Ladění chyb

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

Petr Velan

Fakulta informatiky
Masarykova univerzita
velan@ics.muni.cz

Brno, 2. října 2013

C Preprocesor

Inspirováno přednáškou Rudy Čejky, FIT VUT

Co umí preprocesor

- Vkládání zdrojových souborů
  - Zcela libovolné soubory s libovolným obsahem!

Co umí preprocesor

- Vkládání zdrojových souborů
  - Zcela libovolné soubory s libovolným obsahem!
- Předdefinovaná makra
  - `__LINE__`, `__DATE__`, `__TIME__`, `__func__`, ...

Co umí preprocesor

- Vkládání zdrojových souborů
  - Zcela libovolné soubory s libovolným obsahem!
- Předdefinovaná makra
  - `__LINE__`, `__DATE__`, `__TIME__`, `__func__`, ...
- Definice maker
  - ```
    #define MA printf("Dneska je úžasný den!\n")
    #define MB(parametr) parametr
    #define MC(fmt, ...) vprintf(fmt, __VA_ARGS__)
    ```
    ```

Co umí preprocesor

- Vkládání zdrojových souborů
  - Zcela libovolné soubory s libovolným obsahem!
- Předdefinovaná makra
  - `__LINE__`, `__DATE__`, `__TIME__`, `__func__`, ...
- Definice maker
  - ```
    #define MA printf("Dneska je úžasný den!\n")
    #define MB(parametr) parametr
    #define MC(fmt, ...) vprintf(fmt, __VA_ARGS__)
    ```
- Podmíněný překlad
  - `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, `#endif`
    ```

Co umí preprocesor

- Vkládání zdrojových souborů
  - Zcela libovolné soubory s libovolným obsahem!
- Předdefinovaná makra
  - `__LINE__`, `__DATE__`, `__TIME__`, `__func__`, ...
- Definice maker
  - `#define MA printf("Dneska je úžasný den!\n")`
    `#define MB(parametr) parametr`
    `#define MC(fmt, ...) vprintf(fmt, __VA_ARGS__)`
- Podmíněný překlad
  - `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, `#endif`
- Ostatní
  - `#error`, `#pragma`, `##`, ...

Co umí preprocesor

- Vkládání zdrojových souborů
  - Zcela libovolné soubory s libovolným obsahem!
- Předdefinovaná makra
  - `__LINE__`, `__DATE__`, `__TIME__`, `__func__`, ...
- Definice maker
  - ```
    #define MA printf("Dneska je úžasný den!\n")
    #define MB(parametr) parametr
    #define MC(fmt, ...) vprintf(fmt, __VA_ARGS__)
    ```
- Podmíněný překlad
  - `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, `#endif`
- Ostatní
  - `#error`, `#pragma`, `##`, ...
- gcc.gnu.org/onlinedocs/cpp/
    ```

Chybné použití preprocesoru I

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

- závorky, závorky, závorky – kdekoli

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

Chybné použití preprocesoru I

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

- závorky, závorky, závorky – kdekoli

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

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

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

Chybné použití preprocesoru I

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

- závorky, závorky, závorky – kdekoli

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

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

#define ADD_TWO(x) x+2

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

Chybné použití preprocesoru I

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

- závorky, závorky, závorky – kdekoli

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

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

#define ADD_TWO(x) x+2

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

- Priorita oprátorů

```
#define RADTODEG(x) (x * 57.29578)
```

Chybné použití preprocesoru I

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

- závorky, závorky, závorky – kdekoli

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

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

#define ADD_TWO(x) x+2

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

- Priorita oprátorů

```
#define RADTODEG(x) (x * 57.29578)
RADTODEG(a + b) → (a + b * 57.29578)
```

Chybné použití preprocesoru I

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

- závorky, závorky, závorky – kdekoli

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

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

#define ADD_TWO(x) x+2

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

- Priorita oprátorů

```
#define RADTODEG(x) (x * 57.29578)
RADTODEG(a + b) → (a + b * 57.29578)

#define RADTODEG(x) ((x) * 57.29578)
```

Chybné použití preprocesoru I

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

- závorky, závorky, závorky – kdekoli

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

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

#define ADD_TWO(x) x+2

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

- Priorita oprátorů

```
#define RADTODEG(x) (x * 57.29578)
RADTODEG(a + b) → (a + b * 57.29578)

#define RADTODEG(x) ((x) * 57.29578)
```

- Závorky kolem bloku kódu

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

Chybné použití preprocesoru I

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

- závorky, závorky, závorky – kdekoli

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

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

#define ADD_TWO(x) x+2

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

- Priorita oprátorů

```
#define RADTODEG(x) (x * 57.29578)
RADTODEG(a + b) → (a + b * 57.29578)

#define RADTODEG(x) ((x) * 57.29578)
```

- Závorky kolem bloku kódu

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

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

#define SWAP(x,y) {a ^= b; b ^= a; a ^= b;}
```

Chybné použití preprocesoru II

- Změna toku programu

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

Chybné použití preprocesoru II

- 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 II

- 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)
```

- Makro jako l-hodnota

```
FOO(x) = y;
```

Chybné použití preprocesoru II

- 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)
```

- Makro jako l-hodnota

```
FOO(x) = y;
```

- Argumenty s vedlejšími efekty

```
#define MIN(a,b) ((a)>(b)?(b):(a))
```

`int z = MIN(x++, y--);//x` a `y` jsou vyhodnoceny 2 krát!

Chybné použití preprocesoru III

- 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 III

- 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

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__`
- `__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__`
- `__DATE__`, `__TIME__`

`ident(1)` (*strings(1)*):

```
static const char rcsid[] __attribute__((__used__)) = "$Id$"
```

Zprávy pro debuggovací účely

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

Využívejte makra preprocesoru

- `__FILE__`, `__LINE__`
- `__DATE__`, `__TIME__`

`ident(1)` (*strings(1)*):

```
static const char rcsid[] __attribute__((__used__)) = "$Id$"
```

`DEBUG_MSG`

```
extern int debug_level;
#ifdef DEBUG
#   define DEBUG_MSG(level,format,args...) \
        if(debug_level>=level){fprintf(stderr,format,##args);}
#else
#   define DEBUG_MSG(level,format,args...)
#endif
```

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 v `/etc/sysconfig/syslog`, `/etc/syslog.conf`[1]

Funkce ze `syslog.h`:

- `openlog()`
- `syslog()`
- `closelog()`

[1] podle použitého daemona se může lišit

úkol

- použijte připravený debug.c a rozšiřte makro DEBUG_MSG o možnost zapisovat do syslogu

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.
- **Výkonostní problémy**
  - v určitých situacích program významně zpomalí a konzumuje nepřiměřené množství zdrojů
  - význam obvykle vysoký
  - úloha pro profilování
- **Bezpečnostní problémy**
  - problémy s přístupovými právy, neoprávněnými zásahy do paměti, uživatelským vstupem, souběhem, ...
  - význam se liší podle účelu aplikace
  - odlišný způsob reportování chyb!

Postupy – co nefunguje

Kompletní testování stavový prostor je obvykle příliš velký

Statistické testování nefunguje u SW – výskyt chyby je ovlivnitelný příliš mnoha faktory

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

Postupy – co funguje

unit testy testování malých, jasně definovaných částí. Následovat může 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
- pokud je definované makro `NDEBUG`, assert() nic neprovádí
- pozor na vedlejší efekty testovací podmínky

Unit testy – framework check

- check.sourceforge.net
- jeden z mnoha Unit test frameworků
- hlavičkový soubor: `#include <check.h>`
- knihovna: `-lcheck`

Unit testy – framework check

- check.sourceforge.net
- jeden z mnoha Unit test frameworků
- hlavičkový soubor: `#include <check.h>`
- knihovna: `-lcheck`

úkol

- naimplementujte funkci kontrolující zletilost uživatele (18 let)
- funkce vrací 0 v případě nezletilosti
- pro tuto funkci si vytvořte Unit test ve frameworku check

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

Nástroje pro debuggování a reverzní inženýrství

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

Debuggery

`gdb(1)`

- základní textový debugger, se kterým jste se už setkali

Debuggery

`gdb(1)`

- základní textový debugger, se kterým jste se už setkali
- těžce ovladatelný $\rightarrow$ těžce použitelný
- základem pro grafické nadstavby `ddd(1)`, `kdbg`, ...

Debuggery

`gdb(1)`

- základní textový debugger, se kterým jste se už setkali
- těžce ovladatelný → těžce použitelný
- základem pro grafické nadstavby `ddd(1)`, `kdbg`, ...

DEMO – práce s kdbg(1)

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`
- `--simulate-cache=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
- oprofile

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
- oprofile

DEMO – práce s kcachegrindem (callgrind)

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`
- `pref 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 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/

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
  - identifikace verze zdrojového kódu
  - ...
- Seznamte se s možnostmi různých ladících nástrojů a vyberte si vhodné nástroje pro vás.

Zdroje

syslog

- www.linode.com/wiki/index.php/Syslog_Howto
- 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.sourceforge.net/html/Home.html