] = 42;
```

Pole úzce souvisí s ukazateli. Pole můžeme chápat jako ukazatel na první prvek pole. Aritmetika s ukazateli nám pak umožňuje pole procházet, jak ilustruje následující příklad.

```
int a[3];
a[0]     = 1;         // priřadí prvnímu prvku pole hodnotu 1
*a       = 1;         // to samé s využitím ukazatele
a[1]     = 2;         // priřadí druhému prvku pole hodnotu 2
*(a + 1) = 2;         // to samé s využitím ukazatele
int *b   = a;         // přiřadí do b začátek pole a
b++;                  // pole b bude začínat na druhém prvku pole a
printf("%i\n", b[0]); // vypíše 2
```

1.2.3 Dynamická alokace paměti

K dynamické alokaci paměti slouží funkce `void *malloc(size_t size)`, která alokuje minimálně `size` bytů paměti a vrací na ni ukazatel, jak ukazují následující příklady.

```
int *p = (int *) malloc(sizeof(int));      // alokuje pamět pro jednu hodnotu typu int
int *a = (int *) malloc(sizeof(int) * 10); // alokuje pole typu int o velikosti deset prvků
```

Alokovanou paměť je nutné uvolnit pomocí funkce `free`.

Paměť vrácená funkcí `malloc` není nijak inicializovaná a může obsahovat libovolná data. Pokud potřebujeme paměť vynulovanou, můžeme použít funkci `calloc`. V případě, že potřebujeme změnit velikost alokované paměti, použijeme funkci `realloc`. Poznamenejme, že pokud chceme pomocí funkce realloc zvětšit množství alokované paměti, dojde k alokaci nového místa a data jsou do něj následně překopírována.

1.3 Řetězce

Specifickým případem ukazatelů jsou řetězce, což jsou ukazatele typu `char *`, které ukazují na první znak řetězce. Konec řetezce je indikován znakem `'\0'` (odpovídá hodnotě 0).

V případě řetězcových literálů je konec řetězce doplněn automaticky. U těchto literálů dejte pozor na to, že tyto řetězce mohou být, a často jsou, neměnné. Nelze tedy provést.

```
char *s = "abc";
s[0] = 'A'; // pravděpodobně selže
```

1.4 Pravdivostní hodnoty

Pro reprezentaci pravdivostních hodnot lze v jazyce C použít libovolný celočíselný typ včetně ukazatelů. Hodnota 0 nebo `NULL` odpovídá hodnotě *nepravda*, cokoliv jiného je chápano jako *pravda*. Následující kód ukazuje nejběžnější případy použití.

```
int a = 1;
int *p = NULL;

if (a == 2) { } // explicitní porovnání
if (a) { }      // test, jestli proměnná ,,a'' obsahuje nenulovou hodnotu
if (!p) { }     // test, zda je ukazatel roven NULL
```

1.5 Strukturované datové typy

Související hodnoty různých datových typů lze spojit do jednoho strukturovaného datového typu. Následující kód ukazuje vytvoření nového datového typu představujícího bod v rovině.

```
struct point {
    int x;
    int y;
};
```

Takto jsme vytvořili nový strukturovaný datový typ `struct point`, se kterým můžeme pracovat například následovně.

```
struct point a = { 2, 4 };
void point_print(struct point p) {
    printf("[%i, %i]\n", p.x, p.y);
}
```

Pokud chceme zavést jednodušší pojmenování, můžeme zavést alias pomocí klíčového slova `typedef`, jak ukazuje následující příklad.

```
typedef struct point {
    int x;
    int y;
} point;
```

Takto nám vznikne strukturovaný datový typ pojmenovaný jen `point`. Použití `typedef` pro deklaraci strukturovaného datavého typu není nezbytné.

Pozor, strukturované datové typy se předávájí hodnotou, tzn. předáme-li do funkce argument, kterým je hodnota strukturovaného datového typu, dojde k vytvoření kopie jeho hodnoty.

Vyzkoušejme si:

```
void point_move(struct point p, int dx, int dy) {
    p.x += dx;
    p.y += dy;
}

point_print(a);          // vypíše [2, 4]
point_move(a, 13, 37);
point_print(a);          // vypíše [2, 4]
```

Všimněme si, že v důsledku předávání argumentu `p` hodnotou došlo ke změně jednotlivých složek pouze v rámci funkce `point_move`. Pokud bychom chtěli, aby změna hodnot byla viditelná i mimo rozsah funkce `point_move`, musíme hodnotu `p` předat odkazem, a pak s ní i tak pracovat, jak ukazuje následující příklad.

```
void point_move(struct point *p, int dx, int dy) {
    (*p).x += dx;
    p->y += dy;
}

point_move(&a, 13, 37);
```

Obraty `(*p).x` a `p->x` mají shodný význam, avšak ten druhý je srozumitelnější a běžnější.

2 Operátory

Jazyk C nabízí širokou paletu operátorů, některé jsou intuitivní, např. aritmetické nebo relační operátory, těm se nebudeme podrobněji věnovat, jiné jsme již zmínili (reference a dereference), a teď zmíníme jen ty méně obvyklé nebo ty, které mají určité specifické chování. Chování základních operátorů, se kterými se běžně setkáte popisuje následující výčet.

- `a = b` (přiřazení) přiřadí do prvního operandu hodnotu druhého operandu a vyhodnotí se na jeho hodnotu
- `a++` (inkrementace) zvýší hodnotu operandu o 1 a vyhodnotí se na hodnotu před inkrementací
- `++a` (inkrementace) zvýší hodnotu operandu o 1 a vyhodnotí se na aktuální hodnotu
- `a--`, `--a` (dekrementace) analogicky inkrementaci
- `a ? b : c` (podmíněný výraz, ternární operátor) – pokud je první operand *pravda*, vyhodnotí se na druhý operand, jinak se vyhodnotí na třetí operand

- `a && b` (logický součin) – vyhodnotí se na *pravda*, pokud jsou oba operandy *pravdivé*, jinak se vyhodnotí na nepravda
- `a || b` (logický součet) – vyhodnotí se na *pravda*, pokud je alespoň jeden operand *pravda*, jinak se vyhodnotí na nepravda

U operátorů `&&` a `||` dochází ke zkrácenému vyhodnocení. Pokud je jasné, že hodnota výrazu bude pravda nebo nepravda již po vyhodnocení prvního operandu, druhý operand se již nevyhodnocuje, například následující kód je zcela validní.

```
int a = 1;
if (a || (1 / 0)) { ... }
```

2.1 Bitové operace

Při programování na té nejnižší úrovni často potřebujeme pracovat s jednotlivými bity, k čemuž slouží operátory `&` (bitový součin), `|` (bitový součet), `^` (bitová non-ekvivalence, výlučné nebo, XOR), `~` (negace, inverze bitů), << a >> (bitové posuny).

Jejich použití ukazují následující příklady:

```
  0101 0011          0101 0011          0101 0011
& 1001 0010        | 1001 0010        ^ 1001 0010
-----------        -----------        -----------
  0001 0010          1101 0011          1100 0001


   0101 0011          1001 0010          1001 0010
<<         1       >>         1       >>         1
------------       ------------       ------------
   1010 0110          1100 1001          0100 1001
                    znaménková         neznaménková
                      varianta           varianta
```

Pozor, u bitového posunu vpravo záleží na tom, jestli danou operaci provádíme s hodnotou znaménkového nebo neznaménkového typu. Pokud máme neznaménkovou hodnotu (např. `unsigned char`), doplní se vždy jako nejvyšší bit 0. U znaménkových typů (např. `int`) se doplní kopie nejvyššího bitu.

3 Úkoly k procvičení

1. Napište funkci `void int2bits(char *, int)`, která převede číslo na textový řetězec představující jeho zápis v binární podobě.
2. Napište funkci `int bits2int(char *)`, která převede textový řetězec představující zápis čísla v binární podobě (tj. `"010110010010..."`) na hodnotu typu `int`.

3. Implementujte funkci `void my_memcpy(void *dest, void *src, size_t size)`, která se chová jako funkce memcpy a přenese po jednolivých bytech obsah paměti z jednoho místa na druhé, předpokládejte, že úseky paměti se nepřekrývají.

4. Navrhněte vhodnou strukturu pro spojový seznam obsahující dvě hodnoty jméno (textový řetězec) a věk (celé číslo). Napište funkci, která bude přidávat prvky do seznamu a funkci, která vypíše obsah tohoto seznamu.

5. Napište funkci `short encode_date(char day, char month, short year)`, která zakóduje datum do 16bitového čísla následovně: `YYYY-YYYM-MMMD-DDDD`[4].

6. Napište funkci `void decode_date(short date, int *day, int *month, int *year)`, která dekóduje datum vytvořené předchozí funkcí a vrátí hodnoty pomocí předaných ukazatelů.

7. Napište funkci, která zjistí, v jakém pořadí jsou vyhodnocovány argumenty.

[4] Úlohu není možné vyřešit tak, aby měla univerzální řešení, a je nutné pracovat s nějakou kompromisní variantou.