# LECTURE 14

# TOOLS FOR PROGRAM CORRECTNESS

Today:

1. Documentation
2. Testing
3. Static analysis
4. Dynamic analysis

- Each uncovers bugs
- For each, there are useful tools (compilers can help!)

# DOCUMENTATION

# Documentation is GOOD

- Allows others to understand your code
- Allows yourself (in a few weeks) to understand your own code
- Helps make your thought process and assumptions explicit

# Types of documentation

- Reference manuals
- Tutorials
- Questions and answers (Q&A)

# Reference manuals

- Authoritative source of information

  If the code does not do what the manual says, then the code is wrong.

- Must be complete

- Must use precise language

  Even at the cost of legibility

- Examples: “`man`” pages, C standard, IEEE-754 specifications

# Tutorials

- Beginner-friendly
- Usually emphasize getting things to work quickly

  even at the cost of completeness
- Good tutorials do not sacrifice accuracy (but many bad ones do)
- Examples: various books (K&R C, Think Python) and intro material

# Questions and answers (Q&A)

- Prioritize quick answers to frequently asked questions
- Not exhaustive
- Examples: Stack Overflow, various FAQs

When reading documentation:

- as a beginner, aim for tutorials and Q&As
- as you become an expert, you need a reference manual.

When writing documentation:

- ideally, you write all three!

# Automated documentation

Automated documentation systems

- read and parse source code
- find functions (methods, classes, …)
- create a (PDF or webpage) document containing function signatures
- specially-formatted comments in the source code are copied into the documentation

  along with the corresponding function signatures

# Doxygen

Search or go to...

libeigen › eigen

Project

- eigen
- Manage
- Plan
- Code
- Build
- Deploy
- Operate
- Monitor
- Analyze

```cpp
/** This is the "in place" version of transpose(): it replaces \c *this by its own transpose.
  * Thus, doing
  * \code
  * m.transposeInPlace();
  * \endcode
  * has the same effect on m as doing
  * \code
  * m = m.transpose().eval();
  * \endcode
  * and is faster and also safer because in the latter line of code, forgetting the eval() results
  * in a bug caused by \ref TopicAliasing "aliasing".
  *
  * Notice however that this method is only useful if you want to replace a matrix by its own transpose.
  * If you just need the transpose of a matrix, use transpose().
  *
  * \note if the matrix is not square, then \c *this must be a resizable matrix.
  * This excludes (non-square) fixed-size matrices, block-expressions and maps.
  *
  * \sa transpose(), adjoint(), adjointInPlace() */
template<typename Derived>
EIGEN_DEVICE_FUNC inline void DenseBase<Derived>::transposeInPlace()
{
  eigen_assert((rows() == cols() || (RowsAtCompileTime == Dynamic && ColsAtCompileTime == Dynamic))
               && "transposeInPlace() called on a non-square non-resizable matrix");
  internal::inplace_transpose_selector<Derived>::run(derived());
}
```

Table of contents



## ◆ transposeInPlace()

template<typename Derived >

void **Eigen::DenseBase**< Derived >::transposeInPlace

This is the "in place" version of **transpose()**: it replaces `*this` by its own transpose. Thus, doing

```
m.transposeInPlace();
```

has the same effect on m as doing

```
m = m.transpose().eval();
```

and is faster and also safer because in the latter line of code, forgetting the **eval()** results in a bug caused by **aliasing**.

Notice however that this method is only useful if you want to replace a matrix by its own transpose. If you just need the transpose of a matrix, use **transpose()**.

**Note**

if the matrix is not square, then `*this` must be a resizable matrix. This excludes (non-square) fixed-size matrices, block-expressions and maps.

**See also**

**transpose()**, adjoint(), adjointInPlace()

# Python docstrings

```python
def complex(real=0.0, imag=0.0):
    """Form a complex number.

    Keyword arguments:
    real -- the real part (default 0.0)
    imag -- the imaginary part (default 0.0)
    """
    if imag == 0.0 and real == 0.0:
        return complex_zero
    ...
```

# Automated documentation systems

- General:
  - doxygen
  - sphinx

- Python-specific:
  - pdoc
  - PyDoc
  - pydoctor

Note: Some projects choose to **not** use automated documentation.

# TESTING

```c
/*
  This functions returns:
    5 if one or both of its arguments are 5
    0 otherwise
*/
int five_if_some_five(int a, int b)
{
    if (a != 5)
        a = 0;

    if (b != 5)
        b = 0;

    return a | b;
}
```

```c
int tests()
{
    int errors = 0;

    errors += (five_if_some_five(100, 100) != 0);
    errors += (five_if_some_five(100,   5) != 5);

    return errors;
}
```

# Test coverage

- line coverage:

  is every line of code covered by some test case?

- branch coverage:

  for every conditional branch, is there a test covering each of the two possibilities (taking the branch or not taking it)?

```
clang -Wall -O3 --coverage -c -o five.o five.c
clang -Wall -O3 --coverage -o test test.c five.o
```

```
./test
```

```
Errors: 0
```

```
gcov five.c
```

```
File 'five.c'
Lines executed:100.00% of 4
Creating 'five.c.gcov'

Lines executed:100.00% of 4
```

```
gcov -b five.c
```

```
File 'five.c'
Lines executed:100.00% of 4
Branches executed:100.00% of 4
Taken at least once:75.00% of 4
No calls
Creating 'five.c.gcov'

Lines executed:100.00% of 4
```

```
function five_if_some_five called 2 returned 100% blocks executed 100%
        2:   22:int five_if_some_five(int a, int b)
        -:   23:{
        2:   24:        if (a != 5)
branch  0 taken 100% (fallthrough)
branch  1 taken 0%
        -:   25:                a = 0;
        -:   26:
        2:   27:        if (b != 5)
branch  0 taken 50% (fallthrough)
branch  1 taken 50%
        -:   28:                b = 0;
        -:   29:
        2:   30:        return a | b;
        -:   31:}
```

# Line coverage vs. branch coverage

```c
/*
  This functions returns:
    5 if one or both of its arguments are 5
    0 otherwise
*/
int five_if_some_five(int a, int b)
{
    if (a != 5)
        a = 0;

    if (b != 5)
        b = 0;

    return a | b;
}
```

```c
int tests()
{
    int errors = 0;

    errors += (five_if_some_five(100, 100) != 0);

    return errors;
}
```

Line coverage: 100%

Branch coverage: 50%

# How does it work?

```
clang -Wall -O3 --coverage -c -o five.o five.c
```

```
/*
  This functions returns:
    5 if one or both of its arguments are 5
    0 otherwise
*/
int five_if_some_five(int a, int b)
{
    line_covered(4);
    if (a != 5) {                        // line 4
        branch_covered(4, 1);
        line_covered(5);                 // line 5
        a = 0;
    } else {
        branch_covered(4, 0);
    }

    line_covered(7);
    if (b != 5) {                       // line 7
        branch_covered(7, 1);
        line_covered(8);
        b = 0;                          // line 8
    } else {
        branch_covered(7, 0);
    }

    line_covered(10);
    return a | b;                       // line 10
}
```

# Limitations of test coverage measures (1)

```c
/*
  This functions returns:
    5 if one or both of its arguments are 5
    0 otherwise
*/
int WRONG_five_if_some_five(int a, int b)
{
    return a | b;
}
```

```c
int test()
{
    return (WRONG_five_if_some_five(0, 5) != 5);
}
```

Line coverage: 100%

Branch coverage: 100%

# Limitations of test coverage measures (2)

```c
/*
  This functions returns:
    5 if one or both of its arguments are 5
    0 otherwise
*/
int WRONG_five_if_some_five(int a, int b)
{
    if (a != 5)
        a = 0;

    if (b != 5)
        b = 0;

    return a + b;
}
```

```c
int tests()
{
    int errors;

    errors += (WRONG_five_if_some_five(100, 100) != 0);
    errors += (WRONG_five_if_some_five(  5, 100) != 5);
    errors += (WRONG_five_if_some_five(100,   5) != 5);

    return errors;
}
```

Line coverage: 100%

Branch coverage: 100%

# Assertions

- Assertions are used to document (and check) assumptions made in the code.
- An assertion failure
  - should correspond to a bug in your code,
  - triggers an immediate crash (`abort()`) of your program.

```c
#include <assert.h>

int gcd(int a, int b)
{
    if (a < b) {
        int r = a;
        a = b;
        b = r;
    }

    while (b != 0) {
        assert(a >= b);     // <---- this should always be true

        int r = a % b;
        a = b;
        b = r;
    }

    return a;
}
```

# Disabling assertions

```
clang -D NDEBUG -Wall -O3 -o main main.c
```

(equivalent to

```
#define NDEBUG
```

at the beginning of every file)

# Error vs assertion failure

- an error happens when, for external reasons, your program cannot run
  - examples: out of memory, file cannot be read, network unreachable

- an assertion fails if a fundamental assumption in your code is violated
  - indicates a bug in your code

# STATIC ANALYSIS

- **Static** analysis operates on the source code

  (before any assembly or executable code is produced)
- Compilers do advanced case analysis on the code

  (in order to produce faster code)
- The same analysis can be used to find (potential) bugs

- Not an exact science
  - Relies on heuristics to detect hazardous code
  - Suffers from false negatives and false positives

# Clang's static analyzer

If you use a `Makefile`, run

```
scan-build make
```

> result

# Python linters

- A “linter” is a static analyzer
- Typically, linters enforce a specific coding style

Examples:

- Pylint
- flake8
- mypy (adds static type checking)

```python
def fib(n):
    a, b = 0, 1
    while a < n:
        yield a
        a, b = b, a+b
```

```python
def fib(n: int) -> Iterator[int]:
    a, b = 0, 1
    while a < n:
        yield a
        a, b = b, a+b
```

# DYNAMIC ANALYSIS

- **Dynamic** analysis operates on the running executable (during testing)
- by adding runtime checks
- can find more bugs than static analysis…
- … but only for those bugs are triggered by some test!

# Sanitizers

With **sanitizers**, runtime checks are added by the **compiler**.

# UBSan

- The "undefined behavior sanitizer" detects many types of undefined behavior (at runtime)
- triggers an immediate crash (with an explanation message)
- Pass "`-fsanitize=undefined`" to `gcc` or `clang`

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

int f(int a, int b)
{
    printf("a = %d, b = %d\n", a, b);

    int r = a / b;

    printf("We survived!\n");

    return r;
}

int main(int argc, char **argv)
{
    int i = (argc < 2) ? 5 : strtol(argv[1], NULL, 0);
    int r = f(10, i);
    printf("r = %d\n", r);
}
```

## Without UBSan:

```
gcc -O3 -o timetravel timetravel.c
./timetravel 0
```

```
a = 10, b = 0
We survived!
Floating point exception (core dumped)
```

## With UBSan:

```
clang -O3 -fsanitize=undefined -o timetravel timetravel.c
./timetravel 0
```

```
a = 10, b = 0
timetravel.c:8:12: runtime error: division by zero
SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior timetravel.c:8:12 in
UndefinedBehaviorSanitizer:DEADLYSIGNAL
==3245281==ERROR: UndefinedBehaviorSanitizer: FPE on unknown address 0x00000042b43d (pc 0x00000042b43d bp 0x7ffdb30690f0 sp
    #0 0x42b43d in f /home/poirrier/courses/softeng/code/std/timetravel.c:8:12
    #1 0x42b43d in main /home/poirrier/courses/softeng/code/std/timetravel.c:18:10
    #2 0x7fd43af4db89 in __libc_start_call_main (/lib64/libc.so.6+0x27b89) (BuildId: 3ebe8d97a0ed3e1f13476a02665c5a9442adcd
    #3 0x7fd43af4dc4a in __libc_start_main@GLIBC_2.2.5 (/lib64/libc.so.6+0x27c4a) (BuildId: 3ebe8d97a0ed3e1f13476a02665c5a9
    #4 0x4033d4 in _start (/home/poirrier/courses/softeng/code/std/timetravel+0x4033d4) (BuildId: a42ae4bf9188c9d93ff828ccd

UndefinedBehaviorSanitizer can not provide additional info.
SUMMARY: UndefinedBehaviorSanitizer: FPE /home/poirrier/courses/softeng/code/std/timetravel.c:8:12 in f
==3245281==ABORTING
```

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

static int (*function_pointer) ();

static int erase_all_files()
{
    return printf("Deleting all your files\n");
}

void this_function_is_never_called()
{
    function_pointer = erase_all_files;
}

int main() {
    return (*function_pointer) ();
}
```

```
./ub
```

```
Deleting all your files
```

## Pros

- Fixes the anything-can-happen problem with undefined behavior

  (we get a crash with an explanation instead)
- No false positives

## Cons

- Not all types of undefined behavior detected (most are)
- Does not always stop the compiler from exploiting undefined behavior
- Overhead (~3x slowdown)
- Needs good tests

# ASan

- The “address sanitizer” detects many types memory access errors (at runtime)
- Separate from UBSan because it uses different mechanisms
- triggers an immediate crash (with an explanation message)
- Pass “`-fsanitize=address`” to `gcc` or `clang`

```c
#include <stdio.h>

char *f()
{
    char buffer[16];

    snprintf(buffer, sizeof(buffer), "Hello");

    return buffer;
}

int main()
{
    char *s = f();

    printf("Here is the return value of f():\n");
    printf("%s\n", s);
    return 0;
}
```

```
clang -O3 -fsanitize=address -o bug bug.c
./bug
```

```
Here is the return value of f():
=================================================================
==3245688==ERROR: AddressSanitizer: stack-use-after-scope on address 0x7f604b800020 at pc 0x00000043cd41 bp 0x7ffd5bb0da70
READ of size 1 at 0x7f604b800020 thread T0
    #0 0x43cd40 in puts (/home/poirrier/courses/softeng/code/std/bug+0x43cd40) (BuildId: fd60803d545d3b62b6353b1498d16e17a
    #1 0x4f39d1 in main (/home/poirrier/courses/softeng/code/std/bug+0x4f39d1) (BuildId: fd60803d545d3b62b6353b1498d16e17a
    #2 0x7f604d60db89 in __libc_start_call_main (/lib64/libc.so.6+0x27b89) (BuildId: 3ebe8d97a0ed3e1f13476a02665c5a9442adc
    #3 0x7f604d60dc4a in __libc_start_main@GLIBC_2.2.5 (/lib64/libc.so.6+0x27c4a) (BuildId: 3ebe8d97a0ed3e1f13476a02665c5a
    #4 0x41d324 in _start (/home/poirrier/courses/softeng/code/std/bug+0x41d324) (BuildId: fd60803d545d3b62b6353b1498d16e1

Address 0x7f604b800020 is located in stack of thread T0 at offset 32 in frame
    #0 0x4f393f in main (/home/poirrier/courses/softeng/code/std/bug+0x4f393f) (BuildId: fd60803d545d3b62b6353b1498d16e17a

  This frame has 1 object(s):
    [32, 48) 'buffer.i' <== Memory access at offset 32 is inside this variable

. . .
```

# ASan detects (1)

- Out-of-bounds accesses to heap, stack and globals

```
int a[10];

printf("%d\n", a[20]);
```

- Use-after-free

```
free(pointer);

printf("%d\n", *pointer);
```

# ASan detects (2)

- Use-after-return

```
int *f()
{
    int a[10];
    return a;
}

void g()
{
    int *pointer = f();
    printf("%d\n", pointer[0]);
}
```

- Use-after-scope

```
void g()
{
    int *pointer;

    if (1) {
        int a[10];
        pointer = a;
    }

    printf("%d\n", pointer[0]);
}
```

# ASan detects (3)

- Double-free, invalid free

```c
void *other_pointer = pointer;

free(pointer);
free(other_pointer);
```

```c
int a[10];
free(a);
```

- Memory leaks

```c
void f()
{
    void *ptr = malloc(10);
}
```

## Pros

- Detects most memory issues
- No false positives

## Cons

- Not every memory issue detected (many are)
- Overhead (~2x slowdown)
- Needs good tests

# Valgrind

- Valgrind adds runtime checks on already-compiled executable.
- It is a hybrid interpreter / JIT compiler for machine code.
- It adds checks around all memory accesses.
  - Detects uses of invalid pointers (incl. uninitialized memory)
  - Detects memory leaks (at exit)

# Valgrind requires compiling with the “-ggdb” option (gcc / clang)

```
valgrind --leak-check=full ./truthtable all ../data/parse_04.cnf
```

```
==3244248== Memcheck, a memory error detector
==3244248== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==3244248== Using Valgrind-3.21.0 and LibVEX; rerun with -h for copyright info
==3244248== Command: ./truthtable all ../data/parse_04.cnf
==3244248==
../data/parse_04.cnf: -3 is out of bounds (n = 2)
==3244248==
==3244248== HEAP SUMMARY:
==3244248==     in use at exit: 262,144 bytes in 1 blocks
==3244248==   total heap usage: 3 allocs, 2 frees, 266,712 bytes allocated
==3244248==
==3244248== 262,144 bytes in 1 blocks are definitely lost in loss record 1 of 1
==3244248==    at 0x484182F: malloc (vg_replace_malloc.c:431)
==3244248==    by 0x4023EF: di_push (parse.c:94)
==3244248==    by 0x4023EF: dimacs_parse_f (parse.c:215)
==3244248==    by 0x402541: dimacs_parse (parse.c:268)
==3244248==    by 0x401201: run (main.c:12)
==3244248==    by 0x401201: main (main.c:62)
==3244248==
==3244248== LEAK SUMMARY:
==3244248==    definitely lost: 262,144 bytes in 1 blocks
==3244248==    indirectly lost: 0 bytes in 0 blocks
==3244248==      possibly lost: 0 bytes in 0 blocks
==3244248==    still reachable: 0 bytes in 0 blocks
==3244248==         suppressed: 0 bytes in 0 blocks
==3244248==
==3244248== For lists of detected and suppressed errors, rerun with: -s
==3244248== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

## Pros

- Detects almost all memory issues (that happen at runtime)

## Cons

- Large overhead (~10x slowdown)
- Needs good tests

# FUZZING

# We need good tests

- Dynamic analysis tools are useful
- but only if we have good test cases
- and enough of them
- $\Rightarrow$ How do we generate good tests?

On a basic level, a fuzzer proceeds as follows:

1. take a (mostly valid) example input file
2. run the tested program with that input file
3. check for crashes
4. modify the input file:
    - random modifications
    - truncations, duplications
    - mergers with other example input files
5. go back to 2

# Advanced fuzzers

- use test coverage techniques

  to determine which input files are “interesting”,

  in that they cover previously-uncovered source code

- use static analysis techniques

  to determine input file modifications that could trigger specific code branches

# AFL++

- open source project (https://aflplus.plus/)
- takes as an input a directory with many (mostly valid) example input files
- generates modified input files that (try to) yield crashes

```
afl-fuzz -i directory/with/example/inputs/ -o directory/for/crash/files/ -- ./executable @@
```

```
poirrier@dev:~/courses/softeng/hw02/glpk-5.0/afl_work

      american fuzzy lop ++4.09a {default} (../examples/glpsol) [fast]
┌─ process timing ─────────────────────────────────────┬─ overall results ────┐
│        run time : 0 days, 0 hrs, 0 min, 17 sec      │  cycles done : 2     │
│   last new find : 0 days, 0 hrs, 0 min, 6 sec       │ corpus count : 93    │
│last saved crash : none seen yet                     │saved crashes : 0     │
│ last saved hang : none seen yet                     │  saved hangs : 0     │
├─ cycle progress ─────────────────────┬─ map coverage┴──────────────────────┤
│  now processing : 54.22 (58.1%)      │    map density : 0.25% / 0.28%      │
│  runs timed out : 0 (0.00%)          │ count coverage : 3.94 bits/tuple    │
├─ stage progress ─────────────────────┼─ findings in depth ─────────────────┤
│  now trying : splice 5               │ favored items : 6 (6.45%)           │
│ stage execs : 56/57 (98.25%)         │  new edges on : 8 (8.60%)           │
│ total execs : 73.4k                  │ total crashes : 0 (0 saved)         │
│  exec speed : 4212/sec               │  total tmouts : 0 (0 saved)         │
├─ fuzzing strategy yields ────────────┴───────────────┬─ item geometry ───────┤
│   bit flips : disabled (default, enable with -D)   │    levels : 4         │
│  byte flips : disabled (default, enable with -D)   │   pending : 28        │
│ arithmetics : disabled (default, enable with -D)   │  pend fav : 0         │
│  known ints : disabled (default, enable with -D)   │ own finds : 37        │
│  dictionary : n/a                                  │  imported : 0         │
│havoc/splice : 29/49.2k, 8/23.5k                    │ stability : 100.00%   │
│py/custom/rq : unused, unused, unused, unused       ├───────────────────────┘
│    trim/eff : 91.92%/96, disabled                  │          [cpu000:  8%]
└─ strategy: explore ────────── state: started :-) ──┘
```