Jazyk C pro pokročilé

- Obsah
  - Proč používat jazyk C
  - Rozdíly oproti Javě
  - Objektově orientované programování v C
  - Použití preprocesoru
  - Kódovací standardy
  - Optimalizující překladač
  - Standard C99

Proč používat jazyk C

- Starý jazyk (1970 – 1978)
- Něco mezi assemblerem a moderními OO jazyky.
- Knihovna standardních funkcí je dlouhodobě zavedená a standardizovaná – přenositelnost zdrojových kódů
- Kompilátor jazyka C je vždy první kompilátor, který se pro nový procesor portuje
  - Na portaci se podílí výrobce procesoru – rychlý kód
  - Většina architektur a instrukčních sad procesorů je ovlivněna požadavky a filozofií jazyka C

Proč používat jazyk C (II.)

- Základní datové typy jazyka jsou definovány tak, aby odpovídaly vlastnostem cílového procesoru. Programy jsou efektivní a při troše snahy lze psát programy přenositelné mezi 8, 16, 32 i 64 bitovými procesory.
- Knihovny napsáné v jazyce C se snadno začleňují do jiných (modernějších) jazyků.
- Jazyk umožňuje i velmi nízkoúrovňový zápis kódu natolik přizpůsobený zpracovávání instrukcí procesorem, že téměř není potřeba psát části programu v assembleru (lepší přenositelnost).

Rozdíly oproti jazyku Java

- Jazyk nedefinuje koncepci objektů. Lze si naprogramovat „ručně“.
- Neexistují výjimky – obtížnější ošetřování chyb (goto).
- Použití pointerů je explicitní.
- Není automatická správa paměti. **Garbage collector** lze také doprogramovat. Lze použít i **reference counting**.
- Rozhraní programových modulů není zakompilováno v .o souborech, ale v samostatných .h souborech.

Objektové programování v C

- Základní vlastnosti OOP
  - zapouzdřenost
  - polymorfizmus
  - dědičnost

Zapouzdřenost

- Datové položky deklarujeme jako strukturu
- Metody jsou funkce
- Parametr this se musí předávat explicitně

```
typedef struct bod {
   int x, y;
} bod_t;
void bod_init
   (bod_t *this) {
   this->x = 0;
   this->y = 0;
}
```

```
bod_t A, *B;
bod_init(&A);
B = malloc(sizeof( bod_t)
  );
bod_init(B);
```

```
void bod_draw(bod_t *this, int color) {
   drawpixel(this->x, this->y, color);
}
```

Polymorfizmus

- Tabulka virtualních metod (VMT) – pointery na funkce.
- Použití: drivery, síťové protokoly atd.

```
typedef struct bod {
    int x, y;
    void (*draw)(struct int color);
} bod_t;

void bod_draw2(bod_t *this, int color) {
    fillcircle(this->x, this->y, 5, color);
}
void bod_init(bod_t *this, int x, y) {
    this->x = x; this->y = y;
    this->draw = bod_draw;
}
void bod_init2(bod_t *this, int x, y) {
    this->x = x; this->y = y;
    this->draw = bod_draw2;
}
```

Polymorfizmus - použití

```
bod_t  A, B;

bod_init(&A, 10, 20);
bod_init2(&B, 20, 10);

A.draw(&A, WHITE);
B.draw(&B, RED);
```

Dědičnost

```
typedef struct kruznice {
   bod_t bod;
   int polomer;
} kruznice_t;


kruznice_init(kruznice_t *this, x, y, r) {
   bod_init(&this->bod, x, y);
   this->polomer = r;
   this->bod.draw = kruznice_draw;
}
```

- Nevýhody: nutnost přetypovávání
- Příklady: Grafická knihovna GTK, Jádro Linuxu, ...

Preprocesor
Priority operátorů

- Mocný nástroj. Při špatném použití vede k chybám.
- Makra by se měly tvářit jako prvky jazyka (proměnné, funkce) a neměly by mít vedlejší efekty.

```
#define MIN_NUM 10
#define KONST MIN_NUM + 1

x = 2*KONST;

#define KONST (MIN_NUM + 1)

#define ceil_div(x, y) (x + y - 1) / y
a = ceil_div (b & c, sizeof (int));

#define ceil_div(x, y)  ((x) + (y) - 1) / (y)
#define ceil_div(x, y) (((x) + (y) - 1) / (y))
```

Preprocesor II.
Složené příkazy

```
#define START_WD(t) \
   set_reg(CTRL, 0x10); set_reg(TIME, t)

if (watchdog)
   START_WD(10);

#define START_WD(t) \
   { set_reg(CTRL, 0x10); set_reg(TIME, t); }

if (watchdog)
   START_WD(10);
else
   printf(„no watchdog“);

#define START_WD(t) \
   do { set_reg(CTRL, 0x10); set_reg(TIME, t); } while (0)
```

Preprocesor III.
Zdvojení vedlejších efektů

```
#define MIN(x,y) ((x) < (y) ? (x) : (y))

next = min (x + y, foo (z));

/* GNU Extension */
#define min(X, Y)                       \
({ typeof (X) x_ = (X);                 \
   typeof (Y) y_ = (Y);                 \
   (x_ < y_) ? x_ : y_; })

/* Non GNU */
{
  int tem = foo (z);
  next = min (x + y, tem);
}
```

Kódovací standardy

- Zajišťují čitelnost kódu pro ostatní lidi z firmy/projektu.
- Zabraňování častým chybám.
- Definují:
  - odsazování
  - pojmenovávání funkcí, proměnných, parametrů
  - formát komentářů (automatická dokumentace)
  - dělení kódu do souborů
  - používání typů
- MISRA – pravidla pro použití jazyka C pro aplikace kritické z hlediska bezpečnosti (safety-critical applications)
  - Původně pro automobilový průmysl. Používá se i jinde.

Linux Codign Style
/usr/src/linux/Documentation/CodingStyle

- Odsazování o 8 mezer (tabulátor)
  - dobře čitelné i po 20 hodinách za monitorem
- Dělení řádků delších než 80 znaků
- Umisťování složených závorek
- Jména proměnných, funkcí
  - globální proměnné pojmenovávat srozumitelně, u některých lokálních to není nutné (tmp, i)
  - (v knihovnách používat prefixy, prefix „_“ je vyhražen pro POSIX)
- Funkce
  - vejde se na obrazovku 80x24 znaků (výjimka: dlouhý switch příkaz)
  - Dlouhé funkce dělit do menších se srozumitelným názvem (použití inline modifikátoru)
  - Max. 5 – 10 lokálních proměnných.

Linux Codign Style II.
/usr/src/linux/Documentation/CodingStyle

- Centralizované opouštění funkcí (ošetření chyb)
  - použití goto
  - čitelnější než použití v podmíněných příkazech

```
int fun(int a)
{
        int result = 0;
        char *buffer = kmalloc(SIZE);

        if (buffer == NULL)
                return -ENOMEM;

        if (condition1) {
                while (loop1) {
                        ...
                }
                result = 1;
                goto out;
        }
        ...
out:
        kfree(buffer);
        return result;
}
```

Linux Codign Style III.
/usr/src/linux/Documentation/CodingStyle

- Datové struktury používají reference counting
- Makra
- Hlášení jádra se vypisují bez teček na konci
- Alokace paměti
  - p = kmalloc(sizeof(*p), ...);
- Další, nepsaná pravidla:
  (http://www.kroah.com/linux/talks/ols_2002_kernel_codingstyle_paper/codingstyle.ps)
  - místo typů (**typedef struct {...} urb_t**) používat **struct urb**
  - používání standardních otestovaných funkcí (práce s řetězci, dynamické seznamy, pořadí bytů, atd.)
  - Nepoužívat číselné konstanty. Definovat pro ně makra.
  - Omezit používání #ifdef v .c souborech
    - nadefinovat makra v .h, která třeba nedělají nic

MISRA

Některá pravidla

- Výrazy s && nebo || v if (…) se povinně závorkují
- pro logické operace použít typ BOOL definovaný pomocí typedef (statické analyzátory chybu odhalí)
- Lokální proměnné se nesmí jmenovat stejně jako globální proměnné
- Pokud to jde, číselné konstanty mají příponu určující typ (0x123456L), osmičkové konstanty jsou zakázány (0123)
- Operátor čárka se nepoužívá (pouze ve for (…))
- Pokud to není nutné, nepoužívat přetypování.
- Nesmí se používat continue, goto, break (kromě switch)
- Každý switch má default část, každý case má break.

MISRA II.

Některá pravidla

- Nepoužívat pointerovou aritmetiku (výjimečně ++ a --)
- Nad ukazateli nepoužívat relační operátory kromě == a !=.
- Zákaz používání globálních proměnných. Existují výjimky.
- Nepoužívat rekurzivní funkce.

Optimalizující překladač (GCC)

- Použití modifikátoru **volatile** pro přístup k hardwaru a pro proměnné modifikované v obslužné rutině přerušení.
- Vyšší úroveň
  - Odstranění nepoužitých příkazů (if (0))
  - Odstranění nepoužitých proměnných
  - Propagace konstant
  - Propagace proměnných do výrazu
    - x = a + c1;
    - if (x == c2) goto ... else goto ...
    - if (a == (c2 - c1)) goto ... else goto …
  - Eliminace násobných ukládání do proměnných
  - Optimalizace cyklů (náhrada operaci SIMD instrukcemi atd.)
  - Zjednodušení vestavěných funkcí (např. memcpy).
  - Volání funkcí na konci jiné funkce může být nahrazeno skokem.

Optimalizující překladač (GCC)

- Nižší úroveň
  - Eliminace výpočtů stejných podvýrazů pomocí ukládání mezivýsledků do pomocných proměnných.
  - Výběr adresních módu v závislosti na jejich „ceně“.
  - Optimalizace cyklů (rozbalování, modulo scheduling, …)
  - Kombinování více operací do jedné instrukce
  - Alokace správných registrů pro operandy a proměnné, určení co bude v registru a co proměnná na zásobníku, proměnné mohou být i přesouvány mezi registrem a zásobníkem
  - Přeskupení instrukcí tak, aby byly vykonány nejrychleji

Standard C99

- definuje následující typy (stdint.h)
  - int32_t , uint32_t, int16_t
- možnost deklarovat lokální proměnné jinde než na začátku funkce
- Standardizuje inline funkce
- přidává boolean typ
- Povoluje komentáře ve stylu C++ (//)
- Podpora pro pole různé délky na konci struktur (int pole[])
- Makra s proměnným počtem parametrů
- Kvalifikátor **restrict** – dovoluje kompilátoru lépe optimalizovat přístup k proměnným