

# Modern C++ Programming

## 12. C++ ECOSYSTEM

---

*Federico Busato*

University of Verona, Dept. of Computer Science  
2022, v3.16



## 1 Debugging

- Assertion
- Execution Debugging (gdb)

## 2 Memory Debugging

- valgrind
- Stack Protection

## 3 Sanitizers

- Address Sanitizer
- Leak Sanitizer
- Memory Sanitizers
- Undefined Behavior Sanitizer

## **4** Debugging Summary

## **5** Code Checking and Analysis

- Compiler Warnings
- Static Analyzers

## **6** Code Testing

- Unit Test
- Code Coverage
- Fuzz Testing

## **7** Code Quality

- clang-tidy

## 8 CMake

- ctest

## 9 Code Documentation

- doxygen

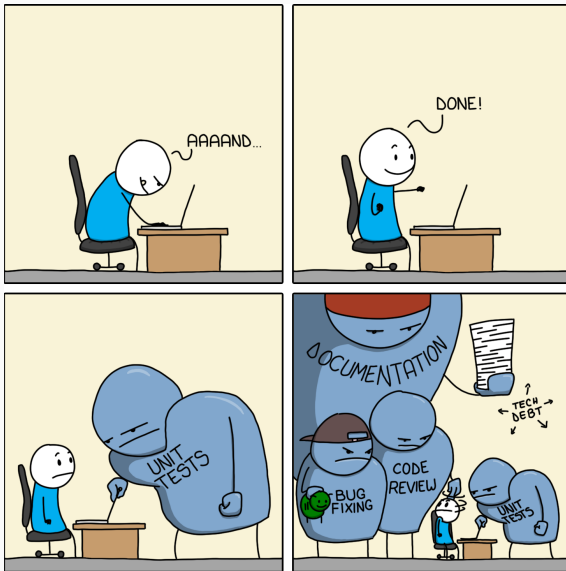
## 10 Code Statistics

- Count Lines of Code
- Cyclomatic Complexity Analyzer

## 11 Other Tools

- Code Formatting - `clang-format`
- Compiler Explorer
- Code Transformation - `CppInsights`
- Code Autocompletion - `TabNine`, `Kite`
- Local Code Search - `ripgrep`
- Code Search Engine - `searchcode`, `grep.app`
- Code Benchmarking - `Quick-Bench`
- Font for Coding

# Feature Complete



# Debugging

---

## Is this a bug?

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



# Program Errors

A **program error** is a set of conditions that produce an *incorrect result* or *unexpected behavior*

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. The program must terminate. e.g. out-of-bound, division by zero, etc.*

# Dealing with Program Errors and Bugs

Software defects can be identified 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

*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

# 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

```
#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 disabled by defining the `NDEBUG` macro

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

# 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/Watchpoints

Command	Abbr.	Description
<code>breakpoint &lt;file&gt;:&lt;line&gt;</code>	b	insert a breakpoint in a specific line
<code>breakpoint &lt;function_name&gt;</code>	b	insert a breakpoint in a specific function
<code>breakpoint &lt;ref&gt; if &lt;condition&gt;</code>	b	insert a breakpoint with a conditional statement
<code>delete</code>	d	delete all breakpoints or watchpoints
<code>delete &lt;breakpoint_number&gt;</code>		delete a specific breakpoint
<code>clear [function_name/line_number]</code>		delete a specific breakpoint
<code>enable/disable &lt;breakpoint_number&gt;</code>		enable/disable a specific breakpoint
<code>watch &lt;expression&gt;</code>		stop execution when the value of expression changes (variable, comparison, etc.)

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

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



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

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

### 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**

---

Microsoft: 70% of all security bugs are memory safety issues

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

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**

Solutions:

- 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-3.18.1.tar.bz2
$ tar xf valgrind-3.18.1.tar.bz2
$ cd valgrind-3.18.1
$ ./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 to by other heap variables that are not deleted. The common case is global variables pointing to newly allocated heap memory

```
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

```
#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...>
```

## Stack size check:

- `-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



Adding `_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` .

```
#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
```

# Sanitizers

---

**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

- 
- [clang.llvm.org/docs/AddressSanitizer.html](http://clang.llvm.org/docs/AddressSanitizer.html)
  - [github.com/google/sanitizers/wiki/AddressSanitizer](https://github.com/google/sanitizers/wiki/AddressSanitizer)
  - [gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html](http://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>
```

- 
- [clang.llvm.org/docs/LeakSanitizer.html](http://clang.llvm.org/docs/LeakSanitizer.html)
  - [github.com/google/sanitizers/wiki/AddressSanitizerLeakSanitizer](https://github.com/google/sanitizers/wiki/AddressSanitizerLeakSanitizer)
  - [gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html](http://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html)

# Memory Sanitizers

Memory Sanitizer is 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

- 
- [clang.llvm.org/docs/MemorySanitizer.html](http://clang.llvm.org/docs/MemorySanitizer.html)
  - [github.com/google/sanitizers/wiki/MemorySanitizer](https://github.com/google/sanitizers/wiki/MemorySanitizer)
  - [gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html](http://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html)

# Undefined Behavior Sanitizer

UndefinedBehaviorSanitizer is a *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>
```

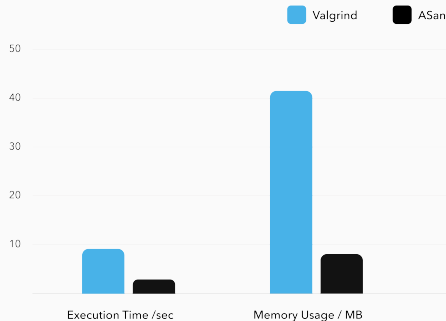
`-fsanitize=integer` Checks for undefined or suspicious integer behavior (e.g. unsigned integer overflow)

`-fsanitize=nullability` Checks passing null as a function parameter, assigning null to an lvalue, and returning null from a function

- 
- [clang.llvm.org/docs/UndefinedBehaviorSanitizer.html](http://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html)
  - [gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html](http://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html)

# Sanitizers vs. Valgrind

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 **





# Debugging Summary

---

# How to Debug Common Errors

## Segmentation fault

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

## Double free or corruption

- gdb
- valgrind

## Infinite execution

- gdb + (CTRL + C)

## Incorrect results

- valgrind + assertion + gdb + UndefinedBehaviorSanitizer

# Code Checking and Analysis

---

# Compiler Warnings

**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 (~50 warnings)

**-Wextra** Enables some extra warning flags that are not enabled by **-Wall** (~15 warnings)

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

Enable ALL warnings (only clang) **-Weverything**

# GCC Warnings

Additional GCC warning flags ( $\geq 5.0$ ):

```
-Wcast-align
-Wcast-qual
-Wconversion
# -Wfloat-conversion
# -Wsign-conversion
-Wdate-time
-Wdouble-promotion
-Weffc++
# -Wdelete-non-virtual-dtor
# -Wnon-virtual-dtor
-Wformat-signedness
-Winvalid-pch
-Wlogical-op
-Wmissing-declarations
-Wmissing-include-dirs
-Wodr
```

```
-Wold-style-cast
-Wpragmas
-Wredundant-decls
-Wshadow
-Wsign-promo*
-Wstrict-aliasing
-Wstrict-overflow=1 # 5
-Wswitch-bool
# -Wswitch-default
# -Wswitch-enum
-Wtrampolines
-Wunused-macros
-Wuseless-cast
-Wvla
-Wformat=2
-Wno-long-long
```

# Static Analyzers - clang static analyzer



The Clang Static Analyzer is a source code analysis tool that finds bugs in C/C++ programs at compile-time

It find bugs by reasoning about the semantics of code (may produce false positives)

Example:

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

**How to use:**

```
scan-build make
```

scan-build is included in the LLVM suite

# Static Analyzers - cppcheck



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) `-fanalyzer`

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>
```

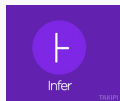
```
cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=ON .  
cppcheck --enable=<enable_flags> --project=compile_commands.json
```

## Static Analyzers - PVS-Studio, FBInfer



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.




FBInfer is a static analysis tool (also available online) to checks for null pointer deferencing, memory leak, coding conventions, unavailable APIs, etc.

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



## Static Analyzers - DeepCode, SonarSource

 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

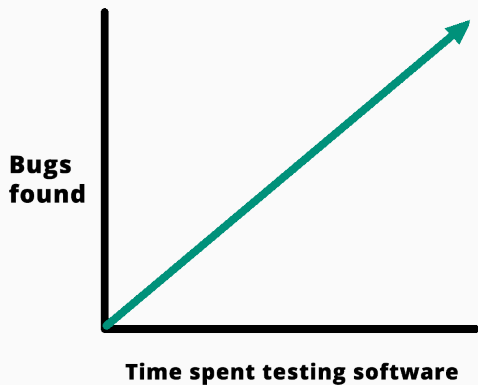


SonarSource is a static analyzer which inspects source code for bugs, code smells, and security vulnerabilities for multiple languages (C++, Java, etc.)

SonarLint plugin is available for Visual Code, Visual Studio Code, Eclipse, and IntelliJ IDEA

# Code Testing

---



see Case Study 4: The \$440 Million Software Error at Knight Capital

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

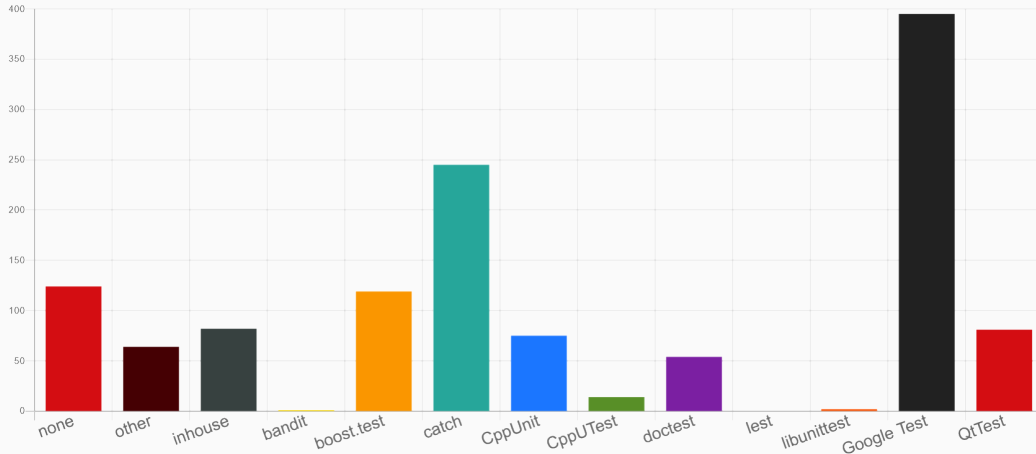
## **Unit Testing Benefits:**

- Increases confidence in changing/ maintaining code
- The cost of fixing a defect detected during unit testing is lesser in comparison to that of defects detected at higher levels
- Debugging is easy. When a test fails, only the latest changes need to be debugged

## **C++ Unit testing frameworks:**

- catch
- doctest
- Google Test
- CppUnit
- Boost.Test

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



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](https://github.com/catchorg/Catch2)
  - The Little Things: Testing with Catch2

```
#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 test suite runs

gcov is a tool you can use in conjunction with GCC to test code coverage in programs

gcovr is a utility for managing `gcov` and generating code coverage results

## Step for code coverage:

- compile with `--coverage` flag (objects + linking)
- run the test
- visualize the results with `gcovr`



program.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_to_cover>
# generate coverage.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

Test: coverage.info

Date: 2018-02-09

	Hit	Total	Coverage
Lines:	6	7	85.7 %
Functions:	3	3	100.0 %

Filename	Line Coverage ↕	Functions ↕
program.cpp	<div><div></div></div> 85.7 % 6 / 7	100.0 % 3 / 3

Current view: [top level](#) - [home/ubuntu/workspace/prove](#) - program.cpp (source / functions)

Test: coverage.info

Date: 2018-02-09

	Hit	Total	Coverage
Lines:	6	7	85.7 %
Functions:	3	3	100.0 %

Line data Source code

```

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

```
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-*
```

# CMake

---



CMake is an *open-source*, cross-platform family of tools designed to build, test and package software

CMake is used to control the software compilation process using simple platform and compiler independent configuration files, and *generate* native Makefile/Ninja and workspaces that can be used in the compiler environment of your choice

CMake features:

- Turing complete language (if/else, loops, functions, etc.)
- Multi-platform (Windows, Linux, etc.)
- Open-Source
- Generate: makefile, ninja, etc.
- Supported by many IDEs: Visual Studio, Clion, Eclipse, etc.



- 19 reasons why CMake is actually awesome
- An Introduction to Modern CMake
- Effective Modern CMake
- Awesome CMake
- Useful Variables

# Install CMake

## Using PPA repository

```
$ wget -O - https://apt.kitware.com/keys/kitware-archive-latest.asc 2>/dev/null |  
  gpg --dearmor - | sudo tee /etc/apt/trusted.gpg.d/kitware.gpg >/dev/null  
$ sudo apt-add-repository 'deb https://apt.kitware.com/ubuntu/ focal main' # bionic, xenial  
$ sudo apt update  
$ sudo apt install cmake cmake-curses-gui
```

## Using the installer or the pre-compiled binaries: [cmake.org/download/](https://cmake.org/download/)

```
# download the last cmake package, e.g. cmake-x.y.z-linux-x86_64.sh  
$ sudo sh cmake-x.y.z-linux-x86_64.sh
```

## A Minimal Example

CMakeLists.txt:

```
project(my_project)           # project name

add_executable(program program.cpp) # compile command
```

```
# we are in the project root dir
$ mkdir build # 'build' dir is needed for isolating temporary files
$ cd build
$ cmake ..    # search for CMakeLists.txt directory
$ make        # makefile automatically generated
```

```
Scanning dependencies of target program
[100%] Building CXX object CMakeFiles/out_program.dir/program.cpp.o
Linking CXX executable program
[100%] Built target program
```

# Parameters and Message

CMakeLists.txt:

```
project(my_project)
add_executable(program program.cpp)

if (VAR)
    message("VAR is set, NUM is ${NUM}")
else()
    message(FATAL_ERROR "VAR is not set")
endif()
```

```
$ cmake ..
VAR is not set
$ cmake -DVAR=ON -DNUM=4 ..
VAR is set, NUM is 4
...
[100%] Built target program
```

# Language Properties

```
project(my_project
        DESCRIPTION "Hello World"
        HOMEPAGE_URL "github.com/"
        LANGUAGES    CXX)

cmake_minimum_required(VERSION 3.15)

set(CMAKE_CXX_STANDARD          14) # force C++14
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS        OFF) # no compiler extensions

add_executable(program ${PROJECT_SOURCE_DIR}/program.cpp) #$
# PROJECT_SOURCE_DIR is the root directory of the project
```

# Target Commands

```
add_executable(program) # also add_library(program)

target_include_directories(program
    PUBLIC include/
    PRIVATE src/)
# target_include_directories(program SYSTEM ...) for system headers

target_sources(program # best way for specifying
    PRIVATE src/program1.cpp # program sources and headers
    PRIVATE src/program2.cpp
    PUBLIC include/header.hpp)

target_compile_definitions(program PRIVATE MY_MACRO=ABCEF)

target_compile_options(program PRIVATE -g)

target_link_libraries(program PRIVATE boost_lib)

target_link_options(program PRIVATE -s)
```

# Build Types

```
project(my_project)                # project name
cmake_minimum_required(VERSION 3.15) # minimum version

add_executable(program program.cpp)

if (CMAKE_BUILD_TYPE STREQUAL "Debug") # "Debug" mode
                                        # cmake already adds "-g -O0"

    message("DEBUG mode")
    if (CMAKE_COMPILER_IS_GNUCXX)      # if compiler is gcc
        target_compile_options(program "-g3")
    endif()
elseif (CMAKE_BUILD_TYPE STREQUAL "Release") # "Release" mode
    message("RELEASE mode")                # cmake already adds "-O3 -DNDEBUG"
endif()
```

```
$ cmake -DCMAKE_BUILD_TYPE=Debug ..
```

# Custom Targets and File Managing

```
project(my_project)
add_executable(program)

add_custom_target(echo_target          # makefile target name
                  COMMAND echo "Hello" # real command
                  COMMENT "Echo target")

# find all .cpp file in src/ directory
file(GLOB_RECURSE SRCS ${PROJECT_SOURCE_DIR}/src/*.cpp)
# compile all *.cpp file
target_sources(program PRIVATE ${SRCS}) # prefer the explicit file list instead
```

```
$ cmake ..
$ make echo_target
```



## Local and Cached Variables

*Cached variables* can be reused across multiple runs, while *local variables* are only visible in a single run. Cached `FORCE` variables can be modified only after the initialization

```
project(my_project)

set(VAR1 "var1")                                # local variable
set(VAR2 "var2" CACHE STRING "Description1")    # cached variable
set(VAR3 "var3" CACHE STRING "Description2" FORCE) # cached variable
option(OPT "This is an option" ON)              # boolean cached variable
                                                # same of var2

message(STATUS "${VAR1}, ${VAR2}, ${VAR3}, ${OPT}")
```

```
$ cmake .. # var1, var2, var3, ON
$ cmake -DVAR1=a -DVAR2=b -DVAR3=c -DOPT=d .. # var1, b, var3, d
```

# Manage Cached Variables

```
$ cmake . # or 'cmake-gui'
```

```
Page 1 of 1
CMAKE_BUILD_TYPE      Release
CMAKE_INSTALL_PREFIX  /usr/local
OPT                   ON
VAR2                  var2
VAR3                  var3

CMAKE_BUILD_TYPE: Choose the type of build, options are: None(CMAK
Press [enter] to edit option Press [d] to delete an entry
Press [c] to configure
Press [h] for help          Press [q] to quit without generating
Press [t] to toggle advanced mode (Currently Off)
```

# Find Packages

```
project(my_project)                # project name
cmake_minimum_required(VERSION 3.15) # minimum version

add_executable(program program.cpp)

find_package(Boost 1.36.0 REQUIRED) # compile only if Boost library
                                   # is found

if (Boost_FOUND)
    target_include_directories("${PROJECT_SOURCE_DIR}/include" PUBLIC ${Boost_INCLUDE_DIRS})
else()
    message(FATAL_ERROR "Boost Lib not found")
endif()
```

# Compile Commands

Generate JSON compilation database (`compile_commands.json`)

It contains the exact compiler calls for each file that are used by other tools

```
project(my_project)
cmake_minimum_required(VERSION 3.15)

set(CMAKE_EXPORT_COMPILE_COMMANDS ON) # <--

add_executable(program program.cpp)
```

Change the C/C++ compiler:

```
CC=clang CXX=clang++ cmake ..
```

CTest is a testing tool (integrated in CMake) that can be used to automate updating, configuring, building, testing, performing memory checking, performing coverage

```
project(my_project)
cmake_minimum_required(VERSION 3.5)
add_executable(program program.cpp)

enable_testing()

add_test(NAME Test1          # check if "program" returns 0
         WORKING_DIRECTORY ${PROJECT_SOURCE_DIR}/build
         COMMAND ./program <args>) # command can be anything

add_test(NAME Test2          # check if "program" print "Correct"
         WORKING_DIRECTORY ${PROJECT_SOURCE_DIR}/build
         COMMAND ./program <args>)

set_tests_properties(Test2
                     PROPERTIES PASS_REGULAR_EXPRESSION "Correct")
```

Basic usage (call ctest):

```
$ make test      # run all tests
```

ctest usage:

```
$ ctest -R Python      # run all tests that contains 'Python' string
$ ctest -E Iron        # run all tests that not contain 'Iron' string
$ ctest -I 3,5         # run tests from 3 to 5
```

Each ctest command can be combined with other tools (e.g. valgrind)

## ctest with Different Compile Options

It is possible to combine a custom target with ctest to compile the same code with different compile options

```
add_custom_target(program-compile
    COMMAND mkdir -p test-release test-ubsan test-asan # create dirs
    COMMAND cmake .. -B test-release # -B change working dir
    COMMAND cmake .. -B test-ubsan -DUBSAN=ON
    COMMAND cmake .. -B test-asan -DASAN=ON
    COMMAND make -C test-release -j20 program # -C run make in a
    COMMAND make -C test-ubsan -j20 program # different dir
    COMMAND make -C test-asan -j20 program)

enable_testing()

add_test(NAME Program-Compile
    COMMAND make program-compile)
```



xmake is a cross-platform build utility based on Lua.

Compared with `makefile/CMakeLists.txt`, the configuration syntax is more concise and intuitive. It is very friendly to novices and can quickly get started in a short time. Let users focus more on actual project development

Comparison: `xmake` vs `cmake`



# Code

# Documentation

---

Doxygen is the de facto standard tool for generating documentation from annotated C++ sources

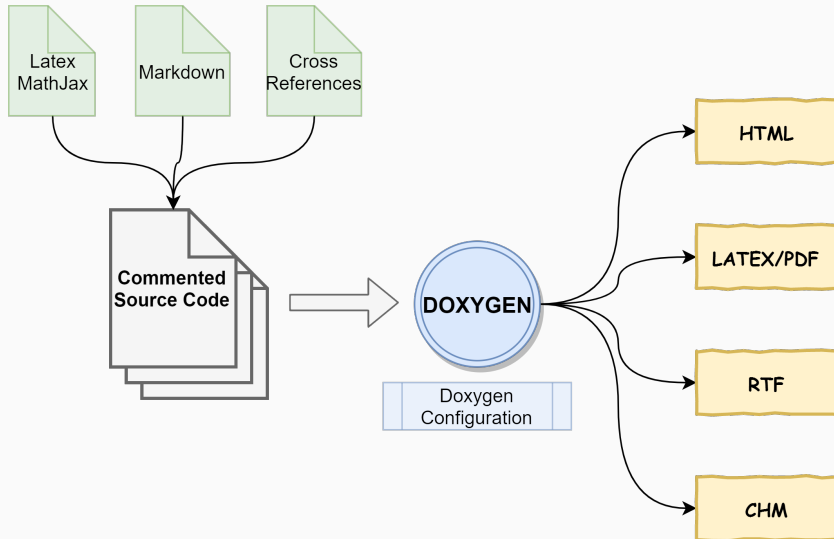
## Doxygen usage

- comment the code with `///` or `/** comment */`
- generate doxygen base configuration file

```
$ doxygen -g
```

- modify the configuration file Doxyfile
- generate the documentation

```
$ doxygen <config_file>
```



Doxygen requires the following tags for generating the documentation:

- `@file` Document a file
- `@brief` Brief description for an entity
- `@param` Run-time parameter description
- `@tparam` Template parameter description
- `@return` Return value description

- *Automatic cross references* between functions, variables, etc.
- *Specific highlight.* Code `<code>`, input/output parameters  
`@param[in] <param>`
- *Latex/MathJax* `$<code>$`
- *Markdown* ([Markdown Cheatsheet link](#)), Italic text `*<code>*`, bold text `**<code>**`, table, list, etc.
- Call/Hierarchy graph can be useful in large projects (requires graphviz)  
`HAVE_DOT = YES`  
`GRAPHICAL_HIERARCHY = YES`  
`CALL_GRAPH = YES`  
`CALLER_GRAPH = YES`

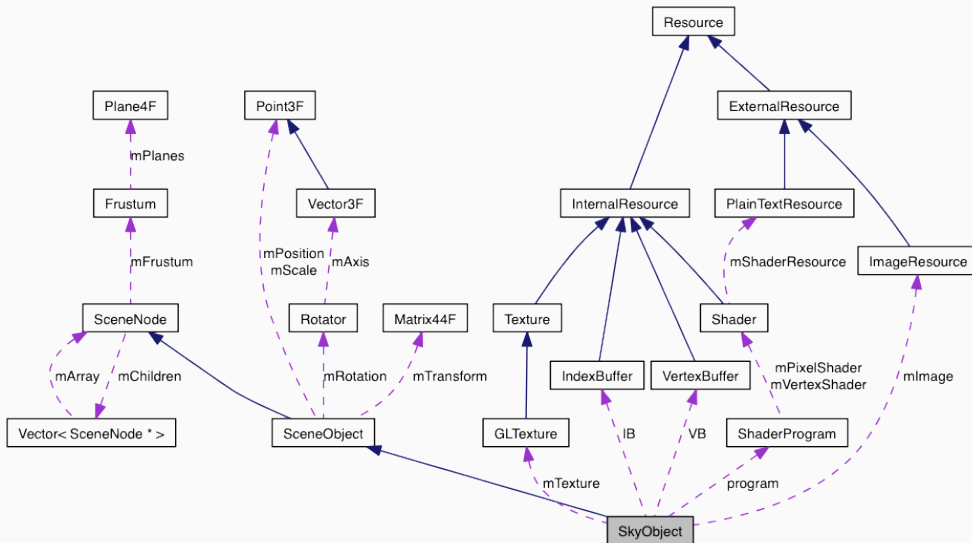
```
/**
 * @file
 * @copyright MyProject
 * license BSD3, Apache, MIT, etc.
 * @author MySelf
 * @version v3.14159265359
 * @date March, 2018
 */

/// @brief Namespace brief description
namespace my_namespace {

/// @brief "Class brief description"
/// @tparam R "Class template for"
template<typename R>
class A {
```

```
/**
 * @brief "What the function does?"
 * @details "Some additional details",
 *           $\sqrt{a}$ 
 * @tparam T Type of input and output
 * @param[in] input Input array
 * @param[out] output Output array
 * @return `true` if correct,
 *         `false` otherwise
 * @remark it is useful if ...
 * @warning the behavior is undefined if
 *          @p input is nullptr
 * @see related_function
 */
template<typename T>
bool my_function(const T* input, T* output);

/// @brief
void related_function();
```



# Doxygen Alternatives

`M.CSS` Doxygen C++ theme

`Doxypress` Doxygen fork

`clang-doc` LLVM tool

`Sphinx` Clear, Functional C++ Documentation with Sphinx + Breathe  
+ Doxygen + CMake

`standardese` The nextgen Doxygen for C++ (experimental)

`HDoc` The modern documentation tool for C++ (alpha)

`Adobe Hyde` Utility to facilitate documenting C++



# Code Statistics

---

# Count Lines of Code - cloc

cloc counts blank lines, comment lines, and physical lines of source code in many programming languages

```
$cloc my_project/
```

```
4076 text files.
```

```
3883 unique files.
```

```
1521 files ignored.
```

```
http://cloc.sourceforge.net v 1.50 T=12.0 s (209.2 files/s, 70472.1 lines/s)
```

Language	files	blank	comment	code
C	135	18718	22862	140483
C/C++ Header	147	7650	12093	44042
Bourne Shell	116	3402	5789	36882

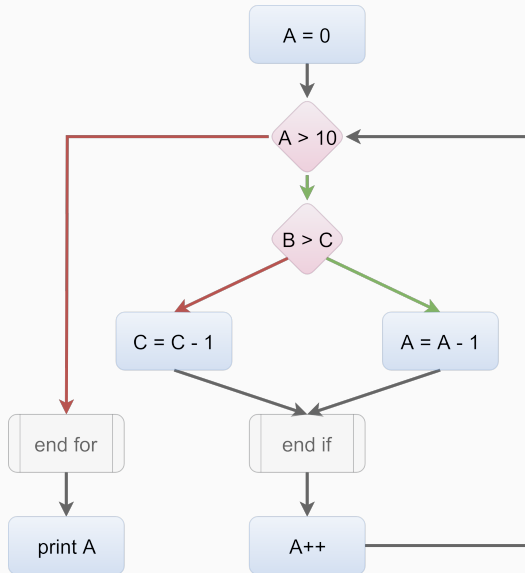
**Features:** filter by-file/language, SQL database, archive support, line count diff, etc.

Lizard is an extensible Cyclomatic Complexity Analyzer for many programming languages including C/C++

**Cyclomatic Complexity:** is a software metric used to indicate the complexity of a program. It is a quantitative measure of the number of linearly independent paths through a program source code

```
$lizard my_project/
=====
NLOC      CCN   token  param   function@line@file
-----
10         2     29      2     start_new_player@26@./html_game.c
6          1      3      0     set_shutdown_flag@449@./httpd.c
24         3     61      1     server_main@454@./httpd.c
-----
```

- CCN: cyclomatic complexity (should not exceed a threshold)
- NLOC: lines of code without comments
- token: Number of conditional statements



CCN = 3

CC	Risk Evaluation
1-10	a simple program, <i>without much risk</i>
11-20	more complex, <i>moderate risk</i>
21-50	complex, <i>high risk</i>
> 50	untestable program, <i>very high risk</i>

CC	Guidelines
1-5	The routine is probably fine
6-10	Start to think about ways to simplify the routine
> 10	Break part of the routine

Risk: Lizard: 15, OCLint: 10

- [www.microsoftpressstore.com/store/code-complete-9780735619678](http://www.microsoftpressstore.com/store/code-complete-9780735619678)
- [blog.feabhas.com/2018/07/code-quality-cyclomatic-complexity](http://blog.feabhas.com/2018/07/code-quality-cyclomatic-complexity)

# Other Tools

---

## Code Formatting - clang-format

clang-format is a tool to automatically format C/C++ code (and other languages)

```
$ clang-format <file/directory>
```

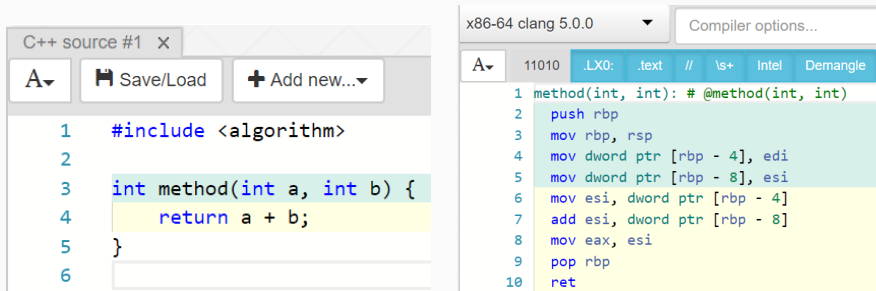
clang-format searches the configuration file .clang-format file located in the closest parent directory of the input file

clang-format example:

```
IndentWidth: 4
UseTab: Never
BreakBeforeBraces: Linux
ColumnLimit: 80
SortIncludes: true
```

# Compiler Explorer (assembly and execution)

Compiler Explorer is an interactive tool that lets you type source code and see assembly output, control flow graph, optimization hint, etc.



The screenshot displays the Compiler Explorer web application. On the left, a window titled 'C++ source #1' contains the following C++ code:

```
1  #include <algorithm>
2
3  int method(int a, int b) {
4      return a + b;
5  }
6
```

On the right, the assembly output for 'x86-64 clang 5.0.0' is shown. The assembly code is as follows:

```
1  method(int, int): # @method(int, int)
2      push rbp
3      mov rbp, rsp
4      mov dword ptr [rbp - 4], edi
5      mov dword ptr [rbp - 8], esi
6      mov esi, dword ptr [rbp - 4]
7      add esi, dword ptr [rbp - 8]
8      mov eax, esi
9      pop rbp
10     ret
```

**Key features:** support multiple architectures and compilers



# Code Transformation - CppInsights

CppInsights See what your compiler does behind the scenes

[About](#)

Source:

```
1 #include <cstdio>
2 #include <vector>
3
4 int main()
5 {
6     const char arr[10]{2,4,6,8};
7
8     for(const char& c : arr)
9     {
10         printf("c=%c\n", c);
11     }
12 }
```

Insight:

```
1 #include <cstdio>
2 #include <vector>
3
4 int main()
5 {
6     const char arr[10]{2,4,6,8};
7
8     {
9         auto&& __range1 = arr;
10         const char * __begin1 = __range1;
11         const char * __end1 = __range1 + 10L;
12
13         for( ; __begin1 != __end1; ++__begin1 )
14         {
15             const char & c = *__begin1;
16             printf("c=%c\n", static_cast<int>(c));
17         }
18     }
19 }
```

# Code Autocompletion - TabNine

TabNine uses deep learning to provide code completion

Features:

- Support all languages
- C++ semantic completion is available through clangd
- Project indexing
- Recognize common language patterns
- Use even the documentation to infer this function name, return type, and arguments

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

```
1 import os
2 import sys
3
4 # Count lines of code in the given directory, separated by file extension
5 def main(directory):
6     line_count = {}
7     for filename in os.listdir(directory):
8         _, ext = os.path.splitext(filename)
9         if ext not in line_count:
10             line_count[ext] = 0
11         for line in open(os.path.join(directory, filename)):
12             line_count[ext] += 1
13             line_count[ext] += 1          13%
14             line_count[ext]          Tab 20%
15             line_count[ext] +=       3 14%
16             line_count[ext].append(  4 3%
17             line                      5 23%
```

# Code Autocompletion - Kite

Kite adds AI powered code completions to your code editor

Support 13 languages

Available for Visual Studio Code, IntelliJ, Sublime, Atom, Vim, + others

```
1 import os
2 import sys
3
4 def count_py_files_in_repos(dirname):
5     if os.path.exists(os.path.join(dirname, '.git')):
6         count = 0
7         for root, dirs, files in os.walk(dirname):
8             count += len([f for f in files if f.endswith('.py')])
9             print('{} has {} Python files'.format(dirname, count))
10             format(dirname, count)
11             format(...)
12             format(<name>)
```

Ripgrep is a code-searching-oriented tool for regex pattern

## Features:

- Default recursively searches
- Skip .gitignore patterns, binary and hidden files/directories
- Windows, Linux, Mac OS support
- Up to 100x faster than GNU grep

```
[andrew@Cheetah rust] rg -i rustacean
src/doc/book/nightly-rust.md
92:[Mibbit][mibbit]. Click that link, and you'll be chatting with other Rustaceans

src/doc/book/glossary.md
3:Not every Rustacean has a background in systems programming, nor in computer

src/doc/book/getting-started.md
176:Rustaceans (a silly nickname we call ourselves) who can help us out. Other great
376:Cargo is Rust's build system and package manager, and Rustaceans use Cargo to

src/doc/book/guessing-game.md
444:it really easy to re-use libraries, and so Rustaceans tend to write smaller

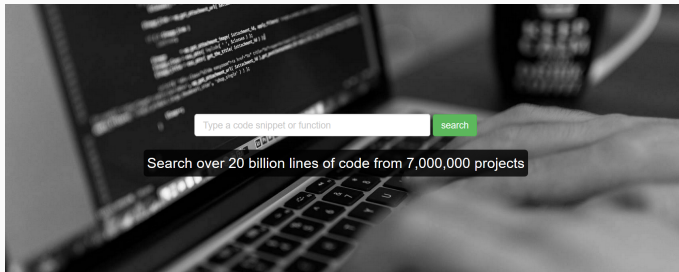
CONTRIBUTING.md
322:* [rustaceans.org][ro] is helpful, but mostly dedicated to IRC
333:[ro]: http://www.rustaceans.org/
[andrew@Cheetah rust] □
```

# Code Search Engine - searchcode

Searchcode is a free source code search engine

## Features:

- Search over 20 billion lines of code from 7,000,000 projects
- Search sources: github, bitbucket, gitlab, google code, sourceforge, etc.



`grep.app` searches across a half million GitHub repos

## // `grep.app`

Search across a half million git repos

☐

Case sensitive

☐

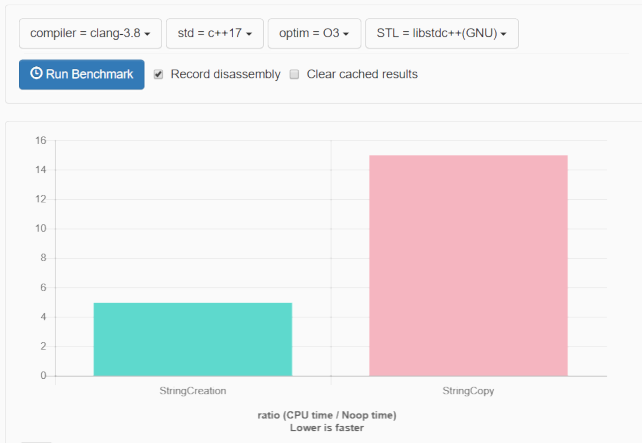
Regular expression

☐

Whole words

# Code Benchmarking - Quick-Bench

Quick-benchmark is a micro benchmarking tool intended to quickly and simply compare the performances of two or more code snippets. The benchmark runs on a pool of AWS machines



# Font for Coding

Many editors allow adding optimized fonts for programming which improve legibility and provide extra symbols (ligatures)

Scope	→ ⇒ :: __	-> => :: __
Equality	= ≡ ≠ ≠≡ == === ≠≡ ≠≡	== === != !=/ == === != !=
Comparisons	≤ ≥ ≤ ≥ ⇔	<= >= <= >= <=>

Some examples:

- JetBrains Mono
- Fira Code
- Microsoft Cascadia
- Consolas Ligaturized