

Coding Examples

Jan Faigl

Department of Computer Science

Faculty of Electrical Engineering

Czech Technical University in Prague

Lecture 09

B3B36PRG – Programming in C

Overview of the Lecture

- Part 1 – Undefined behaviour and inspecting implementation

Program Compilation

Undefined Behaviour

Comparing C to Machine Code

- Part 2 – Debugging

Debugging

- Part 3 – Examples

Named pipes

Multi-thread Applications – Semestral Project

Part I

Part 1 – Undefined behaviour and inspecting implementation

Arguments of the `main()` Function

- During the program execution, the OS passes to the program the number of arguments (`argc`) and the arguments (`argv`).

In the case we are using OS.

- The first argument is the name of the program.

```
1 int main(int argc, char *argv[])
2 {
3     int v;
4     v = 10;
5     v = v + 1;
6     return argc;
7 }
```

`lec09/var.c`

- The program is terminated by the `return` in the `main()` function.
- The returned value is passed back to the OS and it can be further use, e.g., to control the program execution.

Reminder

Example of Compilation and Program Execution

- Building the program by the `clang` compiler – it automatically joins the compilation and linking of the program to the file `a.out`.

```
clang var.c
```

- The output file can be specified, e.g., program file `var`.

```
clang var.c -o var
```

- Then, the program can be executed as follows.

```
./var
```

- The compilation and execution can be joined to a single command.

```
clang var.c -o var; ./var
```

- The execution can be conditioned to successful compilation.

```
clang var.c -o var && ./var
```

Programs return value — 0 means OK.

Logical operator && depends on the command interpret, e.g., `sh`, `bash`, `zsh`.

Reminder

Example – Program Execution under Shell

- The return value of the program is stored in the variable `$?`.

sh, bash, zsh

- Example of the program execution with different number of arguments.

```
./var
```

```
./var; echo $?
```

```
1
```

```
./var 1 2 3; echo $?
```

```
4
```

```
./var a; echo $?
```

```
2
```

Reminder

Example – Processing the Source Code by Preprocessor

- Using the `-E` flag, we can perform only the preprocessor step.

```
gcc -E var.c
```

Alternatively `clang -E var.c`

```
1 # 1 "var.c"
2 # 1 "<built-in>"
3 # 1 "<command-line>"
4 # 1 "var.c"
5 int main(int argc, char **argv) {
6     int v;
7     v = 10;
8     v = v + 1;
9     return argc;
10 }
```

lec09/var.c

Reminder

Example – Compilation of the Source Code to Assembler

- Using the `-S` flag, the source code can be compiled to Assembler.

```
clang -S var.c -o var.s
```

```

1  .file "var.c"
2  .text
3  .globl main
4  .align 16, 0x90
5  .type main,@function
6  main:
   # @main
7  .cfi_startproc
8  # BB#0:
9  pushq %rbp
10 .Ltmp2:
11 .cfi_def_cfa_offset 16
12 .Ltmp3:
13 .cfi_offset %rbp, -16
14 movq %rsp, %rbp
15 .Ltmp4:
16 .cfi_def_cfa_register %rbp
17 movl $0, -4(%rbp)
18 movl %edi, -8(%rbp)
19 movq %rsi, -16(%rbp)
20 movl $10, -20(%rbp)
21 movl -20(%rbp), %edi
22 addl $1, %edi
23 movl %edi, -20(%rbp)
24 movl -8(%rbp), %eax
25 popq %rbp
26 ret
27 .Ltmp5:
28 .size main, .Ltmp5-main
29 .cfi_endproc
30
31
32 .ident "FreeBSD clang version 3.4.1 (
   tags/RELEASE_34/dot1-final 208032)
   20140512"
33 .section ".note.GNU-stack","",
   @progbits

```


Undefined Behaviour

- There are some statements that can cause **undefined behavior** according to the C standard.
 - `c = (b = a + 2) - (b - 1);`
 - `j = i * i++;`
- The program may behave differently according to the used compiler, but may also not compile or may not run; or it may even crash and behave erratically or produce meaningless results.
- It may also happen if variables are used without initialization.
- **Avoid statements that may produce undefined behavior!**

Example of Undefined Behaviour

- C standard does not define the behaviour for the overflow of the integer value ([signed](#))
 - E.g., for the complement representation, the expression can be `127 + 1` of the `char` equal to `-128` (see [lec09/demo-loop_byte.c](#)).
 - Representation of integer values may depend on the architecture and can be different, e.g., when binary or inverse code is used.
- Implementation of the defined behaviour can be computationally expensive, and thus the behaviour is not defined by the standard.
- **Behaviour is not defined and depends on the compiler**, e.g. `clang` and `gcc` without/with the optimization `-O2`.

```
■ for (int i = 2147483640; i >= 0; ++i) {  
    printf("%i %x\n", i, i);  
} lec09/int_overflow-1.c
```

Without the optimization, the program prints 8 lines, for `-O2`, the program compiled by `clang` prints 9 lines and `gcc` produces infinite loop.

```
■ for (int i = 2147483640; i >= 0; i += 4) {  
    printf("%i %x\n", i, i);  
} lec09/int_overflow-2.c
```

Program compiled by `gcc` and `-O2` crashed. *Take a look to the asm code using the compiler parameter `-S`.*

Compiler Explorer

The screenshot shows the Compiler Explorer interface with three main panes:

- Source Code:** Shows a C program with a `square` function and a `main` function that calls it with the value 10.
- Preprocessor Output:** Shows the code after preprocessing, including comments and the expanded function calls.
- Assembly:** Shows the x86-64 assembly code generated by GCC 12.2, including stack frame setup, function calls, and return values.

```
1 int square(int num)
2 {
3     return num * num;
4 }
5
6 int main(void)
7 {
8     int a = square(10);
9     return 0;
10 }
11
12
```

```
1 /* <7 lines filtered>
2
3 int square(int num)
4 {
5     return num * num;
6 }
7
8 int main(void)
9 {
10     int a = square(10);
11     return 0;
12 }
13
14
```

```
1 square:
2     push    rbp
3     mov     rbp, rsp
4     mov     DWORD PTR [rbp-4], edi
5     mov     eax, DWORD PTR [rbp-4]
6     imul   eax, eax
7     pop     rbp
8     ret
9
10 main:
11     push    rbp
12     mov     rbp, rsp
13     sub     rsp, 16
14     mov     edi, 10
15     call   square
16     mov     DWORD PTR [rbp-4], eax
17     mov     eax, 0
18     leave
19     ret
```

<https://godbolt.org/z/K9r1eWqcd>

Compiler Explorer – Analysis of the Optimized Code

- Effect of the code optimization `-O2` on the resulting code that contains undefined behavior (integer overflow).

The screenshot shows the Compiler Explorer interface with three panels. The left panel shows the source code, the middle panel shows the assembly code for the default optimization level, and the right panel shows the assembly code for the `-O2` optimization level.

Source Code (C source #1):

```

1 int main(void)
2 {
3     int ret = 0;
4     for (int i = 2147483640; i >= 0; ++i) {
5         ret += i;
6     }
7     return ret;
8 }

```

Assembly Code (x86-64 gcc 12.2):

Default Optimization:

```

1 main:
2     push    rbp
3     mov     rbp, rsp
4     mov     DWORD PTR [rbp-4], 0
5     mov     DWORD PTR [rbp-8], 2147483640
6     jmp     .L2
7 .L3:
8     mov     eax, DWORD PTR [rbp-8]
9     add     DWORD PTR [rbp-4], eax
10    add     DWORD PTR [rbp-8], 1
11 .L2:
12    cmp     DWORD PTR [rbp-8], 0
13    jns     .L3
14    mov     eax, DWORD PTR [rbp-4]
15    pop     rbp
16    ret

```

-O2 Optimization:

```

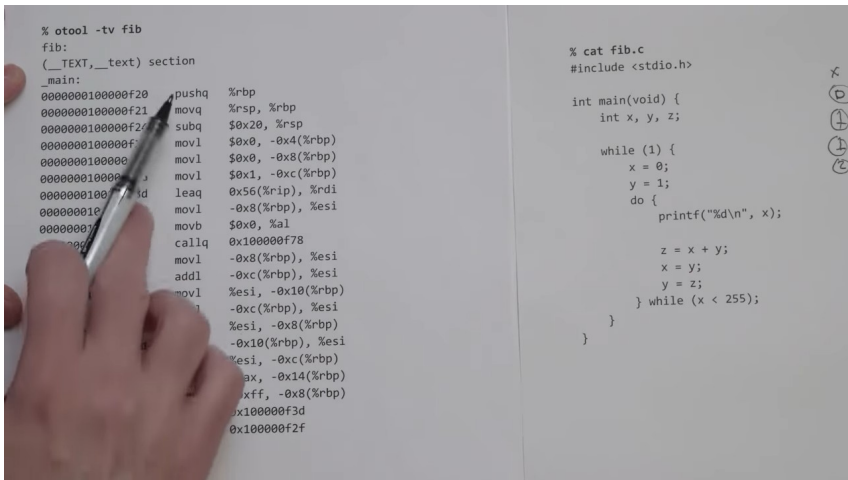
1 main:
2     .L2:
3     jmp     .L2

```

The `-O2` optimization has replaced the entire loop with a single `jmp .L2` instruction, effectively removing the loop body and the overflow. The assembly code for the default optimization shows the loop body being executed once, with the value of `ret` being updated to `2147483640`.

<https://godbolt.org/z/G3GEz4vbw>

Comparing C to Machine Code



<https://www.youtube.com/watch?v=y0yaJXpAYZQ>

Part II

Part 2 – Debugging

Debugging the Code

- Principally there are two ways of debugging: **stepping** (program animation) and **logging**.
- **Stepping** is interactive debugging that might be suitable for relatively small, less complex codes, and non real-time applications.
 - In stepping, we use **breakpoints**, **watches** to stop the program execution at certain conditions and then inspect variables and stepping next instructions.
 - In C, most of the visual interfaces uses **gdb**.
 - It might be suitable to compile the program with **debugging information**, e.g., using `-g` flag.

```
clang -g main.c -o main
```
- **Logging** can range from simple print messages to `stderr` to sophisticated **loggers**, such as `log4c`.
- We can further enjoy tools such as **valgrind** for dynamic analysis, specifically for bugs in memory access.

For more than 20 years, see <https://valgrind.org/>.

Debugging using gdb (or VS Code)

- Interactive example of debugging or watch the available examples and tutorials.

The screenshot shows a GDB session for a C program named 'hello.c'. The code in the editor is as follows:

```

1 #include <stdio.h>
2
3 int
4 main(void)
5 {
6     int i = 0;
7     printf("Hello, world!\n");
8     printf("I is %d\n", i);
9     i++;
10    printf("I is now %d\n", i);
11    return 0;
12 }
13
14
15
16
17
18
19
20
21
22
23

```

The GDB console shows the following commands and output:

```

$lld process 7418 In: main
gdb) python print('hello world')
$lld world
gdb) b main
breakpoint 2 at 0x400505: file hello.c, line 6.
gdb) b 9
breakpoint 3 at 0x40050a: file hello.c, line 9.
gdb) python print (gdb.breakpoints())
<gdb.Breakpoint object at 0x7f211240b3db>, <gdb.Breakpoint object at 0x7f211240b1e8>
gdb) python print (gdb.breakpoints()[0].location)
$in
gdb) python gdb.Breakpoint('?')
breakpoint 4 at 0x40050c: file hello.c, line 7.
gdb)

```

The video thumbnail on the right shows Greg Law speaking at CppCon 2015. The text overlay on the video reads: "GREG LAW" and "Give me fifteen minutes and I'll change your view of GDB". The website "www.CppCon.org" is also visible.

- CppCon 2015: Greg Law " Give me 15 minutes & I'll change your view of GDB."

<https://www.youtube.com/watch?v=PorfLSr3DDI>

Example of using valgrind

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int *a = malloc(2 * sizeof *a);

    for (int i = 0; i < 3; ++i) {
        a[i] = i;
    }
    for (int i = 0; i < 3; ++i) {
        printf("%d\n", a[i]);
    }
    //free(a);
    return 0;
}
```

- Try to compile the program with and w/o `-g`.
- See the **valgrind** output with and w/o calling `free()`.

```
$ clang -g mem_val.c -o mem_val
$ valgrind ./mem_val
....
==87826== Invalid write of size 4
==87826==    at 0x201999: main (mem_val.c:9)
==87826==    Address 0x5400048 is 0 bytes after
        a block of size 8 alloc'd
==87826==    at 0x4853B74: malloc (in /usr/
        local/libexec/valgrind/vgpreload_memcheck-
        amd64-freebsd.so)
==87826==    by 0x201978: main (mem_val.c:6)
==87826==
....
0
lec09/mem_val.c
```

Part III

Part 3 – Examples

Communication using Named Pipes

- Implement two applications **main** and **module** that communicates through named pipes.

`lec09/pipes/create_pipes.sh`

`lec09/pipes/prg lec09_main.c, lec09/pipes/prg-lec09-module.c`

- **module** opens pipe `/tmp/prg-lec09.pipe` for reading.
- **main** opens pipe `/tmp/prg-lec09.pipe` for writing.
- The applications communicate using simple character orienter protocol.
 - 's' – stop.
 - 'e' – enable (start).
 - 'b' – bye.
 - '1'-'5' – set sleep period to 50 ms, 100 ms, 200 ms, 500 ms, 1000 ms.
- The pipe can be opened using functions from the `prg_io_nonblock` library.

`lec09/pipes/prg_io_nonblock.h, lec09/pipes/prg_io_nonblock.c`

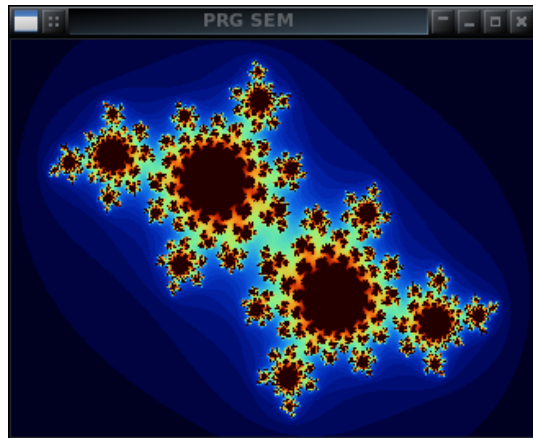
- Examine the provide code and test it.

The example is without threads.

Remote Control of Computational Application (Module) – Semetral Project

- Implement multi-thread application with separate threads for sources of asynchronous events.
 - User input from `stdin` (**keyboard**).
 - Pipe reading from the computational module.
- Use simple visualization using `sdl`.
- Implement the main program logic in the main (**boss**) thread using `event queue`.
 - The main thread reads from the queue.
 - The secondary threads (keyboard and pipe) write to the queue.
- The main thread manages output resources (**visualization, write to pipe**).
Eventually also `stdout` or even `stderr`, which is, however, not required.
- Use the example of multi-thread application from Lecture 8.

<https://cw.fel.cvut.cz/wiki/courses/b3b36prg/semestral-project/start>



Summary of the Lecture

Topics Discussed

- Program compilation.
- Undefined behaviour.
- Comments on debugging.
- Named pipes.
- Semetral project.