Modern C++ Programming

14. Debugging and Testing

Federico Busato

2024-03-29

Table of Contents

1 Debugging Overview

2 Assertions

3 Execution Debugging

- Breakpoints
- Watchpoints / Catchpoints
- Control Flow
- Stack and Info
- Print
- Disassemble

Table of Contents

Table of Contents

Table of Contents

Feature Complete
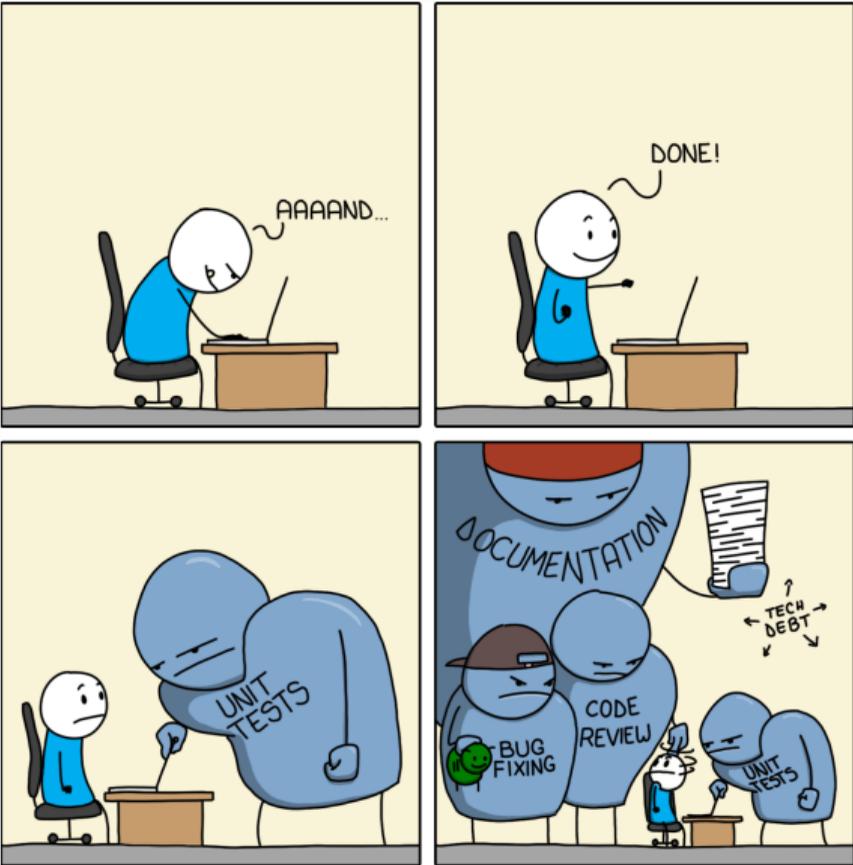
Debugging Overview

```
for (int i = 0; i <= (2^32) - 1; i++) {
```

"Software developers spend 35-50 percent of their time validating and debugging software. The cost of debugging, testing, and verification is estimated to account for 50-75 percent of the total budget of software development projects"
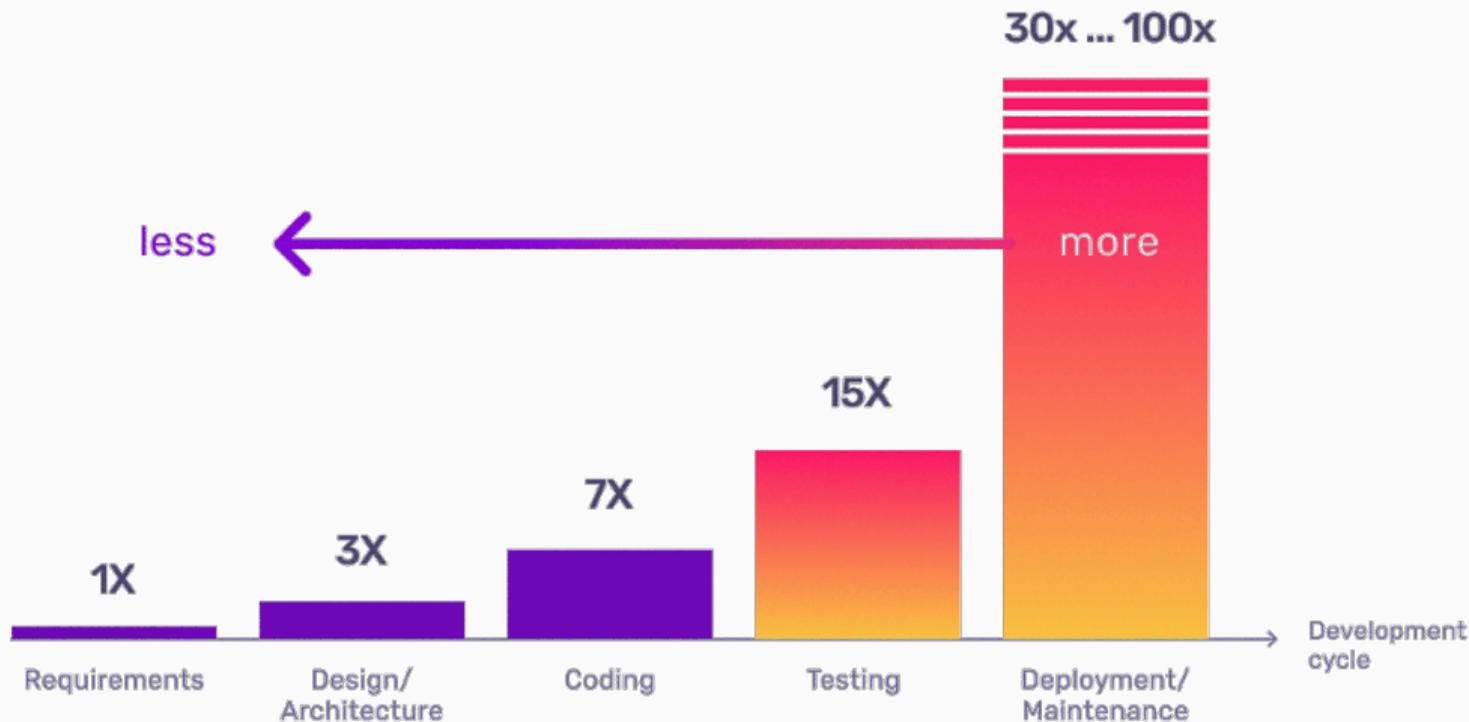
from: John Regehr (on Twitter)
The Debugging Mindset

Errors, Defects, and Failures

- An **error** is a human mistake. *Errors* lead to *software defects*

- A **defects** is an unexpected behavior of the software (correctness, performance, etc.). *Defects* potentially lead to *software failures*

- A **failure** is an observable incorrect behavior

Some examples:

- **The Millennium Bug** (2000): $100 billion
- **The Morris Worm** (1988): $10 million (single student)
- **Ariane 5** (1996): $370 million
- **Knight's unintended trades** (2012): $440 million
- **Bitcoin exchange error** (2011): $1.5 million
- **Pentium FDIV Bug** (1994): $475 million
- **Boeing 737 MAX** (2019): $3.9 million

see also:

```
11 of the most costly software errors in history
Historical Software Accidents and Errors
List of software bugs
```

Types of Software Defects

Ordered by fix complexity, (time to fix):

(1) Typos, Syntax, Formatting (seconds)

(2) Compilation Warnings/Errors (seconds, minutes)

(3) Logic, Arithmetic, Runtime Errors (minutes, hours, days)

(4) Resource Errors (minutes, hours, days)

(5) Accuracy Errors (hours, days)

(6) Performance Errors (days)

(7) Design Errors (weeks, months)

Causes of Bugs

- *C++ is very error prone language*, see 60 terrible tips for a C++ developer

- *Human behavior*, e.g. copying & pasting code is very common practice and can introduce subtle bugs $\rightarrow$ check the code carefully, deep understanding of its behavior

Program Errors

A **program error** is a set of conditions that produce an *incorrect result* or *unexpected behavior*, including performance regression, memory consumption, early termination, etc.

We can distinguish between two kind of errors:

Recoverable *Conditions that are not under the control of the program*. They indicates *"exceptional"* run-time conditions. e.g. file not found, bad allocation, wrong user input, etc.

Unrecoverable *It is a synonym of a bug*. It indicates a problem in the program logic. The program must terminate and modified. e.g. out-of-bound, division by zero, etc.

A *recoverable* should be considered *unrecoverable* if it is extremely rare and difficult to handle, e.g. bad allocation due to out-of-memory error

Dealing with Software Defects

Software defects can be identifies by:

Dynamic Analysis A _mitigation_ strategy that acts on the runtime state of a program.
  Techniques: Print, run-time debugging, sanitizers, fuzzing, unit test support, performance regression tests
  Limitations: Infeasible to cover all program states

Static Analysis A _proactive_ strategy that examines the source code for (potential) errors.
  Techniques: Warnings, static analysis tool, compile-time checks
  Limitations: Turing's undecidability theorem, exponential code paths

Assertions

Unrecoverable Errors and Assertions

Unrecoverable errors cannot be handled. They should be prevented by using *assertion* for ensuring *pre-conditions* and *post-conditions*

An **assertion** is a statement to detect a violated assumption. An assertion represents an *invariant* in the code

It can happen both at *run-time* (`assert`) and *compile-time* (`static_assert`). Run-time assertion failures should never be exposed in the normal program execution (e.g. release/public)

Assertion

```cpp
#include <cassert>     // <-- needed for "assert"
#include <cmath>       // std::is_finite
#include <type_traits> // std::is_arithmetic_v

template<typename T>
T sqrt(T value) {
    static_assert(std::is_arithmetic_v<T>,       // precondition
                  "T must be an arithmetic type");
    assert(std::is_finite(value) && value >= 0); // precondition
    int ret = ...                                // sqrt computation
    assert(std::is_finite(value) && ret >= 0 &&  // postcondition
           (ret == 0 || ret == 1 || ret < value));
    return ret;
}
```

Assertions may slow down the execution. They can be disable by define the `NDEBUG` macro

```cpp
#define NDEBUG // or with the flag "-DNDEBUG"
```

Execution
Debugging

Execution Debugging (gdb)

How to compile and run for debugging:

```
g++ -O0 -g [-g3] <program.cpp> -o program
gdb [--args] ./program <args...>
```

-O0 Disable any code optimization for helping the debugger. It is implicit for most compilers

-g Enable debugging
- stores the *symbol table information* in the executable (mapping between assembly and source code lines)
- for some compilers, it may disable certain optimizations
- slow down the compilation phase and the execution

-g3 Produces enhanced debugging information, e.g. macro definitions. Available for most compilers. Suggested instead of **-g**

gdb - **Breakpoints**

Command	Abbr.	Description
breakpoint <*file*>:<*line*>	b	insert a breakpoint in a specific line
breakpoint <*function_name*>	b	insert a breakpoint in a specific function
breakpoint <*ref*> if <*condition*>	b	insert a breakpoint with a conditional statement
delete	d	delete all breakpoints or watchpoints
delete <*breakpoint_number*>	d	delete a specific breakpoint
clear [*function_name/line_number*]		delete a specific breakpoint
enable/disable <*breakpoint_number*>		enable/disable a specific breakpoint
info breakpoints	**info** b	list all active breakpoints

gdb - **Watchpoints / Catchpoints**

Command	Abbr.	Description
watch <*expression*>		stop execution when the value of expression <u>changes</u> (variable, comparison, etc.)
rwatch <*variable/location*>		stop execution when variable/location <u>is read</u>
delete <*watchpoint_number*>	d	delete a specific watchpoint
info watchpoints		list all active watchpoints
catch throw		stop execution when an *exception* is thrown

gdb - **Control Flow**

Command	Abbr.	Description
run [args]	r	run the program
continue	c	continue the execution
finish	f	continue until the end of the current function
step	s	execute next line of code (follow function calls)
next	n	execute next line of code
until <*program_point*>		continue until reach line number, function name, address, etc.
CTRL+C		stop the execution (not quit)
quit	q	exit
help [<*command*>]	h	show help about command

gdb - Stack and Info

Command	Abbr.	Description
`list`	`l`	print code
`list <function or #start,#end>`	`l`	print function/range code
`up`	`u`	move up in the call stack
`down`	`d`	move down in the call stack
`backtrace [full]`	`bt`	prints stack backtrace (call stack) [local vars]
`info args`		print current function arguments
`info locals`		print local variables
`info variables`		print all variables
`info <breakpoints/watchpoints/registers>`		show information about program breakpoints/watchpoints/registers

gdb - **Print**

Command	Abbr.	Description
print <*variable*>	p	print variable
print/h <*variable*>	p/h	print variable in hex
print/**n**b <*variable*>	p/**n**b	print variable in binary (**n** bytes)
print/w <*address*>	p/w	print address in binary
p /s <*char array/address*>		print char array
p *array_var@**n**		print **n** array elements
p (int[4])<*address*>		print four elements of type int
p *(char**)&<*std::string*>		print std::string

gdb - Disassemble

Command	Description
`disasseble <function_name>`	disassemble a specified function
`disasseble <0xStart,0xEnd addr>`	disassemble function range
`nexti <variable>`	execute next line of code (follow function calls)
`stepi <variable>`	execute next line of code
`x/nfu <address>`	examine address n number of elements, f format (d: int, f: float, etc.), u data size (b: byte, w: word, etc.)

gdb - **Notes**

The debugger automatically stops when:

- breakpoint (by using the debugger)
- assertion fail
- segmentation fault
- trigger software breakpoint (e.g. SIGTRAP on Linux)
  github.com/scottt/debugbreak

Full story: www.yolinux.com/TUTORIALS/GDB-Commands.html (it also contains a script to *de-referencing* STL Containers)

gdb reference card V5 link

Memory Debugging

"70% of all the vulnerabilities in Microsoft products are memory safety issues"

Matt Miller, *Microsoft Security Engineer*

"Chrome: 70% of all security bugs are memory safety issues"

Chromium Security Report

"you can expect at least 65% of your security vulnerabilities to be caused by memory unsafety"

What science can tell us about C and C++'s security

```
Microsoft: 70% of all security bugs are memory safety issues
Chrome: 70% of all security bugs are memory safety issues
What science can tell us about C and C++'s security
```

"Memory Unsafety in Apple's OS represents 66.3%- 88.2% of all the vulnerabilities"

"Out of bounds (OOB) reads/writes comprise ~70% of all the vulnerabilities in Android"

Jeff Vander, Google, Android Media Team

"Memory corruption issues are the root-cause of 68% of listed CVEs"

Ben Hawkes, Google, Project Zero

```
Memory Unsafety in Apple's Operating Systems
Google Security Blog: Queue the Hardening Enhancements
Google Project Zero
```

Terms like *buffer overflow*, *race condition*, *page fault*, *null pointer*, *stack exhaustion*, *heap exhaustion/corruption*, *use-after-free*, or *double free* – all describe **memory safety vulnerabilities**

Mitigation:

- Run-time check
- Static analysis
- Avoid unsafe language constructs

valgrind ☑ is a tool suite to automatically detect many memory management and threading bugs

How to install the last version:

```
$ wget ftp://sourceware.org/pub/valgrind/valgrind-3.21.tar.bz2
$ tar xf valgrind-3.21.tar.bz2
$ cd valgrind-3.21
$ ./configure --enable-lto
$ make -j 12
$ sudo make install
$ sudo apt install libc6-dbg #if needed
```

some linux distributions provide the package through `apt install valgrid`, but it could be an old version

Basic usage:

- compile with `-g`

  - ```
    $ valgrind ./program <args...>
    ```

Output example 1:

```
==60127== Invalid read of size 4                    !!out-of-bound access
==60127==    at 0x100000D9E: f(int) (main.cpp:86)
==60127==    by 0x100000C22: main (main.cpp:40)
==60127==  Address 0x10042c148 is 0 bytes after a block of size 40 alloc'd
==60127==    at 0x1000161EF: malloc (vg_replace_malloc.c:236)
==60127==    by 0x100000C88: f(int) (main.cpp:75)
==60127==    by 0x100000C22: main (main.cpp:40)
```

Output example 2:

```
!!memory leak
==19182== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==19182==    at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
==19182==    by 0x8048385: f (main.cpp:5)
==19182==    by 0x80483AB: main (main.cpp:11)


==60127== HEAP SUMMARY:
==60127==     in use at exit: 4,184 bytes in 2 blocks
==60127==   total heap usage: 3 allocs, 1 frees, 4,224 bytes allocated
==60127==
==60127== LEAK SUMMARY:
==60127==    definitely lost: 128 bytes in 1 blocks      !!memory leak
==60127==    indirectly lost: 0 bytes in 0 blocks
==60127==      possibly lost: 0 bytes in 0 blocks
==60127==    still reachable: 4,184 bytes in 2 blocks  !!not deallocated
==60127==         suppressed: 0 bytes in 0 blocks
```

Memory leaks are divided into four categories:

- *Definitely lost*
- *Indirectly lost*
- *Still reachable*
- *Possibly lost*

When a program terminates, it releases all heap memory allocations. Despite this, leaving memory leaks is considered a *bad practice* and *makes the program unsafe* with respect to multiple internal iterations of a functionality. If a program has memory leaks for a single iteration, is it safe for multiple iterations?

A **robust program** prevents any memory leak even when abnormal conditions occur

Definitely lost indicates blocks that are *not deleted at the end of the program* (return from the main() function). The common case is local variables pointing to newly allocated heap memory

```
void f() {
    int* y = new int[3]; // 12 bytes definitely lost
}

int main() {
    int* x = new int[10]; // 40 bytes definitely lost
    f();
}
```

Indirectly lost indicates blocks pointed by other heap variables that are not deleted.
The common case is global variables pointing to newly allocated heap memory

```cpp
struct A {
    int* array;
};

int main() {
    A* x    = new A;       // 8 bytes definitely lost
    x->array = new int[4]; // 16 bytes indirectly lost
}
```

Still reachable indicates blocks that are *not deleted but they are still reachable at the end of the program*

```
int* array;

int main() {
    array = new int[3];
}
// 12 bytes still reachable (global static class could delete it)
```

```
#include <cstdlib>
int main() {
    int* array = new int[3];
    std::abort();                  // early abnormal termination
    // 12 bytes still reachable
    ... // maybe it is delete here
}
```

Possibly lost indicates blocks that are still reachable but pointer arithmetic makes the deletion more complex, or even not possible

```cpp
#include <cstdlib>
int main() {
    int* array = new int[3];
    array++;                      // pointer arithmetic
    std::abort();                 // early abnormal termination
    // 12 bytes still reachable
    ... // maybe it is delete here but you should be able
        // to revert pointer arithmetic
}
```

Advanced flags:

- `--leak-check=full` print details for each "definitely lost" or "possibly lost" block, including where it was allocated

- `--show-leak-kinds=all` to combine with --leak-check=full. Print all leak kinds

- `--track-fds=yes` list open file descriptors on exit (not closed)

- `--track-origins=yes` tracks the origin of uninitialized values (very slow execution)

```
valgrind --leak-check=full --show-leak-kinds=all
        --track-fds=yes --track-origins=yes ./program <args...>
```

Track stack usage:

```
valgrind --tool=drd --show-stack-usage=yes ./program <args...>
```

Hardening Techniques

References

- Compiler Options Hardening Guide for C and C++ [March, 2024]
- Hardened mode of standard library implementations

Compile-time Stack Usage

- `-Wstack-usage=<byte-size>` Warn if the stack usage of a function might exceed byte-size. The computation done to determine the stack usage is conservative (no VLA)

- `-fstack-usage` Makes the compiler output stack usage information for the program, on a per-function basis

- `-Wvla` Warn if a variable-length array is used in the code

- `-Wvla-larger-than=<byte-size>` Warn for declarations of variable-length arrays whose size is either unbounded, or bounded by an argument that allows the array size to exceed byte-size bytes

Use compiler flags for stack protection in GCC and Clang

Compile-time Stack Protection

- `-Wtrampolines` Check whether the compiler generates trampolines for pointers to nested functions which may interfere with stack virtual memory protection

- `-Wl,-z,noexecstack` Enable data execution prevention by marking stack memory as non-executable

Run-time Stack Usage

- `-fstack-clash-protection` Enables run-time checks for variable-size stack allocation validity

- `-fstack-protector-strong` Enables run-time checks for stack-based buffer overflows using strong heuristic

- `-fstack-protector-all` Enables run-time checks for stack-based buffer overflows for all functions

`_FORTIFY_SOURCE` **define**: the compiler provides buffer overflow checks for the following functions:

`memcpy`, `mempcpy`, `memmove`, `memset`, `strcpy`, `stpcpy`, `strncpy`, `strcat`, `strncat`, `sprintf`, `vsprintf`, `snprintf`, `vsnprintf`, `gets`.

Recent compilers (e.g. GCC 12+, Clang 9+) allow detects buffer overflows with enhanced coverage, e.g. dynamic pointers, with `_FORTIFY_SOURCE=3` *

---

*GCC's new fortification level: The gains and costs

```cpp
#include <cstring> // std::memset
#include <string>  // std::stoi
int main(int argc, char** argv) {
    int  size = std::stoi(argv[1]);
    char buffer[24];
    std::memset(buffer, 0xFF, size);
}
```

```
$ gcc -O1 -D_FORTIFY_SOURCE program.cpp -o program
$ ./program 12 # OK
$ ./program 32 # Wrong
$ *** buffer overflow detected ***: ./program terminated
```

Standard Library Precondictions

The standard library provides run-time precondition checks for library calls, such as bounds-checks for strings and containers, and null-pointer checks, etc.

`-D_GLIBCXX_ASSERTIONS` for libstdc++ (GCC)

`-D_LIBCPP_ASSERT` , `_LIBCPP_HARDENING_MODE_EXTENSIVE` for libc++ (LLVM):

- `-fno-strict-overflow` Prevent code optimization (code elimination) due to signed integer undefined behavior

- `-fwrapv` Signed integer has the same semantic of unsigned integer, with a well-defined wrap-around behavior

- `-fno-strict-aliasing` Strict aliasing means that two objects with the same memory address are not same if they have a different type, undefined behavior otherwise. The flag disables this constraint

- `-fno-delete-null-pointer-checks` NULL pointer dereferencing is undefined behavior and the compiler can assume that it never happens. The flag disable this optimization

- `-ftrivial-auto-var-init[=<hex pattern>]` Ensures that default initialization initializes variables with a fixed 1-byte pattern. Explicit uninitialized variables requires the `[[uninitialized]]` attribute

Control Flow Protections

- `-fcf-protection=full` Enable control flow protection to counter Return Oriented Programming (ROP) and Jump Oriented Programming (JOP) attacks on many x86 architectures

- `-mbranch-protection=standard` Enable branch protection to counter Return Oriented Programming (ROP) and Jump Oriented Programming (JOP) attacks on AArch64

Other Run-time Checks

- `-fPIE -pie` Position-Independent Executable enables the support for address space layout randomization, which makes exploits more difficult.

- `-Wl,-z,relro,-z,now` Prevents modification of the Global Offset Table (locations of functions from dynamically linked libraries) after the program startup

- `-Wl,-z,nodlopen` Restrict `dlopen(3)` calls to shared objects

Sanitizers

Address Sanitizer

Sanitizers are compiler-based instrumentation components to perform *dynamic* analysis

Sanitizer are used during development and testing to discover and diagnose memory misuse bugs and potentially dangerous undefined behavior

Sanitizer are implemented in `Clang` (from 3.1), `gcc` (from 4.8) and `Xcode`

Project using Sanitizers:
- Chromium
- Firefox
- Linux kernel
- Android

Address Sanitizer

Address Sanitizer ⧉ is a memory error detector

- heap/*stack*/*global* out-of-bounds
- memory leaks
- use-after-free, use-after-return, use-after-scope
- double-free, invalid free
- initialization order bugs
- * Similar to `valgrind` but faster (50X slowdown)

```
clang++ -O1 -g -fsanitize=address -fno-omit-frame-pointer <program>
```

`-O1` disable inlining

`-g` generate symbol table

- github.com/google/sanitizers/wiki/AddressSanitizer
- gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html

Leak Sanitizer

[LeakSanitizer](#) ⍽ is a run-time *memory leak* detector

- integrated into AddressSanitizer, can be used as standalone tool
- * almost no performance overhead until the very end of the process

```
g++     -O1 -g -fsanitize=address -fno-omit-frame-pointer <program>
clang++ -O1 -g -fsanitize=leak    -fno-omit-frame-pointer <program>
```

- github.com/google/sanitizers/wiki/AddressSanitizerLeakSanitizer
- gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html

Memory Sanitizers

[Memory Sanitizer](#) ⧉ is a detector of *uninitialized* reads

- stack/heap-allocated memory read before it is written
- * Similar to valgrind but faster (3X slowdown)

```
clang++ -O1 -g -fsanitize=memory -fno-omit-frame-pointer <program>
```

`-fsanitize-memory-track-origins=2`
  track origins of uninitialized values

Note: not compatible with Address Sanitizer

- github.com/google/sanitizers/wiki/MemorySanitizer
- gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html

Undefined Behavior Sanitizer

UndefinedBehaviorSanitizer ⬀ is an *undefined behavior* detector

- signed integer overflow, floating-point types overflow, enumerated not in range
- out-of-bounds array indexing, misaligned address
- divide by zero
- etc.
- * Not included in `valgrind`

```
clang++ -O1 -g -fsanitize=undefined -fno-omit-frame-pointer <program>
```

Undefined Behavior Sanitizer

`-fsanitize=<options>`:

`undefined`	All of the checks other than `float-divide-by-zero`, `unsigned-integer-overflow`, `implicit-conversion`, `local-bounds` and the `nullability-*` group of checks
`float-divide-by-zero`	Undefined behavior in C++, but defined by Clang and IEEE-754
`integer`	Checks for undefined or suspicious integer behavior (e.g. unsigned integer overflow)
`implicit-conversion`	Checks for suspicious behavior of implicit conversions
`local-bounds`	Out of bounds array indexing, in cases where the array bound can be statically determined
`nullability`	Checks passing `null` as a function parameter, assigning `null` to an lvalue, and returning `null` from a function

Bug	Valgrind detection	ASan detection
Uninitialized memory read	Yes	No *
Write overflow on heap	Yes	Yes
Write overflow on stack	No	Yes
Read overflow on heap	Yes	Yes
Read underflow on heap	Yes	Yes
Read overflow on stack	No	Yes
Use-after-free	Yes	Yes
Use-after-return	No	Yes
Double-free	Yes	Yes
Memory leak	Yes	Yes
Undefined behavior	No	No **

Valgrind █ ASan

Execution Time /sec   Memory Usage / MB

Valgrind - A neglected tool from the shadows or a serious debugging tool?

Debugging Summary

How to Debug Common Errors

Segmentation fault

- gdb, valgrind, sanitizers
- Segmentation fault when just entered in a function → stack overflow

Double free or corruption

- gdb, valgrind, sanitizers

Infinite execution

- gdb + (CTRL + C)

Incorrect results

- valgrind + assertion + gdb + sanitizers

Compiler Warnings

Compiler Warnings - GCC and Clang

Enable specific warnings:

```
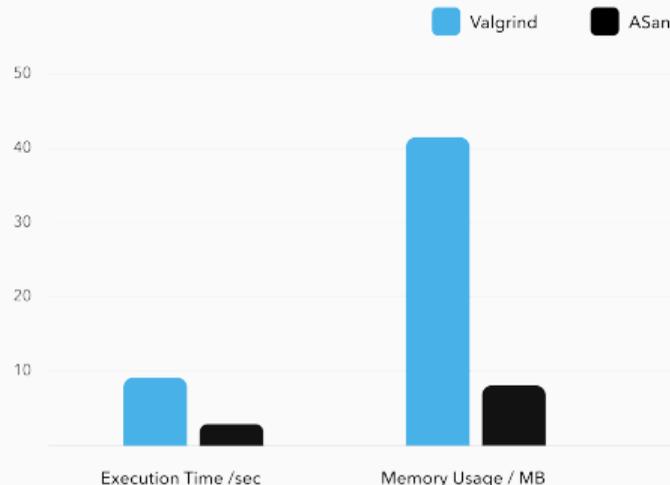g++ -W<warning> <args...>
```

Disable specific warnings:

```
g++ -Wno-<warning> <args...>
```

Common warning flags to minimize accidental mismatches:

-Wall   Enables many standard warnings ($\sim$50 warnings)

-Wextra   Enables some extra warning flags that are not enabled by -Wall ($\sim$15 warnings)

-Wpedantic   Issue all the warnings demanded by strict ISO C/C++

Enable ALL warnings, only clang: -Weverything

Compiler Warnings - MSVC

Enable specific warnings:

```
cl.exe /W<level><warning_id> <args...>
```

Disable specific warnings:

```
cl.exe /We<warning_id> <args...>
```

Common warning flags to minimize accidental mismatches:

- `/W1` Severe warnings
- `/W2` Significant warnings
- `/W3` Production quality warnings
- `/W4` Informational warnings
- `/Wall` All warnings

Static Analysis

Overview

Static analysis is the process of source code examination to find potential issues

Benefits of static code analysis:

- Problem identification before the execution
- Analyze the program outside the execution environment
- The analysis is independent from the run-time tests
- Enforce code quality and compliance by ensuring that the code follows specific rules and standards
- Identify security vulnerabilities

Static Analyzers - `Clang` and `GCC`

The **Clang Static Analyzer** ☞ (LLVM suite) finds bugs by reasoning about the semantics of code (may produce false positives)

```
void test() {
    int i, a[10];
    int x = a[i]; // warning: array subscript is undefined
}
```

```
scan-build make
```

The **GCC Static Analyzer** ☞ can diagnose various kinds of problems in C/C++ code at compile-time (e.g. double-free, use-after-free, stdio related, etc) by adding the `-fanalyzer` flag

Static Analyzers - `cppcheck`

The **MSVC Static Analyzer** ⌇ Enables code analysis and control options (e.g. double-free, use-after-free, stdio related, etc) by adding the `/analyze` flag

cppcheck ⌇ provides code analysis to detect bugs, undefined behavior and dangerous coding construct. The goal is to detect only real errors in the code (i.e. have very few false positives)

```
cppcheck --enable=warning,performance,style,portability,information,error
         <src_file/directory>
```

```
cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=ON .
cppcheck --enable=<enable_flags> --project=compile_commands.json
```

Popular Static Analyzers - `PVS-Studio, SonarLint`

`PVS-Studio` ⬀ is a high-quality *proprietary* (free for open source projects) static code analyzer supporting C, C++

Customers: IBM, Intel, Adobe, Microsoft, Nvidia, Bosh, IdGames, EpicGames, etc.

`SonarSource` ⬀ is a static analyzer which inspects source code for bugs, code smells, and security vulnerabilities for multiple languages (C++, Java, etc.)

`SonarLint` plugin is available for Visual Code, Visual Studio Code, Eclipse, and IntelliJ IDEA
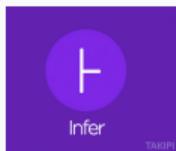
FBInfer ☞ is a static analysis tool (also available online) to checks for null pointer dereferencing, memory leak, coding conventions, unavailable APIs, etc.

Customers: Amazon AWS, Facebook/Ocolus, Instagram, Whatapp, Mozilla, Spotify, Uber, Sky, etc.

DEEP CODE   **deepCode** ☞ is an AI-powered code review system, with machine learning systems trained on billions of lines of code from open-source projects

Available for Visual Studio Code, Sublime, IntelliJ IDEA, and Atom

---

see also: `A curated list of static analysis tool`

Code Testing

Code Testing
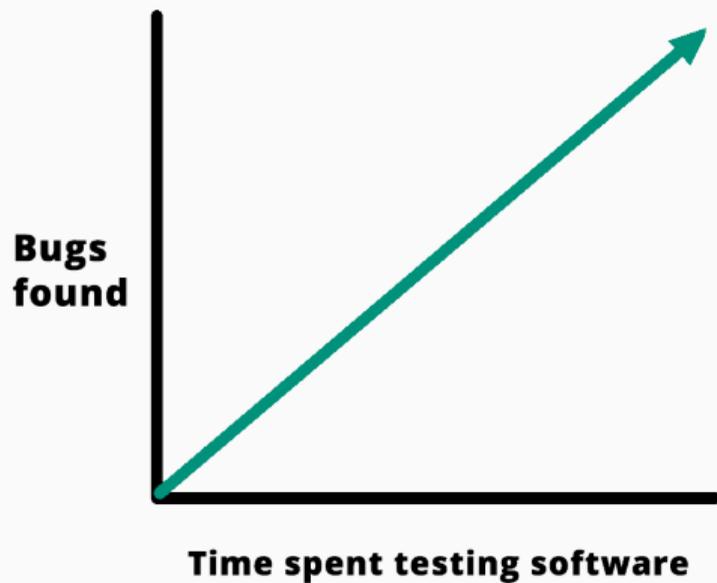
see `Case Study 4:  The $440 Million Software Error at Knight Capital`

Code Testing

Unit Test A *unit* is the smallest piece of code that can be logically isolated in a system. *Unit test* refers to the verification of a *unit*. It supposes the full knowledge of the code under testing (*white-box* testing)

Goals: meet specifications/requirements, fast development/debugging

Functional Test Output validation instead of the internal structure (*black-box* testing)

Goals: performance, regression (same functionalities of previous version), stability, security (e.g. sanitizers), composability (e.g. integration test)

Unit testing involves breaking your program into pieces, and subjecting each piece to a series of tests

Unit testing should observe the following key features:

- **Isolation**: Each unit test should be *independent* and avoid external interference from other parts of the code
- **Automation**: Non-user interaction, easy to run, and manage
- **Small Scope**: Unit tests focus on small portions of code or specific functionalities, making it easier to identify bugs

Popular C++ Unit testing frameworks:

catch, doctest, Google Test, CppUnit, Boost.Test

Meeting C++ Community Survey
Which unit test libraries do you use? (n=865)

35%	**Google Test**	
26%	I don't write unit tests for C++	
17%	I write unit tests but don't use any frameworks	
12%	Catch	
9%	CppUnit	
7%	Boost.Test	
3%	CppUTest	
3%	doctest	
4%	Other	
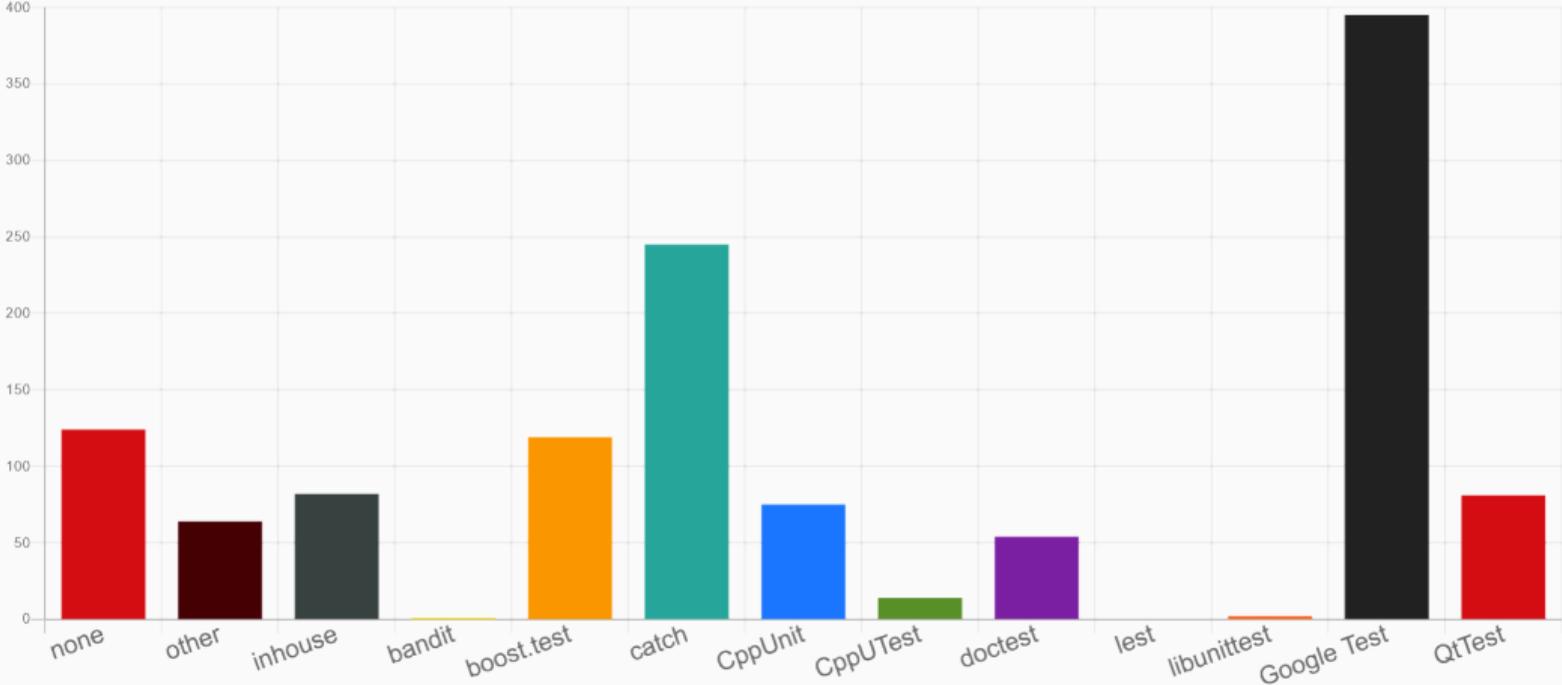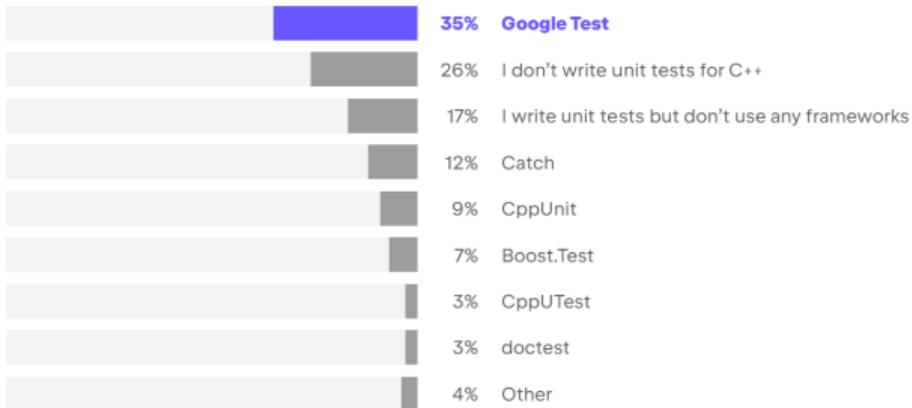
The statistic that a quarter of developers aren't writing unit tests freaks me out. I don't feel strongly about how you express those or what framework you use, but we all do need to be writing tests.

Titus Winters
Principal Engineer at Google

Test-Driven Development (TDD)

Unit testing is often associated with the **Test-Driven Development (TDD)** methodology. The practice involves the definition of *automated functional tests* <u>before</u> implementing the functionality

The process consists of the following steps:

1. Write a test for a new functionality
2. Write the minimal code to pass the test
3. Improve/Refactor the code iterating with the test verification
4. Go to 1.

Test-Driven Development (TDD) - Main advantages

- **Software design**. Strong focus on interface definition, expected behavior, specifications, and requirements before working at lower level

- **Maintainability/Debugging Cost** Small, incremental changes allow you to catch bugs as they are introduced. Later refactoring or the introduction of new features still rely on well-defined tests

- **Understandable behavior**. New user can learn how the system works and its properties from the tests

- **Increase confidence**. Developers are more confident that their code will work as intended because it has been extensively tested

- **Faster development**. Incremental changes, high confidence, and automation make it easy to move through different functionalities or enhance existing ones

Catch2 ↗ is a multi-paradigm test framework for C++

Catch2 features

- Header only and no external dependencies
- Assertion macro
- Floating point tolerance comparisons

Basic usage:

- Create the test program
- Run the test

```
$ ./test_program [<TestName>]
```

- github.com/catchorg/Catch2
- The Little Things: Testing with Catch2

```cpp
#define CATCH_CONFIG_MAIN  // This tells Catch to provide a main()
#include "catch.hpp"        // only do this in one cpp file

unsigned Factorial(unsigned number) {
    return number <= 1 ? number : Factorial(number - 1) * number;
}

"Test description and tag name"
TEST_CASE( "Factorials are computed", "[Factorial]" ) {
    REQUIRE( Factorial(1) == 1 );
    REQUIRE( Factorial(2) == 2 );
    REQUIRE( Factorial(3) == 6 );
    REQUIRE( Factorial(10) == 3628800 );
}

float floatComputation() { ... }

TEST_CASE( "floatCmp computed", "[floatComputation]" ) {
    REQUIRE( floatComputation() == Approx( 2.1 ) );
}
```

Code coverage is a measure used to describe the degree to which the source code of a program is executed when a particular execution/test suite runs

`gcov` and `llvm-profdata/llvm-cov` are tools used in conjunction with compiler instrumentation (gcc, clang) to interpret and visualize the raw code coverage generated during the execution

`gcovr` and `lcov` are utilities for managing `gcov/llvm-cov` at higher level and generating code coverage results

Step for code coverage:

- Compile with `--coverage` flag (objects + linking)
- Run the program / test
- Visualize the results with `gcovr`, `llvm-cov`, `lcov`

program.cpp:

```cpp
#include <iostream>
#include <string>

int main(int argc, char* argv[]) {
    int value = std::stoi(argv[1]);
    if (value % 3 == 0)
        std::cout << "first\n";
    if (value % 2 == 0)
        std::cout << "second\n";
}
```

```
$ gcc -g --coverage program.cpp -o program
$ ./program 9
first
$ gcovr -r --html --html-details <path>  # generate html
# or
$ lcov --coverage --directory . --output-file coverage.info
$ genhtml coverage.info --output-directory <path> # generate html
```

```
    1:    4:int main(int argc, char* argv[]) {
    1:    5:    int value = std::stoi(argv[1]);
    1:    6:    if (value % 3 == 0)
    1:    7:        std::cout << "first\n";
    1:    8:    if (value % 2 == 0)
#####:    9:        std::cout << "second\n";
    4:   10:}
```

Current view:	top level - /home/ubuntu/workspace/prove		Hit	Total	Coverage
Test:	coverage.info	Lines:	6	7	85.7 %
Date:	2018-02-09	Functions:	3	3	100.0 %

Filename	Line Coverage ⬍		Functions ⬍	
program.cpp	85.7 %	6 / 7	100.0 %	3 / 3

Current view:	top level - home/ubuntu/workspace/prove - program.cpp (source / functions)		Hit	Total	Coverage
Test:	coverage.info	Lines:	6	7	85.7 %
Date:	2018-02-09	Functions:	3	3	100.0 %

```
      Line data    Source code
    1       : #include <iostream>
    2       : #include <string>
    3       :
    4     1 : int main(int argc, char* argv[]) {
    5     1 :     int value = std::stoi(argv[1]); // convert to int
    6     1 :     if (value % 3 == 0)
    7     1 :         std::cout << "first";
    8     1 :     if (value % 2 == 0)
    9     0 :         std::cout << "second";
   10     4 : }
```

Coverage-Guided Fuzz Testing

A **fuzzer** is a specialized tool that tracks which areas of the code are reached, and generates *mutations* on the corpus of input data in order to *maximize* the code coverage

[LibFuzzer](#) ⧉ is the library provided by LLVM and feeds fuzzed inputs to the library via a specific fuzzing entrypoint

The *fuzz target function* accepts an array of bytes and does something interesting with these bytes using the API under test:

```cpp
extern "C" int LLVMFuzzerTestOneInput(const uint8_t* Data,
                                      size_t          Size) {
  DoSomethingInterestingWithMyAPI(Data, Size);
  return 0;
}
```

Code Quality

lint: The term was derived from the name of the undesirable bits of fiber

`clang-tidy` ⬦ provides an extensible framework for diagnosing and fixing typical *programming errors*, like *style violations*, *interface misuse*, or *bugs* that can be deduced via static analysis

```
$ cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=ON .
$ clang-tidy -p .
```

clang-tidy searches the configuration file `.clang-tidy` file located in the closest parent directory of the input file

clang-tidy is included in the LLVM suite

Coding Guidelines:

- CERT Secure Coding Guidelines
- C++ Core Guidelines
- High Integrity C++ Coding Standard

Supported Code Conventions:

- Fuchsia
- Google
- LLVM

Bug Related:

- Android related
- Boost library related
- Misc
- Modernize
- Performance
- Readability
- clang-analyzer checks
- bugprone code constructors

`.clang-tidy`

```
Checks: 'android-*,boost-*,bugprone-*,cert-*,cppcoreguidelines-*,
clang-analyzer-*,fuchsia-*,google-*,hicpp-*,llvm-*,misc-*,modernize-*,
performance-*,readability-*'
```