Testing & Debugging

PB173 Programming in Modern C++

Nikola Beneš, Vladimír Štill, Jiří Weiser

Faculty of Informatics, Masaryk University

spring 2016

Three Basic Questions

Readability, Correctness, Efficiency

Three basic questions you should ask yourselves when programming:

- is my program well-written?
- **is my program correct?**
- is my program efficient?

Three Basic Questions

Readability, Correctness, Efficiency

Three basic questions you should ask yourselves when programming:

- is my program well-written?
- **is my program correct?**
- is my program efficient?

How to approach correctness?

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

Testing

- an important part of the development process
- levels of testing
  - unit testing
  - integration testing
  - system testing
- many approaches and frameworks
  - our focus here: CATCH framework

CATCH (C++ Automated Test Cases in Headers)

- `https://github.com/philsquared/Catch`
- advantages:
  - easy to use
  - no dependencies, just one header file
  - readable test cases
  - arbitrary strings as names
  - test cases divided into independent sections
  - use standard C++ operators for comparison

Testing Using CATCH

Simple Example

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

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

Testing Using CATCH

Using Sections

```
TEST_CASE( "Vector size and capacity" ) {
    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 );
    }
}
```

Testing Using CATCH

Using Sections

- for each `SECTION` the `TEST_CASE` is executed from the start
- alternative to the traditional `setup()`/`teardown()` approach
  - CATCH also supports test fixtures, see documentation
- `SECTION`s can be nested to arbitrary depth
  - failure in parent section prevents nested sections from running

Testing Using CATCH

Using Sections

- for each `SECTION` the `TEST_CASE` is executed from the start
- alternative to the traditional `setup()`/`teardown()` approach
  - CATCH also supports test fixtures, see documentation
- `SECTION`s can be nested to arbitrary depth
  - failure in parent section prevents nested sections from running

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

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

Testing Using CATCH

Asserts & Logs

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

- assert condition
- `CHECK`: execution continues even after assertion failure

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

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

`INFO`, `WARN`, `FAIL`

- logging

`CAPTURE`

- log the value of a variable

Testing Using CATCH

Other Useful Information

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

Testing Using CATCH

Other Useful Information

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

Recommended practice

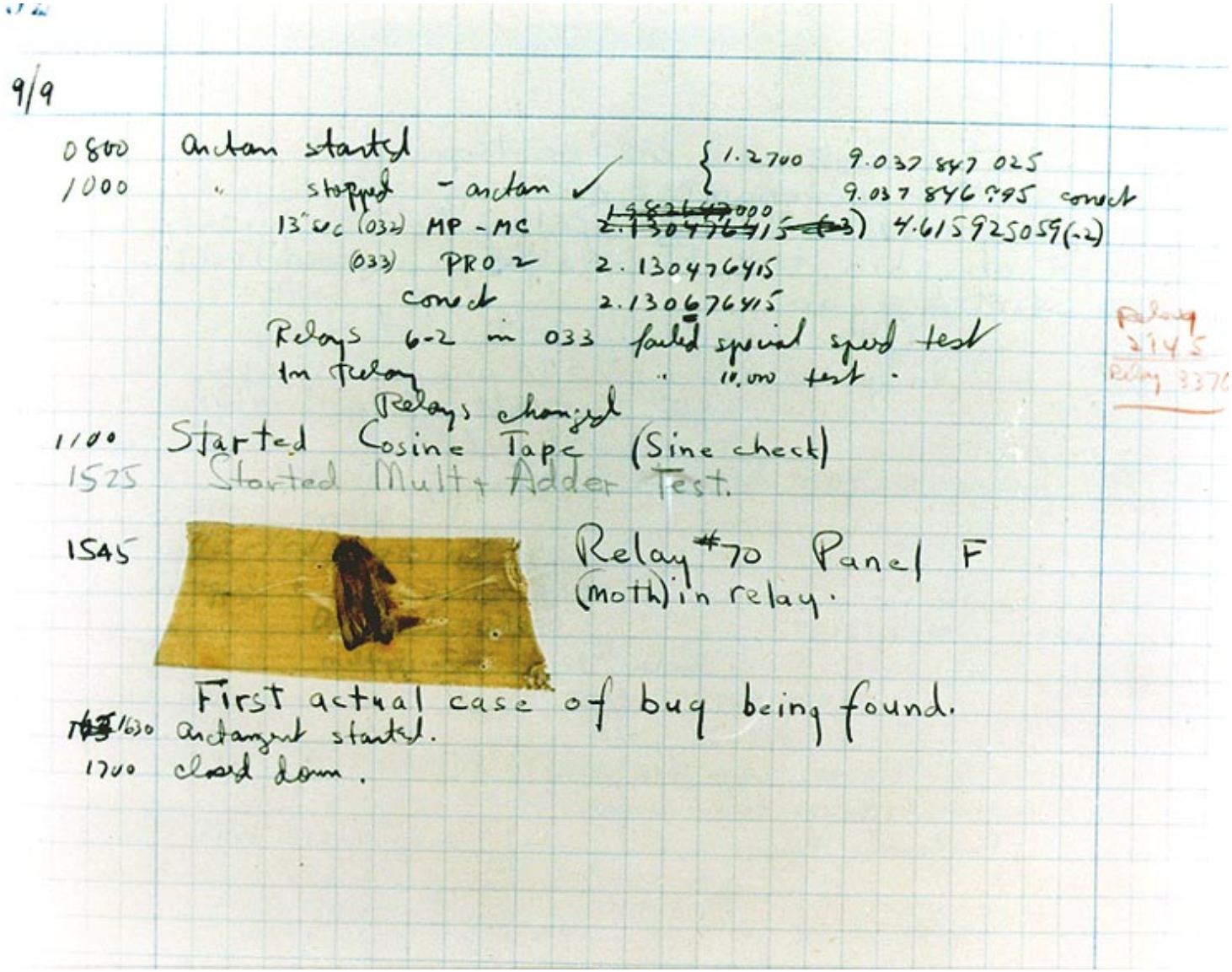
- one main source file with nothing but the `main()` function (possibly generated by CATCH)

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

- other source files for tests

Debugging

Eliminating Bugs



Debugging

Tests Fail, Now What?

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

Debugging

Tests Fail, Now What?

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

Recommendation:

- try to find a minimal example where problem occurs
  - "code bisection"
- bugs are sometimes caused by bad memory management
  - use `valgrind` or similar tools
- to be able to employ debuggers:
  - compile without optimisation
  - compile with debug information (`-g`)

Debuggers

Typical Functions

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

Our Focus Today

- gdb (The GNU Debugger)
  - command-line tool
  - has 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`

Exercise

Exercise no. 1

- source codes with errors in the study materials
- use `gdb` / `cgdb`

Exercise no. 2

- write tests for your `vector` implementation (from the previous seminar)
- use the CATCH framework