

## Coding Examples

Jan Faigl

Department of Computer Science  
Faculty of Electrical Engineering  
Czech Technical University in Prague

Lecture 09

### PRG – 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.

## 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 – 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 – 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 – 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:
7     # @main
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 (
33     tags/RELEASE_34/dot1-final 208032)
34     20140512"
35 .section ".note.GNU-stack","",
36     @progbits
```

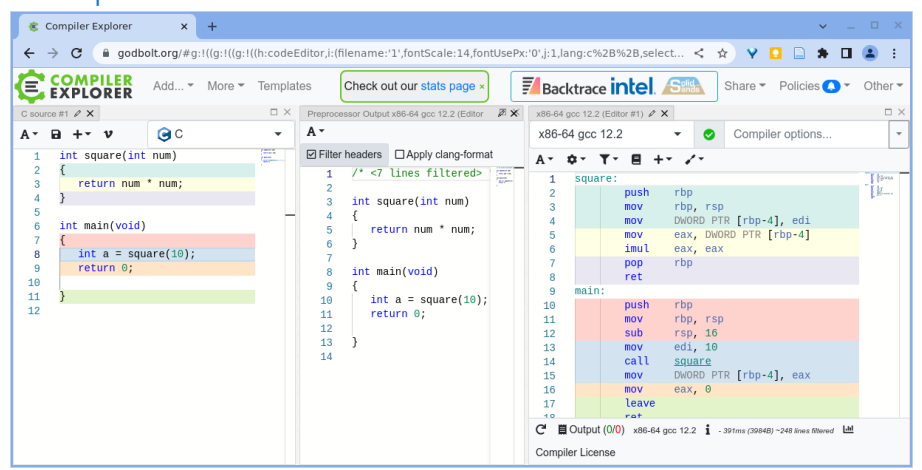
# Undefined Behaviour

- 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

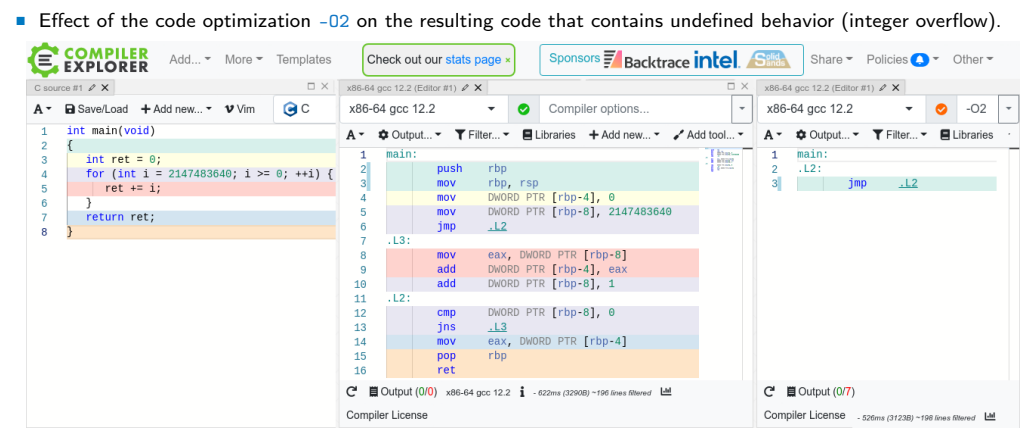
- C standard does not define the behaviour for the overflow of the integer value (**signed**)
  - E.g., for the complement representation, the expression can be `127 + 1` of the `char` equal to `-128` (see [lec09/demo-loop\\_byte.c](#)).
  - Representation of integer values may depend on the architecture 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



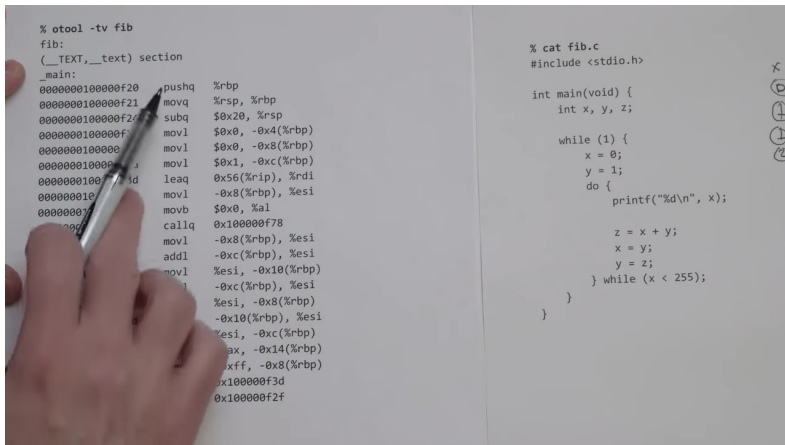
<https://godbolt.org/z/K9r1eWqcd>

# Compiler Explorer – Analysis of the Optimized Code



<https://godbolt.org/z/G3GEz4vbv>

# Comparing C to Machine Code



<https://www.youtube.com/watch?v=y0YaJXpAYZQ>

# Part II Part 2 – Debugging

# Debugging the Code

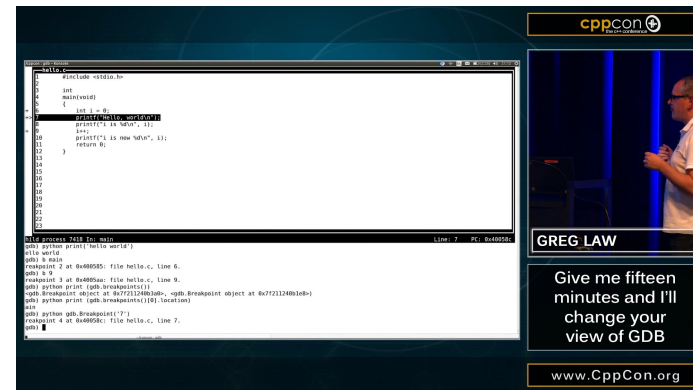
- Principally there are two ways of debugging: **stepping** (program animation) and **logging**.
- Stepping** is interactive debugging that might be suitable for relatively small, less complex codes, and non real-time applications.
  - In stepping, we use **breakpoints**, **watches** to stop the program 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.



- 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;
}
```

```
$ 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
leco9/mem_val.c
```

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

## Part III

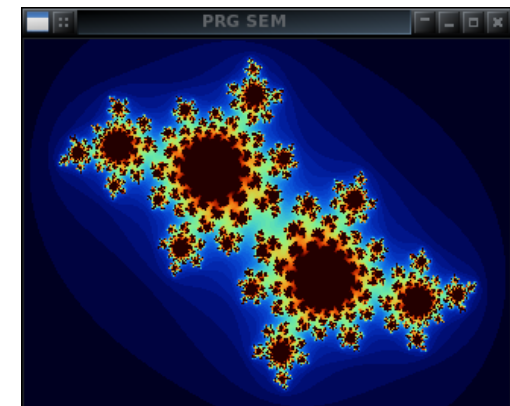
## Part 3 – Examples

## 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.
  
- Next: ANSI C, C99, C11 – differences and extensions