Ladění chyb

Tématicky zaměřený vývoj aplikací v jazyce C
skupina Systémové programování – Linux

Jiří Novosad

Fakulta informatiky
Masarykova univerzita
novosad@fi.muni.cz

7. října 2016

C Preprocesor

základní konstrukce, časté chyby

Co umí preprocesor

- Vkládání zdrojových souborů
  - `#include <file1.h>`, `#include "file2.h"`
  - Měly by mít syntax podobnou jazyku C

Co umí preprocesor

- Vkládání zdrojových souborů
  - `#include <file1.h>`, `#include "file2.h"`
  - Měly by mít syntax podobnou jazyku C
- Předdefinovaná makra
  - `__LINE__`, `__DATE__`, `__TIME__`, `__func__`, ...

Co umí preprocesor

- Vkládání zdrojových souborů
  - `#include <file1.h>`, `#include "file2.h"`
  - Měly by mít syntax podobnou jazyku C
- Předdefinovaná makra
  - `__LINE__`, `__DATE__`, `__TIME__`, `__func__`, ...
- Definice maker
  - ```
    #define MAKRO "hodnota makra az do konce radku"
    #define makro1() printf("Dneska je úžasný den!\n")
    #define makro2(x, y) printf("%d, %d\n", x, y)
    #define makro3(...) printf(__VA_ARGS__)
    ```
    ```

Co umí preprocesor

- Vkládání zdrojových souborů
  - `#include <file1.h>`, `#include "file2.h"`
  - Měly by mít syntax podobnou jazyku C
- Předdefinovaná makra
  - `__LINE__`, `__DATE__`, `__TIME__`, `__func__`, ...
- Definice maker
  - ```
    #define MAKRO "hodnota makra az do konce radku"
    #define makro1() printf("Dneska je úžasný den!\n")
    #define makro2(x, y) printf("%d, %d\n", x, y)
    #define makro3(...) printf(__VA_ARGS__)
    ```
- Podmíněný překlad
  - `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, `#endif`
    ```

Co umí preprocesor

- Vkládání zdrojových souborů
  - `#include <file1.h>`, `#include "file2.h"`
  - Měly by mít syntax podobnou jazyku C
- Předdefinovaná makra
  - `__LINE__`, `__DATE__`, `__TIME__`, `__func__`, ...
- Definice maker
  - ```
    #define MAKRO "hodnota makra az do konce radku"
    #define makro1() printf("Dneska je úžasný den!\n")
    #define makro2(x, y) printf("%d, %d\n", x, y)
    #define makro3(...) printf(__VA_ARGS__)
    ```
- Podmíněný překlad
  - `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, `#endif`
- Ostatní
  - `#`, `##`, `#error`, `#pragma`, ...
  - `#define COMMAND(NAME) { #NAME, NAME ## _command }`
- http://gcc.gnu.org/onlinedocs/cpp/
    ```

Chybné použití preprocesoru I

Obecně používejte jen to, o čem víte, jak to skutečně funguje a když jste si vědomi důsledků.

závorky, závorky, závorky

- Priorita oprátorů

```
#define MULT(x,y) x*y

int z = MULT(3 + 2, 4 + 3);
```

Chybné použití preprocesoru I

Obecně používejte jen to, o čem víte, jak to skutečně funguje a když jste si vědomi důsledků.

závorky, závorky, závorky

- Priorita oprátorů

```
#define MULT(x,y) x*y

int z = MULT(3 + 2, 4 + 3);

int z = 3 + 2*4 + 3;
```

Chybné použití preprocesoru I

Obecně používejte jen to, o čem víte, jak to skutečně funguje a když jste si vědomi důsledků.

závorky, závorky, závorky

- Priorita oprátorů

```
#define MULT(x,y) x*y

int z = MULT(3 + 2, 4 + 3);

int z = 3 + 2*4 + 3;

#define ADD_TWO(x) x+2

int y = ADD_TWO(3) * 5;
```

Chybné použití preprocesoru I

Obecně používejte jen to, o čem víte, jak to skutečně funguje a když jste si vědomi důsledků.

závorky, závorky, závorky

- Priorita oprátorů

```
#define MULT(x,y) x*y

int z = MULT(3 + 2, 4 + 3);

int z = 3 + 2*4 + 3;

#define ADD_TWO(x) x+2

int y = ADD_TWO(3) * 5;

int y = 3+2 * 5;
```

Chybné použití preprocesoru I

Obecně používejte jen to, o čem víte, jak to skutečně funguje a když jste si vědomi důsledků.

závorky, závorky, závorky

- Priorita oprátorů

```
#define MULT(x,y) x*y

int z = MULT(3 + 2, 4 + 3);

int z = 3 + 2*4 + 3;

#define ADD_TWO(x) x+2

int y = ADD_TWO(3) * 5;

int y = 3+2 * 5;


#define MULT(x,y) ((x) * (y))

#define ADD_TWO(x) ((x)+2)
```

Chybné použití preprocesoru II

závorky, závorky, závorky

- Závorky kolem bloku kódu

```
#define SWAP(a,b) a ^= b; b ^= a; a ^= b;


if (x < 0)
    SWAP(x, y);
```

Chybné použití preprocesoru II

závorky, závorky, závorky

- Závorky kolem bloku kódu

```
#define SWAP(a,b) a ^= b; b ^= a; a ^= b;


if (x < 0)
    SWAP(x, y);
else
    SWAP(y, x);
```

Chybné použití preprocesoru II

závorky, závorky, závorky

- Závorky kolem bloku kódu

```
#define SWAP(a,b) a ^= b; b ^= a; a ^= b;


if (x < 0)
    SWAP(x, y);

else
    SWAP(y, x);


#define SWAP(a,b) do { \
    a ^= b;            \
    b ^= a;            \
    a ^= b;            \
} while (0)
```

Chybné použití preprocesoru III

- Změna toku programu

```
#define FOO(x)                  \
do {                            \
    if (blah(x) < 0)            \
        return -EBUGGERED;      \
} while(0)
```

Chybné použití preprocesoru III

- Změna toku programu

```
#define FOO(x)                  \
do {                            \
    if (blah(x) < 0)            \
        return -EBUGGERED;      \
} while(0)
```

- Přístup k lokálním proměnným

```
#define FOO(val) bar(index, val)
```

Chybné použití preprocesoru III

- Změna toku programu

```
#define FOO(x)                  \
do {                            \
    if (blah(x) < 0)            \
        return -EBUGGERED;      \
} while(0)
```

- Přístup k lokálním proměnným

```
#define FOO(val) bar(index, val)
```

- Argumenty s vedlejšími efekty

```
#define circ(a, b) (((a)*(a))+((b)*(b)))
int x = 0, y = 0;
int z = circ(++x, ++y)
```

Chybné použití preprocesoru III

- Změna toku programu

```
#define FOO(x)                  \
do {                            \
    if (blah(x) < 0)            \
        return -EBUGGERED;      \
} while(0)
```

- Přístup k lokálním proměnným

```
#define FOO(val) bar(index, val)
```

- Argumenty s vedlejšími efekty

```
#define circ(a, b) (((a)*(a))+((b)*(b)))
int x = 0, y = 0;
int z = circ(++x, ++y)
```

`// ++x` a `--y` jsou vyhodnoceny 2 krát

Chybné použití preprocesoru III

- Změna toku programu

```
#define FOO(x)                  \
do {                            \
    if (blah(x) < 0)            \
        return -EBUGGERED;      \
} while(0)
```

- Přístup k lokálním proměnným

```
#define FOO(val) bar(index, val)
```

- Argumenty s vedlejšími efekty

```
#define circ(a, b) (((a)*(a))+((b)*(b)))
int x = 0, y = 0;
int z = circ(++x, ++y)
```

`// ++x` a `--y` jsou vyhodnoceny 2 krát

`// z` má nedefinovanou hodnotu

Chybné použití preprocesoru IV

- Středníky

```
#define PRETTY_PRINT(msg) printf(msg);
if (n < 10)
    PRETTY_PRINT("n is less than 10");
else
    PRETTY_PRINT("n is at least 10");
```

Chybné použití preprocesoru IV

- Středníky

```
#define PRETTY_PRINT(msg) printf(msg);
if (n < 10)
    PRETTY_PRINT("n is less than 10");
else
    PRETTY_PRINT("n is at least 10");
```

- Pokud můžete téhož dosáhnout funkcí, nepoužívejte makra

Premature optimization is the root of all evil.
– Donald E. Knuth

Logování událostí

chybové a debuggovací výpisy, syslogd

Formát zpráv

Každá zpráva **MUSÍ** být **SROZUMITELNÁ**!

Zvažte

- kdo bude zprávy číst – jinak bude vypadat debuggovací hláška a jinak chybová hláška, kterou uvidí uživatel
- kolik zpráv je potřeba (podrobnost výpisu) – je dobrý nápad mít několik úrovní vypisovaných hlášek
- jak/čím zprávu zobrazit – stderr, logovací soubory, email, ...

Zprávy pro debuggovací účely

Zprávy jsou pro vás – vypisujte informace, které mají smysl a hlavně identifikují místo/zdroj problému

Využívejte makra preprocesoru

- `__FILE__`, `__LINE__`, (`__func__`)
- `__DATE__`, `__TIME__`

Zprávy pro debuggovací účely

Zprávy jsou pro vás – vypisujte informace, které mají smysl a hlavně identifikují místo/zdroj problému

Využívejte makra preprocesoru

- `__FILE__`, `__LINE__`, (`__func__`)
- `__DATE__`, `__TIME__`

`DEBUG_MSG`

```
extern int debug_level;
#ifdef DEBUG
# define DO_DEBUG 1
#else
# define DO_DEBUG 0
#endif
#define DEBUG_MSG(level, ...)                    \
    if (DO_DEBUG)                                \
        do {                                     \
            if (debug_level >= level) {          \
                fprintf(stderr, __VA_ARGS__);    \
            }                                    \
        } while (0)
```

Syslog

- Komplexní systém pro logování informací pro případnou pozdější analýzu.
- Zprávy lze dělit podle zdroje i podle důležitosti.
- Zprávy lze zapisovat na konkrétní zařízení (tiskárna, konsole), do lokálních souborů (`/var/log/`) nebo na vzdálený server.
- V prostředí GNU/Linuxu postupně několik generací logovacích nástrojů (`syslogd(8)`, `syslog-ng(8)` a `rsyslogd(8)`)
- Konfigurace je např. v `/etc/rsyslog.conf/`, `/etc/rsyslog.d/*`[1]

```
#include <syslog.h>

void openlog(const char *ident, int option, int facility);
void syslog(int priority, const char *format, ...);
void closelog(void);
```

[1] podle použitého daemona se může lišit

úkol

- použijte připravený debug.c a rozšiřte makro DEBUG_MSG:
- na základě hodnoty nové globální proměnné `debug_destination` půjde zpráva na chybový výstup nebo do Syslogu
- hodnotu této proměnné lze také zadat na příkazovém řádku – jako druhý argument
- `debug_level` ja „naše“ úroveň, neodpovídá úrovni ze `syslog(3)`
- Tip: `LOG_PERROR`

Testování

na co si dávat pozor a jak chyby řešit

Motivace

raketa Ariane 5

- nastala výjimka, jejíž ošetření bylo vypnuté z výkonnostních důvodů
- havárie sw $\rightarrow$ havárie rakety

havárie Mars Polar Lander

- signál ze senzorů byl interpretován jako dosednutí na zem, ačkoli zařízení bylo asi ve 40 m
- chyba byla v jediném řádku kódu

Typologie chyb I

- **Textové chyby**
  - typografie, gramatika a překlepy
  - chyby v překladu, problemy s dokumentací
  - obvykle snadná oprava, nízká priorita
  - úloha pro spell-checker
- **Pády aplikace**
  - Segmentation fault, buffer overflows, ...
  - kritický význam, často těžko reprodukovatelné
  - úloha pro debugger
- **Neočekávané chování**
  - program se chová jinak, než uživatel očekává nebo/a je popsáno v dokumentaci
  - význam se může lišit, často přístup bug $\rightarrow$ feature

Typologie chyb II

- **Memory leaky**
  - program neuvolňuje alokovanou paměť
  - význam se liší podle typu aplikace
  - úloha pro valgrind a spol.
- **Výkonnostní problémy**
  - v určitých situacích program významně zpomalí a konzumuje nepřiměřené množství zdrojů
  - význam obvykle vysoký (DoS)
  - úloha pro profilování
- **Bezpečnostní problémy**
  - problémy s přístupovými právy, neoprávněnými zásahy do paměti, uživatelským vstupem, souběhem, ...
  - význam se liší podle účelu aplikace

Postupy – co nefunguje

kompletní testování stavový prostor je příliš velký (až nekonečný)

Testování a hledání chyb je otázkou ceny, jakou chceme zaplatit.

Postupy – co funguje

unit testy testování malých, jasně definovaných částí

systémové/funkční testy testování subsystémů a celého systému

automatizace zvyšuje frekvenci testů a snižuje jejich cenu

testování vstupů náhodné/systematické/hraniční

Testujte co nejdříve – rychle odhalená chyba je levnější.

assert()

```
#include <assert.h>
void assert(scalar expression);
```

- testování podmínky a ukončení aplikace v případě její nesplnění – dáváme najevo, že očekáváme platnost invariantu
- pokud je definované makro `NDEBUG`, `assert()` nic neprovádí
- pozor na vedlejší efekty testovací podmínky
  - změna chování při ne/definovaném NDEBUG
- pozor na testování (uživatelských) vstupů
  - bezpečnostní problém, pokud definováno NDEBUG

assert()

```
#include <assert.h>
void assert(scalar expression);
```

- testování podmínky a ukončení aplikace v případě její nesplnění – dáváme najevo, že očekáváme platnost invariantu
- pokud je definované makro `NDEBUG`, `assert()` nic neprovádí
- pozor na vedlejší efekty testovací podmínky
  - změna chování při ne/definovaném NDEBUG
- pozor na testování (uživatelských) vstupů
  - bezpečnostní problém, pokud definováno NDEBUG

```
#include <stdio.h>
#include <assert.h>
int main () {
    int i = 1, j = 0;
    assert(j != 0);
    printf("%d\n", i / j);
    return 0;
}
```

Unit testy – framework check

- https://libcheck.github.io/check/
- jeden z mnoha Unit test frameworků
- hlavičkový soubor: `#include <check.h>`
- knihovna: `-lcheck`

Unit testy – framework check – úkol

úkol

- Naimplementujte funkci kontrolující zletilost uživatele.
- Funkce vrací false (0) v případě, že od narození uživatele (`born`) do času kontroly (`check_time`) neuběhlo více než 18 let.
- Jinak vrací true.

```
#include <time.h>
int is_mature(const struct tm *born, const struct tm *check_time);
```

- Pro tuto funkci si vytvořte Unit test ve frameworku check.
- Inspirovat se můžete ve studijních materiálech: `check/`.
- Makefile bude mít cíl `check`, který spustí testy.
- Důraz kladu na samotné testování – nezapomeňte na hraniční hodnoty, typické hodnoty, ...
- Tip: `-lm -lpthread -lrt -lsubunit`

Nástroje pro debuggování a reverzní inženýrství

debuggery, valgrind, ltrace, strace, ldd, file, object, nm, strip, size, objdump, perf, ...

Debuggery

`gdb(1)`

- textový debugger
- základem pro grafické nadstavby `ddd(1)`, `kdbg`, ...
- `gcc -g`

Valgrind a spol.

Sada nástrojů pro debugování a profilování.

`--tool=`

- memcheck
- cachegrind
- callgrind
- ... (`http://valgrind.org/info/tools.html`)

Další užitečné přepínače

- `--leak-check=yes`
- `--dump-instr=yes`
- `--cache-sim=yes`

Profilování

Vyhledání míst vhodných k optimalizaci – **řešte to, co má smysl!**

Postup

1. (překlad s nestandardními volbami překladače)
2. vygenerování profilovacích dat
3. analýza profilovacích dat

Nástroje

- gprof (překlad programu s volbou -pg)
- callgrind, kcachegrind

Performance Counter Subsystem (perf)

- Abstrakce hardwarových čitačů moderních CPU
- Součást přímo kernelu od verze 2.6.31
- https://perf.wiki.kernel.org/index.php/Main_Page
- Minimální režie, možnosti měření v kernel- i user-space
- Měřit lze i již běžící aplikace

Použití

- `perf record -- ./mujprog`
- `perf report`

Další nástroje (reverzního inženýrství)

- file – informace o souboru
- objdump – vypíše obsah objektového souboru
- ldd – použité dynamické knihovny
- strace – sledování systémových volání a signálů
- ltrace – sledování volání knihovních funkcí
- tcpdump/wireshark – monitorování síťové komunikace

Souborový systém procfs

V Linuxu je (téměř) všechno soubor – i procesy a parametry jádra jsou *soubory* dostupné v souborovém systému procfs (`/proc`).

- obsah souborů generuje jádro při každém open()
- největší část obsahuje informace o běžících procesech
- lze nejen číst, ale i zapisovat a měnit tak chování systému

www.root.cz/serialy/co-pred-nami-taji-proc/

Souborový systém procfs

V Linuxu je (téměř) všechno soubor – i procesy a parametry jádra jsou *soubory* dostupné v souborovém systému procfs (`/proc`).

- obsah souborů generuje jádro při každém open()
- největší část obsahuje informace o běžících procesech
- lze nejen číst, ale i zapisovat a měnit tak chování systému

www.root.cz/serialy/co-pred-nami-taji-proc/

- sysfs
- debugfs

Závěr

domácí úkoly a zdroje

Domácí úkol

- Připravte si pro další úlohy vlastní knihovnu s debugovacími makry a funkcemi
  - výpis chybových hlášek
  - výpis debugovacích hlášek
  - verze s výstupem na stderr i do syslogu (případně další výstupy)
  - různé úrovně výpisů
  - ...
- Seznamte se s možnostmi různých ladících nástrojů a vyberte si vhodné nástroje pro vás.

Zdroje

preprocesor

- gcc.gnu.org/onlinedocs/cpp/

syslog

- www.gnu.org/s/libc/manual/html_node/Submitting-Syslog-Messages.html

debuging/profiling

- www.alexonlinux.com/how-debugger-works
- www.kdbg.org/
- perf.wiki.kernel.org/index.php/Main_Page
- www.fit.vutbr.cz/∼martinek/clang/profiling.html.cs
- valgrind.org/docs/manual/QuickStart.html
- kcachegrind.github.io/html/Home.html