Testing & Debugging

PV264 Advanced Programming in C++

Nikola Beneš Jan Mrázek Vladimír Štill

Faculty of Informatics, Masaryk University

autumn 2020

Three Basic Questions of Programming

Is my program well-written?

- Will someone else be able to read (maintain, refactor) it?
- Will **I** be able to read it (tomorrow, next week, next year)?

Is my program correct?

- Does it do what it is supposed to?
- What is it actually supposed to do?

Is my program efficient?

- time, memory consumption, other resources consumption (data, energy, . . .)

Correctness

How to approach correctness?

- testing
- formal verification (automatic/semi-automatic/manual)
- code inspection
- ...

Testing

- important part of development process
- levels of testing
  - unit testing
  - integration testing
  - system testing
  - ...
- many approaches and frameworks – our focus:
  - unit testing using the **Catch2** framework
  - automated testing using the **RapidCheck** framework

Using Catch2

Catch2 (C++ Automated Test Cases in Headers)

- https://github.com/catchorg/Catch2
- advantages:
  - easy to use
  - no dependencies, one header file
  - readable test cases (support for Behaviour-Driven Development)
  - arbitrary strings as names
  - test cases divided into independent sections
  - use standard C++ operators for comparison

Using Catch2 — Sections

```
#define CATCH_CONFIG_MAIN // provide main()
#include "catch.hpp"

#include <vector>

TEST_CASE("Vector is initialised as empty") {
    std::vector<int> vec;
    REQUIRE(vec.size() == 0);
}
```

Using Catch2 – Sections

```
TEST_CASE("Vector size and capacity") {
    std::vector<int> vec;
    vec.push_back(1);
    vec.push_back(2);
    auto size = vec.size();
    REQUIRE(size == 2);
    SECTION("push_back increases size") {
        vec.push_back(3);
        REQUIRE(vec.size() > size);
    }
    SECTION("erase decreases size") {
        vec.erase(vec.begin());
        REQUIRE(vec.size() < size);
    }
}
```

Using Catch2 – Sections

- for each (leaf) `SECTION` the `TEST_CASE` is executed from the start
- alternative to the traditional text fixture approach (`setup`/`teardown`)
  - Catch2 also supports fixtures, see docs
- `SECTION`s can be arbitrarily nested
  - failure in parent section prevents nested sections from running

- BDD (Behaviour-Driven Development)
- `SCENARIO`, `GIVEN`, `WHEN`, `THEN`

```
SCENARIO("Adding an element to a vector") {
    GIVEN("A vector with no elements") {
        std::vector<int> vec;
        WHEN("an element is added via push_back") {
            vec.push_back(0);
            THEN("the size becomes 1") {
                REQUIRE(vec.size() == 1); } } } }
```

Using Catch2 – Asserts & Logs

`REQUIRE`, `CHECK`, `REQUIRE_FALSE`, `CHECK_FALSE`

- assert condition (`CHECK`: execution continues even after failure)

`REQUIRE_THROWS`, `REQUIRE_NOTHROW`, `CHECK_THROWS`, ...

- assert that an expression throws/does not throw an expression

`INFO`, `WARN`, `FAIL`

- logging

`CAPTURE`

- log the value of a variable

Using Catch2 – Useful Information

- command-line parameters
  - which test(s) to run
  - output format (jUnit, XML, . . .)
- configuration via macros, own `main()`

Using Catch2 – Useful Information

- command-line parameters
  - which test(s) to run
  - output format (jUnit, XML, ...)
- configuration via macros, own `main()`

Recommended practice

- one main source file with nothing but the main function (possibly generated by Catch2)

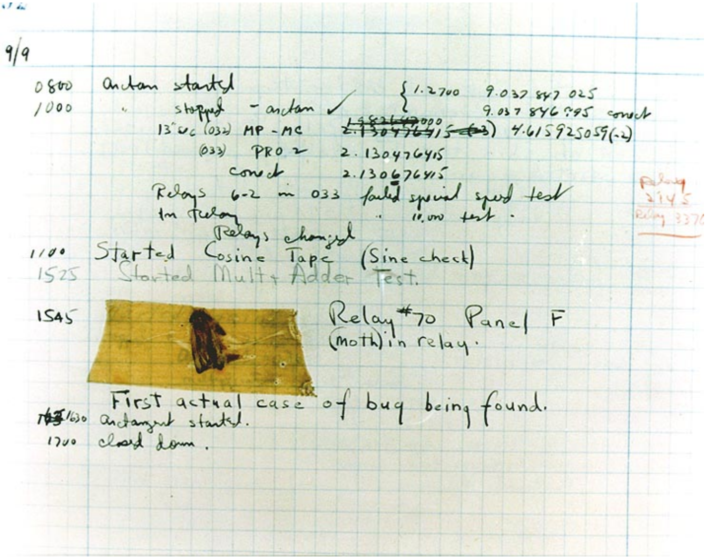
```
#define CATCH_CONFIG_MAIN
#include "catch.hpp"
// end of file
```

- other source files for tests

Using RapidCheck

RapidCheck

- https://github.com/emil-e/rapidcheck
- property-based testing
- similar to Haskell's QuickCheck, Python's hypothesis
- automatically generated test cases
- counterexample shrinking


9/9
0800 Antan started
1000 stopped - antan ✓
Relays 6-2 in 033 failed special speed test
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Mult + Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
1630 Antangent started.
1700 closed down.

Debugging

Tests fail, now what?

- tracing ("`printf` debugging")
- logging
- using debuggers
- using other useful tools

Debugging

Tests fail, now what?

- tracing (“`printf` debugging”)
- logging
- using debuggers
- using other useful tools

Recommendation

- try to find a minimal example where the problem occurs
  - “code bisection”
- bugs are sometimes caused by bad memory management
  - don’t forget about `valgrind` and similar tools
- to be able to employ debuggers:
  - compile without optimisation
  - compile with debug information (`-g`)

Debuggers

Typical Debugger Functions

- pause at specified breakpoints
  - line of code, condition, exception thrown/caught, signals, ...
- evaluate expressions
- step through program
- (modify program state)

Our Focus

- `gdb` (The GNU Debugger)
  - command-line tool
  - many graphical front-ends

Using gdb

Basic commands:

- `help`
- `run` – start the debugged program
- `list` – list specified function or line
- `break` – set breakpoint
- `catch` – set catchpoint (exception breakpoint)
- `info` – show information about the debugged program
  - `info args`, `info registers`, `info breakpoints`, ...
- `step` – step program, steps into functions
- `next` – step program, steps over function calls
- `stepi`, `nexti` – step by instructions, not lines of code
- `print` – evaluate expression
- `examine` – display contents of memory address
- `disp` – evaluate expression each time the program stops
- `continue` – continue running (after breakpoint)
- `kill` – stop execution of the program

Using gdb

Stack commands:

- `backtrace` – print backtrace of stack frames
- `up`, `down`, `frame`, `select-frame` – select stack frame
- `finish` – run until current stack frame returns
- `info locals`, `info frame`

Executing code at runtime:

- `set var = value` – change the value of a variable
- `call func()` – call a function

Watchpoints:

- `watch var` – watch changes (writes) of a variable
- `rwatch var` – watch reads of a variable
- `awatch var` – watch both reads and writes

gdb front-ends

cgdb

- terminal-based front-end for gdb (uses the curses library)
- displays the source code above the gdb session
- https://cgdb.github.io/
- module add cgdb-0.6.6 on faculty computers

Other front-ends: see https://sourceware.org/gdb/wiki/GDB%20Front%20Ends

Assembler

Assembly Language (symbolic machine code)

- low-level; closest to machine code
- commands – machine code instructions

Why do we want to know about it?

- debugging
- computer security
- examine optimisation done by compiler
- sometimes it is good to know what's "under the hood"

Our focus here: brief overview; reading assembly, not writing it

Assembler – Tools

Disassemble

- `clang++ -S`, `g++ -S`, etc.
- `gdb`
  - `disassemble`
  - `x/10i` address (such as `$rip`)
  - (`print`, `disp`)
  - `set disassemble-next-line on`
- `objdump -d`

Show raw bytes

- `hexdump -C`
- `xxd`

Compiler explorer: `https://godbolt.org`

Assembler Notation

Intel

- operands in order *dest*, *src*
  - `mov rax, rbx` moves *from* `rbx` *to* `rax`
  - `add rax, 0x1f` adds `0x1f` *to* `rax`
- memory indexing `[base + index*scale + disp]`
  - `mov eax, [rbx + rcx*4 + 0x10]`

AT&T

- operands in order *src*, *dest*
  - `mov %rbx, %rax`
  - `add $0x1f, %rax`
- memory indexing `disp(base, index, scale)`
  - `movl 0x10(%rbx, %rcx, 4), %eax`
- size indicated in the instruction mnemonic
  - `movb`, `movw`, `movl`, `movq` (1, 2, 4, and 8 bytes)
- immediate values with $, registers with %

Assembler notation

How to use the Intel syntax?

- `clang++ -S -masm=intel`
- `objdump -d -M intel`
- `gdb`
  - `set disassembly-flavor intel`

x86(-64) Architecture

Registers

- instruction pointer: `ip` (16 bit), `eip` (32 bit), `rip` (64 bit)
- stack pointer: `sp` (16 bit), `esp` (32 bit), `rsp` (64 bit)
- general purpose: `ax`, `bx`, `cx`, `dx` (`eax`, `rax`, . . .)
  - lower 8 bits: `al`, `bl`, `cl`, `dl`
- source/destination: `si`, `di` (`esi`, `rsi`, . . .)
- stack frame base pointer: `bp` (`ebp`, `rbp`)
- 64 bit general purpose: `r8`, `r9`, . . . , `r15`
  - low 32 bits: `r8d`, . . .
  - low 16 bits: `r8w`, . . .
  - low 8 bits: `r8b`, . . .
- floating-point (80 bit) registers `st0`, . . . , `st7`
- XMM 128 bit registers `xmm0`, . . . , `xmm15`

x86(-64) Architecture

Stack

- memory area given by OS to programs
- LIFO data structure; x86 stack grows towards lower addresses
- `esp` (`rsp`) points to the top of the stack
- main use: return address, function arguments, local variables, temporary storage

`PUSH` value

- decrements `esp` (`rsp`) and then stores the given value at the memory address given by (the new value of) `esp` (`rsp`)

`POP` register

- copies the value from the memory address given by `esp` (`rsp`) into the given register and then increments `esp` (`rsp`)

x86(-64) Architecture

How do function calls work?

- parameters are stored somewhere (see below)
- `call` address
  - push address of next instruction on stack
  - jump to address
- `ret` (return from function)
  - pops address from stack and jumps to it

Calling conventions

- 32bit: many different possibilities
  - *cdecl*: arguments passed on the stack in reverse order
- 64bit: two main approaches (Microsoft x64, System V AMD64)
  - both use registers to pass (some of) the arguments
  - registers used also depend on type (integers, floats) of arguments

x86(-64) Architecture

Function frames (standard entry/exit sequence)

- at the beginning of a function:

  ```
  push rbp
  mov rbp, rsp
  sub rsp, 0x10
  ```
  (allocate 16 bytes on stack for local variables)
- `rbp` is the base frame pointer
  - local values referenced as `[rbp + 0x08]`, ...
  - note that `[rbp]` holds the value of the previous `rbp`
- at the end of a function:

  ```
  mov rsp, rbp
  pop rbp
  ```

Note: Optimisations (frame pointer omission optimisation) may eliminate this. (`-f[no-]omit-frame-pointer`)

x86(-64) Instructions

Move instruction

- `MOV` – copy value from *src* to *dest*

Arithmetic and logic instructions

- `ADD`, `SUB`, `MUL`, ...
- `AND`, `OR`, `XOR`, ...

Test instructions

- `CMP` – performs `SUB`; does not save the result, only sets *flags*
- `TEST` – similar to `CMP`, performs `AND`

Jump instructions

- `JMP` – unconditional jump
- `Jxx` – conditional jump, reacts to *flags*
  - `JZ` – jump if zero
  - `JBE` – jump if below or equal
  - ...

Optimisations

What can the compiler optimise for us?

Optimisations

What can the compiler optimise for us?

- speed
  - rearranging memory accesses
  - inline functions
  - tail recursion (sometimes even non-tail recursion)
  - loop unrolling
- space
  - collapse common code
- *obvious*
  - constant propagation

. . . and much more