

# Základy programování

```
1 #include <stdio.h>
2 char cell[10];
3 int csv_read_cell(FILE *db)
4 {
5     int c, i = 0;
6     while ((c = fgetc(db)) != EOF && c != ':' && c != '\n')
7         cell[i++] = c;
8     cell[i] = '\0';
9     return i;
10 }
11 int main(int argc, *argv[]) {
12     FILE *csv = fopen("1", "r");
13     int i = csv_read_cell(csv);
14     while (i > 0) {
15         printf("%s\n", cell);
16         i = csv_read_cell(csv);
17     }
18     return 0;
19 }
```

"buggy.c" 19L, 377C 12,3 All

## Ladění programů

## 2020/2021

Title:FIT\_zkracene\_barevne\_CMYK\_CZ.e  
Creator:Adobe Illustrator(R) 16.0  
CreationDate:23.09.15  
CreationDate:23.09.15  
LanguageLevel:2

# Motivace

- Testování a **ladění** základní součástí každodenní aktivity programátora.
- Příklad: 1998, křižník USS Yorktown, na vstupu (člověk) omylem nula => dělení nulou. Zastavení pohonu na několik hodin.
- Příklad: 1999, NASA Mars Climate Orbiter označen za ztracený, dva moduly kominukovaly v různých jednotkách (metrické a imperiální).
- A spousta dalších ... (Ariane 5, East coast blackout, Boeing 777 ADIRU)

# Osnova přednášky

- Program nedělá, co by měl
  - *Někdy* program nefunguje. Najdeme případ, kdy.
- Chyby a jejich klasifikace
- Koncept ladění
- Ladění pomocí zdrojových kódů
  - Přezkoumání kódu (code review)
  - Záznam událostí (log)
- Interaktivní ladění
  - Debugger

# Ladění programů

- Ladění je proces **hledání a redukování chyb** a defektů v programech tak, aby se programy chovaly podle očekávání.
  - Víme o chybě, známe způsob spuštění programů pro projevení chyby.
  - Systematicky hledáme moment a místo v programu, které chybu způsobuje.
- Testování je systematický proces **zlepšování kvality** programů.
  - např.: Tvoříme testovací sadu, která by ukázala, v jakých případech program funguje.
  - Předmět ITS (Testování a dynamická analýza), ATA (Automatizované testování)
  - Standard: **ISO/IEC/IEEE 29119 Software Testing**

# Program nedělá, co by měl

```
#include <stdio.h>
char cell[100];
int csv_read_cell(FILE *db)
{
    int c, i = 0;
    while ((c = fgetc(db)) != EOF && c != ',' || c != '\n')
        cell[i++] = c;
    cell[i] = '\n';
    return i;
}
int main(int argc, char *argv[]) {
    FILE *csv = fopen(argv[1], "r");
    int len = csv_read_cell(csv);
    while (len > 0) {
        printf(cell);
        len = csv_read_cell(csv);
    }
    return 0;
}
```

# Program nedělá, co by měl

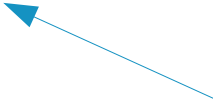
- Program má načítat buňky z CSV souboru (comma separated value) a vypisovat na výstup:

```
$ cat data.csv  
Cau,jak,se,mas?
```

```
$ ./buggy data.csv  
Segmentation fault (core dumped)
```

# Program nedělá, co by měl

```
#include <stdio.h>
char cell[100];
int csv_read_cell(FILE *db)
{
    int c, i = 0;
    while ((c = fgetc(db)) != EOF && c != ',' || c != '\n')
        cell[i++] = c;
    cell[i] = '\n';
    return i;
}
int main(int argc, char *argv[]) {
    FILE *csv = fopen(argv[1], "r");
    int len = csv_read_cell(csv);
    while (len > 0) {
        printf(cell);
        len = csv_read_cell(csv);
    }
    return 0;
}
```




V hlavě programátora: “Dokud není EOF a načtený znak není čárka nebo konec řádku, ...”

# Program nedělá, co by měl

```
#include <stdio.h>
char cell[100];
int csv_read_cell(FILE *db)
{
    int c, i = 0;
    while ((c = fgetc(db)) != EOF && c != ',' && c != '\n')
        cell[i++] = c;
    cell[i] = '\n';
    return i;
}
int main(int argc, char *argv[]) {
    FILE *csv = fopen(argv[1], "r");
    int len = csv_read_cell(csv);
    while (len > 0) {
        printf(cell);
        len = csv_read_cell(csv);
    }
    return 0;
}
```

špatný operátor



V hlavě programátora: “Dokud není EOF a načtený znak není čárka nebo konec řádku, ...”  




# Program nedělá, co by měl

- Program má načítat buňky z CSV souboru (comma separated value) a vypisovat na výstup:

```
$ cat data.csv  
Cau,jak,se,mas?
```

```
$ ./buggy_v2 data.csv  
Cau  
jak  
se
```

```
mas?
```

# Program nedělá, co by měl

- Program má načítat buňky z CSV souboru (comma separated value) a vypisovat na výstup:

```
$ cat data2.csv  
jitka.kreslikova@vut.cz,Jitka Kreslikova  
ales.smrcka@vut.cz,Ales Smrcka
```

```
$ ./buggy_v2 data2.csv  
jitka.kreslikova@vut.cz  
Jitka Kreslikova  
vut.cz  
ales.smrcka@vut.cz  
t.cz  
Ales Smrcka  
vut.cz  
t.cz
```

# Program nedělá, co by měl

```
#include <stdio.h>
char cell[100];
int csv_read_cell(FILE *db)
{
    int c, i = 0;
    while ((c = fgetc(db)) != EOF && c != ',' || c != '\n')
        cell[i++] = c;
    cell[i] = '\n';
    return i;
}
int main(int argc, char *argv[]) {
    FILE *csv = fopen(argv[1], "r");
    int len = csv_read_cell(csv);
    while (len > 0) {
        printf(cell);
        len = csv_read_cell(csv);
    }
    return 0;
}
```

globální proměnná

špatný operátor

buffer overflow

špatné ukončení

kontrola otevření

meze polí

buffer overflow

neuzavřený soubor

# Chyba, bug, klasifikace

- Chyba/defekt/selhání - obecně **bug** (definice později, pro tuto přednášku si vystačíme s pojmem “chyba”).
- Základní klasifikace podle obtížnosti odhalení:
  - Syntaktické chyby
  - Chyby sestavení
  - Základní sémantické chyby
  - Sémantické chyby
- Praxe: *“Existují jen dva druhy chyb. Triviální a velmi, velmi obtížné.”*

# Klasifikace chyb

- Syntaktické chyby:
  - *měly by být* odhalitelné překladačem; ne vždy na místě, na které se překladač odkazuje.
  - “měly by být” = i překladač může (a obsahuje) chyby
- Chyby sestavení:
  - chyby z neaktualizovaných binárních souborů (nepřeložené nové zdrojové soubory, nekompatibilita knihoven)
- Základní sémantické chyby:
  - neinicializované položky, mrtvý kód, nekompatibilita datových typů, ...

# Klasifikace chyb

- Sémantické chyby:
  - chyby neodhalitelné překladačem - jsou syntakticky správné, ale logicky špatné (neodpovídají zadání)
  - špatný operátor, špatné pořadí argumentů funkce, nekontrolované meze polí, neošetřené krajní případy, přetečení, chyby paměti, off-by-one, stack-frame, chyby v rekurzi, zaokrouhlovací chyby, statická vs. dynamická data, ...

# Koncept ladění

- Proces ladění odpovídá konceptu:
  - 1.Reprodukce (bug report)
  - 2.Diagnóza
  - 3.Oprava (bug fix)
  - 4.Integrace (reflect, commit)
- Oprava a integrace není tématem přednášky.

# Reprodukce chyby (reportér)

- Umíme chybu vyvolat účelně?
  - ...na všech počítačích?
  - ...pro každého vývojáře?
- Projevuje se chyba na poslední verzi?
- Minimalizace počtu kroků k reprodukci chyby.
- Minimalizace počtu vnějších parametrů.
- Pokud se projevuje náhodně, jistě lze náhodu odstranit (nedeterminismus vs. determinismus).
- Automatizace chyby (vede k tzv. unit a regresním testům)



# Diagnóza chyby (vývojář)

- Pochopení principu chyby, tj. hlavní příčiny (tzv. root cause)
- Prozkoumání chyby:
  - Odpovídají vstupní podmínky správnému použití?
    - Ne? → je třeba opravit požadavky použití (tj. dokumentaci programu)
  - Není chyba ve vnějším prostředí?
    - Jsou soubory a data správná? Knihovny očekávané verze? Architektura počítače, operační systém ok? Dostatečné prostředky (paměť, volné místo, rychlost/spolehlivost sítě, ...)?
  - Chyba je v programu?
    1. Minimalizace (“ukrajování”) kódu.
    2. Krokování nejmenšího kusu kódu, ve kterém se chyba projevuje
    3. Hledání místa, kde chyba nastala (nemusela se projevit).

# Pochopení principu chyby

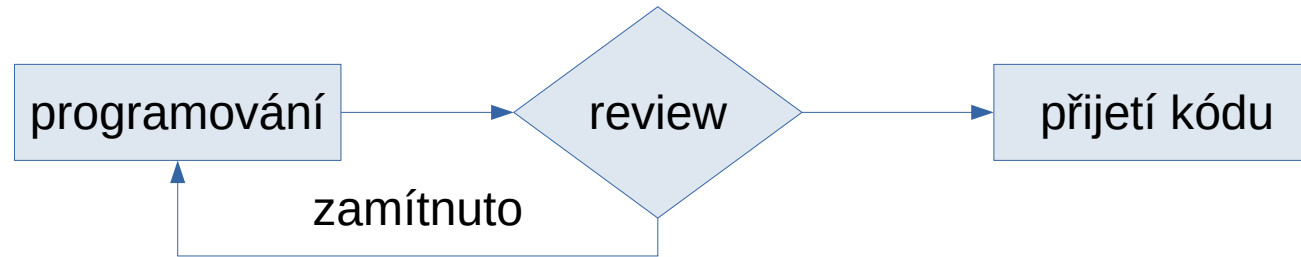
- Uspěchaná oprava chyby vede k zavedení dalších chyb
  - chyby mají tendenci se družit,
  - projev chyby na jistém místě neznamená, že chyba je v tomto místě.
- Základní check-list:
  - ✓ nesplést si projev a reálnou příčinu chyby
  - ✓ podobné chyby nejsou v jiných částech kódu
  - ✓ jedná se o chybu programování, nikoliv návrhu

# Techniky ladění

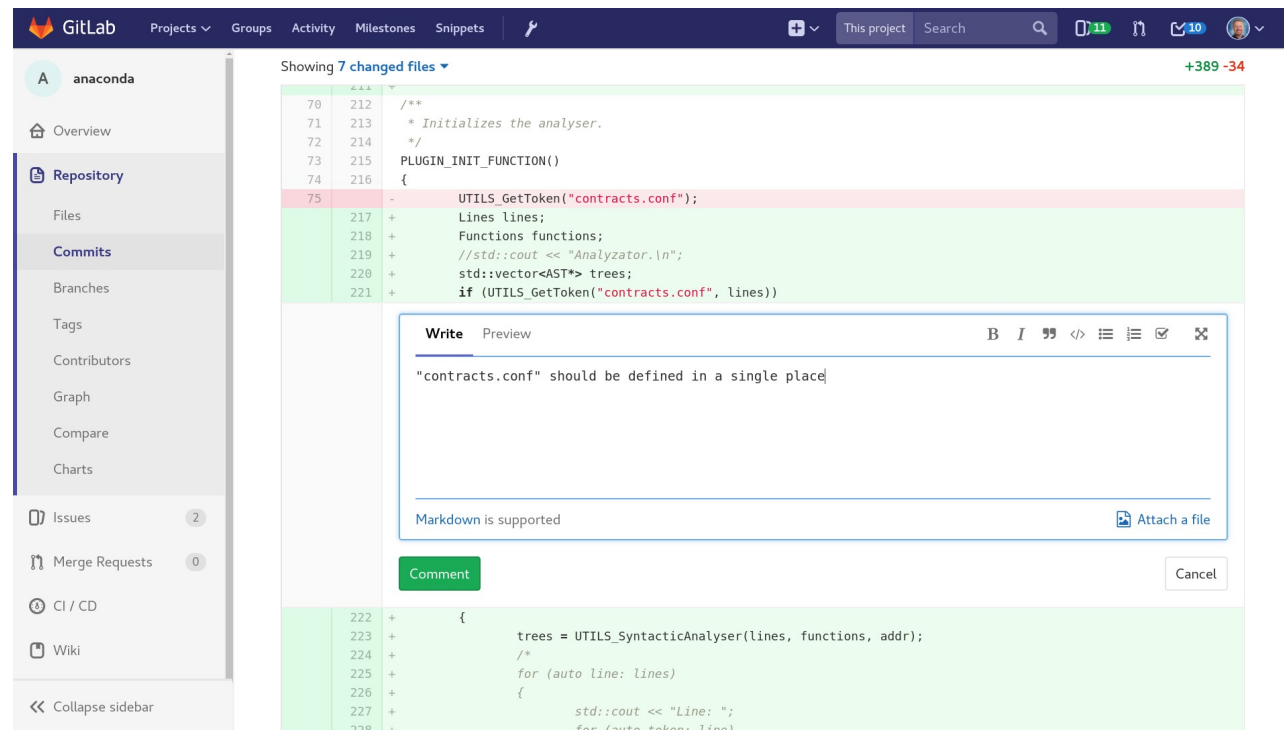
- Statická x dynamická analýza
  - statická analýza zkoumá program průchodem a vyšetřením zdrojových kódů **bez jejich spouštění**
  - dynamická analýza **zkoumá průběh** a výsledky provádění programu nebo jeho částí.

# Techniky ladění

- Ladění jako statická analýza = code review



- zahrnuje dva a více vývojářů
- code review = programování + diskuze (neplést s pair-programming)



# Techniky ladění

- Ladění jako dynamická analýza:
  - záznam sekvence událostí (log, logging, “logování”)
  - interaktivní ladění (debugging)
- Záznam, logování:
  - informování o aktuálním stavu programu
  - záznam do jiného souboru (stderr, log)
  - log = soubor určený pro hlášení o událostech (datum a čas, původ hlášení, typ hlášení, zpráva)

# Ladění pomocí výpisu událostí

```
#include <stdio.h>
char cell[100];
int csv_read_cell(FILE *db)
{
    fprintf(stderr, ">csv_read_cell(%x)\n", db);
    int c, i = 0;
    while ((c = fgetc(db)) != EOF && c != ',' && c != '\n') {
        fprintf(stderr, ">csv_read_cell: inside while: c='%c' (%d), i=%d\n", c, c, i);
        cell[i++] = c;
    }
    cell[i] = '\n';
    return i;
}

int main(int argc, char *argv[]) {
    fprintf(stderr, ">opening file %s\n", argv[1]);
    FILE *csv = fopen(argv[1], "r");
    fprintf(stderr, ">reading 1st cell\n");
    int len = csv_read_cell(csv);
    while (len > 0) {
        fprintf(stderr, ">inside while: len=%d\n", len);
        printf(cell);
        fprintf(stderr, ">reading the next cell\n");
        len = csv_read_cell(csv);
    }
    fprintf(stderr, ">exitting program\n");
    return 0;
}
```

```
fprintf(stderr, "fmt text\n", argumenty);
```

# Ladění pomocí výpisu událostí

```
$ cat data2.csv
jitka.kreslikova@vut.cz,Jitka Kreslikova
ales.smrcka@vut.cz,Ales Smrcka
$ ./buggy-fprintf data2.csv
>opening file data2.csv
>reading 1st cell
>csv_read_cell(e6d010)
>csv_read_cell: inside while: c='j' (106), i=0
>csv_read_cell: inside while: c='i' (105), i=1
...
>csv_read_cell: inside while: c='@' (64), i=16
>csv_read_cell: inside while: c='v' (118), i=17
>csv_read_cell: inside while: c='u' (117), i=18
>csv_read_cell: inside while: c='t' (116), i=19
>csv_read_cell: inside while: c='.' (46), i=20
>csv_read_cell: inside while: c='c' (99), i=21
>csv_read_cell: inside while: c='z' (122), i=22
>inside while: len=23
jitka.kreslikova@vut.cz
>reading the next cell
>csv_read_cell(e6d010)
...
>csv_read_cell: inside while: c='k' (107), i=12
>csv_read_cell: inside while: c='o' (111), i=13
>csv_read_cell: inside while: c='v' (118), i=14
>csv_read_cell: inside while: c='a' (97), i=15
>inside while: len=16
Jitka Kreslikova
vut.cz
>reading the next cell
```

# Ladění pomocí výpisu událostí

```
#include <stdio.h>
#define pmesg(s, ...) fprintf(stderr, __FILE__ ":%u: " s "\n", __LINE__, __VA_ARGS__ )
char cell[100];
int csv_read_cell(FILE *db)
{
    pmesg("csv_read_cell(%x)", db);
    int c, i = 0;
    while ((c = fgetc(db)) != EOF && c != ',' && c != '\n') {
        pmesg("c='%c' (%d), i=%d", c, c, i);
        cell[i++] = c;
    }
    cell[i] = '\n';
    return i;
}
int main(int argc, char *argv[]) {
    pmesg("opening file %s", argv[1]);
    FILE *csv = fopen(argv[1], "r");
    pmesg("reading 1st cell", 0);
    int len = csv_read_cell(csv);
    while (len > 0) {
        pmesg("len=%d", len);
        printf(cell);
        pmesg("reading the next cell", 0);
        len = csv_read_cell(csv);
    }
    pmesg("exitting program", 0);
    return 0;
}
```



# Ladění pomocí výpisu událostí

```
$ cat data2.csv
jitka.kreslikova@vut.cz,Jitka Kreslikova
ales.smrcka@vut.cz,Ales Smrcka
$ ./buggy-fprintf data2.csv
buggy-pmesg.c:16: opening file data2.csv
buggy-pmesg.c:18: reading 1st cell
buggy-pmesg.c:6: csv_read_cell(cb5010)
buggy-pmesg.c:9: c='j' (106), i=0
buggy-pmesg.c:9: c='i' (105), i=1
...
buggy-pmesg.c:9: c='@' (64), i=16
buggy-pmesg.c:9: c='v' (118), i=17
buggy-pmesg.c:9: c='u' (117), i=18
buggy-pmesg.c:9: c='t' (116), i=19
buggy-pmesg.c:9: c='.' (46), i=20
buggy-pmesg.c:9: c='c' (99), i=21
buggy-pmesg.c:9: c='z' (122), i=22
jitka.kreslikova@vut.cz
buggy-pmesg.c:23: reading the next cell
buggy-pmesg.c:6: csv_read_cell(cb5010)
...
buggy-pmesg.c:9: c='k' (107), i=12
buggy-pmesg.c:9: c='o' (111), i=13
buggy-pmesg.c:9: c='v' (118), i=14
buggy-pmesg.c:9: c='a' (97), i=15
buggy-pmesg.c:21: len=16
Jitka Kreslikova
vut.cz
buggy-pmesg.c:23: reading the next cell
```

# Podmíněný překlad

- Úprava zdrojových kódů na základě vstupních podmínek překladu.
- Jazyk C definuje tento proces jako **preprocessing** (vykonává preprocesor).
- Preprocesor je ovlivněn direktivy překladače:
  - #include, #define, #if, #ifdef, #error, ...
- Makro (macro) = symbolické jméno za textovou náhradou ve zdrojovém kódu
  - může být parametrické

# Preprocessor

```
#define MACRO_NAME [náhrada]
```

```
#define MACRO_NAME
```

Příklad

```
#define EOF (-1)
```

```
#define FILE struct _file
```

---

```
#define MACRO_NAME(p1,p2) [náhrada]
```

Příklad

```
#define ABS(x)    x<0 ? -x : x
```

```
#define MAX(a,b) a<b ? b : a
```

Pozor na vedlejší efekty:

```
int i, j, a;
```

```
a = MAX(i++, j);
```

```
a = i++<j ? j : i++;
```

# Preprocesor a ladění

- Ve std. knihovně jazyka C makro NDEBUG
  - při normálním překladu nedefinováno
  - využití v hlavičkovém souboru assert.h
  - lze využít i pro podmínění výpisu ladicích hlášení

```
#ifndef NDEBUG
```

```
#define pmsg(s,...) fprintf(stderr, ...)
```

```
#else
```

```
#define pmsg(s,...) (0)
```

```
#endif
```

# Ladění pomocí makra

```
#include <stdio.h>
#ifndef NDEBUG
#define pmsg(s, ...) fprintf(stderr, __FILE__ ":%u: " s "\n",
__LINE__, __VA_ARGS__ )
#else
#define pmsg(...) (0)
#endif
```

NDEBUG nedefinováno

```
int csv_read_cell(FILE *db)
{
    pmsg("csv_read_cell(%x)", db);
    int c, i = 0;
    ...
}
```

Originál

Preprocessed

```
int csv_read_cell(FILE *db)
{
    fprintf(stderr, "buggy-pmsg.c" ":%u: "
               "csv_read_cell(%x)" "\n", 9, db);
    int c, i = 0;
    ...
}
```

# Ladění pomocí makra

```
#include <stdio.h>
#define NDEBUG 1
#ifndef NDEBUG
#define pmsg(s, ...) fprintf(stderr, __FILE__ ":%u: " s "\n",
__LINE__, __VA_ARGS__ )
#else
#define pmsg(...) (0)
#endif
```

NDEBUG definováno



```
int csv_read_cell(FILE *db)
{
    pmsg("csv_read_cell(%x)", db);
    int c, i = 0;
    ...
}
```

Originál

Preprocessed

```
int csv_read_cell(FILE *db)
{
    (0);
    int c, i = 0;
    ...
}
```

# Ladění pomocí ladicích nástrojů

- Ladicí nástroje = nástroje pro podporu dynamického ladění programů:
  - překladač + debug info
  - debugger
  - object inspector
  - disassembler
  - sys/library tracer
- Debugger = interaktivní nástroj pro lokalizaci chyb:
  - typicky součást IDE
  - sledování proměnných (watcher)
  - možnosti záchytných bodů (break-point)
  - analýza stopy volání funkcí (backtrace)

# Debugger

- Debugger je interaktivní nástroj pro krokování programu a sledování jeho vnitřních stavů.
- Musí mít přístup ke všem částem analyzovaného programu:
  - zaručeno buď podporou operačního systému
  - nebo úpravou kódu před spuštěním (tzv. instrumentací kódu).



# Debugger - vlastnosti

- Nejznámější je **gdb** (Gnu DeBugger)
  - řádkový debugger dostupný na většině unixových systémů
  - ovládání pomocí jeho příkazové řádky:

```
$ gdb buggy-pmesg
```

```
(gdb) break main
```

nastavení záchytného bodu

```
Breakpoint 1 at 0x400630: file buggy-pmesg.c, line 21.
```

```
(gdb) run data2.csv
```

spuštění programu s argumenty

```
Starting program: /home/.../buggy-pmesg data2.csv
```

```
Breakpoint 1, main (argc=2, argv=0x7fffffffdd38) at buggy-pmesg.c:21
```

```
21 FILE *csv = fopen(argv[1], "r");
```

```
(gdb) next
```

```
23 int len = csv_read_cell(csv);
```

```
(gdb) next
```

provedení aktuálního řádku

```
24 while (len > 0) {
```

```
(gdb) print len
```

```
$1 = 23
```

```
(gdb)
```

tisk hodnoty výrazu (např. hodnoty proměnné)

# Debugger - vlastnosti

- Pro podporu ladění je třeba do binárního souboru připojit ladicí informace (debug info).
  - Překladač jazyka C: parametr -g
- Nadstavby s GUI v rámci IDE, i jiné jazyky:
  - eclipse, Visual .NET Studio, Xcode, IntelliJ, ...
- I další rozšíření:
  - změna hodnot, změna aktuálního řádku provádění kódu (PLOC = Program LOCation, neplést s SLOC),
  - ladění vícevláknových programů,
  - možnost krokování zpět (návrat do minulosti), ...
- Příklad viz demonstrační cvičení.

# Reference, další odkazy

- ISO C99, kapitola 6.10: Preprocessing directives.
- Josef Reidinger. [Code reviews v praxi](#). 2014.
- P. Adragna. [Software debugging techniques](#). CERN. 2008.
- David Martinek.  
[Nedělejte zbytečné chyby při programování v C](#). FIT VUT. 2011.
- Materiály ke kurzu CS 1206, Virginia Tech.  
[Program Bug Examples](#). 2000.
- A. Karpov, E. Ryzhkov.  
[100 bugs in Open Source C/C++ projects](#). 2012.