Modern C++ Programming

12. C++ Ecosystem

Federico Busato

University of Verona, Dept. of Computer Science
2021, v3.08

Table of Context

Table of Context

Table of Context

Table of Context

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 Exploration - SourceTrail
- Code Benchmarking - Quick-Bench
- Font for Coding
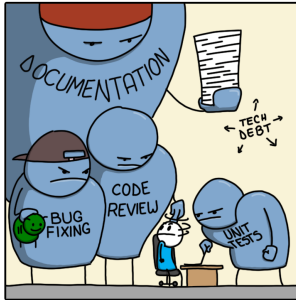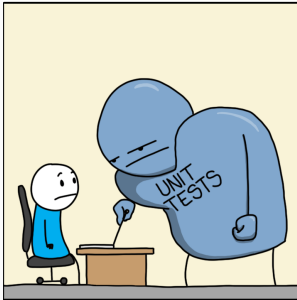
Debugging

Is this a bug?

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

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.

The common ways for handling <u>recoverable</u> errors are:

Exceptions Robust but slower and requires more resources. They also involve code bloat

Error values Fast but difficult to handle in complex programs

<u>Unrecoverable</u> 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 (gdb)

How to compile and run for debugging:

```
g++ -g [-ggdb3] <program.cpp> -o program
gdb [--args] ./program <args...>
```

-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

-ggdb3 Produces debugging information specifically intended for gdb
- the last number produces extra debugging information, for
  example: including macro definitions
- in general, it is not portable across different compiler
  (supported by gcc, clang)

gdb - **Breakpoints/Watchpoints**

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*>		delete a specific breakpoint
clear [*function_name/line_number*]		delete a specific breakpoint
enable/disable <*breakpoint_number*>		enable/disable a specific breakpoint
watch <*expression*>		stop execution when the value of expression changes (variable, comparison, etc.)

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

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	bt	prints stack backtrace (call stack)
backtrace *<full>*	bt	print values of local variables
help [*<command>*]	h	show help about command
info *<args/breakpoints/ watchpoints/registers/local>*		show information about program arguments/breakpoints/watchpoints/ registers/local variables

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

```
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-3.16.1.tar.bz2
$ tar xf valgrind-3.16.1.tar.bz2
$ cd valgrind-3.16.1
$ ./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) (test01.C:86)
==60127==    by 0x100000C22: main (test01.C: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) (test01.C:75)
==60127==    by 0x100000C22: main (test01.C: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 (a.c:5)
==19182==    by 0x80483AB: main (a.c: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
```

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...>
```

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
- etc.
- * Similar to `valgrind` but faster (2X slowdown)

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

`-O1` disable inlining

`-g` generate symbol table

Website:
clang.llvm.org/docs/AddressSanitizer.html
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>
```

Website:
  clang.llvm.org/docs/LeakSanitizer.html
  github.com/google/sanitizers/wiki/AddressSanitizerLeakSanitizer
  gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html

Memory Sanitizers

<u>Memory Sanitizer</u> 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

Website:
  clang.llvm.org/docs/MemorySanitizer.html
  github.com/google/sanitizers/wiki/MemorySanitizer
  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

Website:
  clang.llvm.org/docs/UndefinedBehaviorSanitizer.html
  gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html

Debugging Summary

How to Debug Common Errors

Segmentation fault

- gdb
- valgrind
- Segmentation fault when just entered in a function $\rightarrow$ stack overflow

Double free or corruption

- gdb
- valgrind

Infinite execution

- gdb + (CTRL + C)

Incorrect results

- valgrind + assertion + gdb + UndefinedBehaviorSanitizer

Demangling

Name mangling is a technique used to solve various problems caused by the need to resolve <u>unique names</u>

Transforming C++ ABI (Application binary interface) identifiers into the original source identifiers is called **demangling**

Example (linking error):

```
_ZNSt13basic_filebufIcSt11char_traitsIcEED1Ev
```

After demangling:

```
std::basic_filebuf<char, std::char_traits<char> >::~basic_filebuf()
```

How to demangle:

- `make |& c++filt | grep -P '`.*(?=))'`

- Online Demangler: https://demangler.com

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 ($\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

GCC Warnings

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

```
-Wcast-align                      -Wold-style-cast
-Wcast-qual                       -Wpragmas
-Wconversion                      -Wredundant-decls
# -Wfloat-conversion              -Wshadow
# -Wsign-conversion               -Wsign-promo*
-Wdate-time                       -Wstrict-aliasing
-Wdouble-promotion                -Wstrict-overflow=1  # 5
-Weffc++                          -Wswitch-bool
# -Wdelete-non-virtual-dtor       # -Wswitch-default
# -Wnon-virtual-dtor              # -Wswitch-enum
-Wformat-signedness               -Wtrampolines
-Winvalid-pch                     -Wunused-macros
-Wlogical-op                      -Wuseless-cast
-Wmissing-declarations            -Wvla
-Wmissing-include-dirs            -Wformat=2
-Wodr                             -Wno-long-long
```

gcc.gnu.org/onlinedocs/gcc/Warning-Options.html

github.com/barro/compiler-warnings

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)

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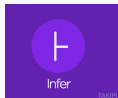
`FBInfer` is a static analysis tool (also available online) to checks for null pointer deferencing, memory leak, coding conventions, unavailable APIs, etc.

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

Static Analyzers - `DeepCode, SonarSource`

`DEEPCODE` **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

Code Testing

Bugs found (vertical axis)

Time spent testing software (horizontal axis)

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

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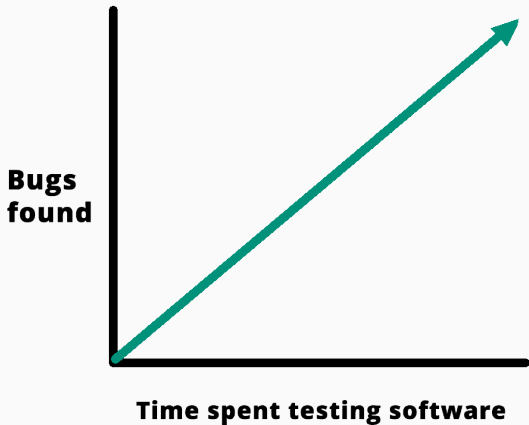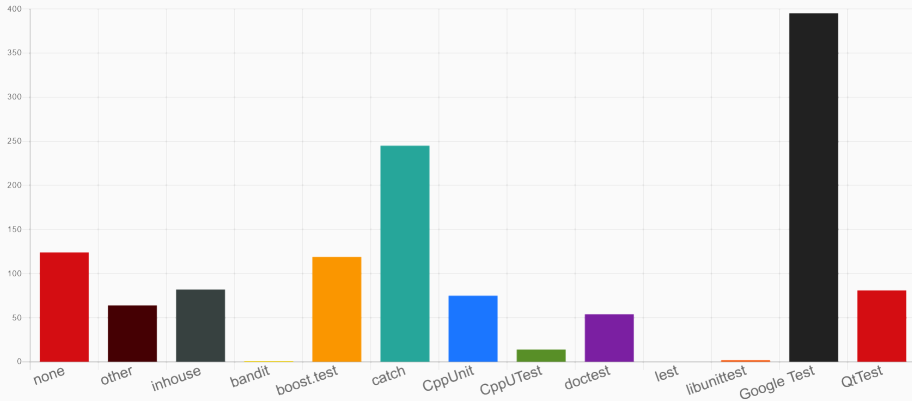$./test_program [<TestName>]
```

Other commands:
github.com/catchorg/Catch2

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

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

float floatComputation() { ... }

"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 );
}

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

`lcov` is a graphical front-end for gcov. It collects gcov data for multiple source files and creates HTML pages containing the source code annotated with coverage information

Step for code coverage:

- compile with `--coverage` flag (objects + linking)
- run the test
- visualize the results with gcov or 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 --std=c++11 --coverage program.cpp -o program
$./program 9
first
$gcov program.cpp
File 'program.cpp'
Lines executed:85.71% of 7
Creating 'program.cpp.gcov'
$lcov --capture --directory . --output-file coverage.info
$genhtml coverage.info --output-directory out
```

`program.cpp.gcov:`

```
   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:}
```

`lcov output:`

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:

```c
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 Overview

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/Ninjia and workspaces that can be used in the compiler environment of your choice

CMake features:

- Turing complete language
- Multi-platform (Windows, Linux, etc.)
- Open-Source
- Generate: makefiles, ninja, etc.
- Supported by many IDE: Visual Studio, Eclipse, etc.

CMake - References

- 19 reasons why CMake is actually awesome

- An Introduction to Modern CMake

- Effective Modern CMake

- Awesome CMake

- Useful Variables

CMakeLists.txt minimal example:

```
project(my_project)              # project name

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

```
$ cmake .      # 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
```

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

set(CMAKE_CXX_STANDARD          14) # force C++14
set(CMAKE_CXX_STANDARD_REQUIRED  ON)
set(CMAKE_CXX_EXTENSIONS        OFF)

add_executable(program)

# indicate include directory
target_include_directories(program
    PUBLIC "${PROJECT_SOURCE_DIR}/include")

# 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})
```

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

add_executable(program)

if (CMAKE_BUILD_TYPE STREQUAL "Debug")       # "Debug" mode
    target_compile_options(program "-g")
    target_compile_options(program "-O1")
    if (CMAKE_COMPILER_IS_GNUCXX)            # if compiler is gcc
        target_compile_options(program "-ggdb3")
    endif()
elseif (CMAKE_BUILD_TYPE STREQUAL "Release") # "Release" mode
    target_compile_options(program "-O2")
endif()

target_sources(program PRIVATE program.cpp)
```

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

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

add_custom_target(rm                      # makefile target name
                 COMMAND rm -rf *.o       # real command
                 COMMENT "Clear build directory")

add_executable(program)
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()
target_sources(program PRIVATE program.cpp)
```

```
$ cmake .
$ make rm
```

Generate JSON compilation database (compile_commands.json)

It contains the exact compiler calls for each file (used by other tools)

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

set(CMAKE_EXPORT_COMPILE_COMMANDS ON)      # <--

add_executable(program)
target_sources(program PRIVATE program.cpp)
```

Change the compiler:

```
CC=gcc CXX=g++ 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)

CMake Alternatives - `xmake`

`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 doxygen.cfg

- generate the documentation

  ```
  $doxygen <config_file>
  ```

An alternative in early development stage is `clang-doc`   57/79
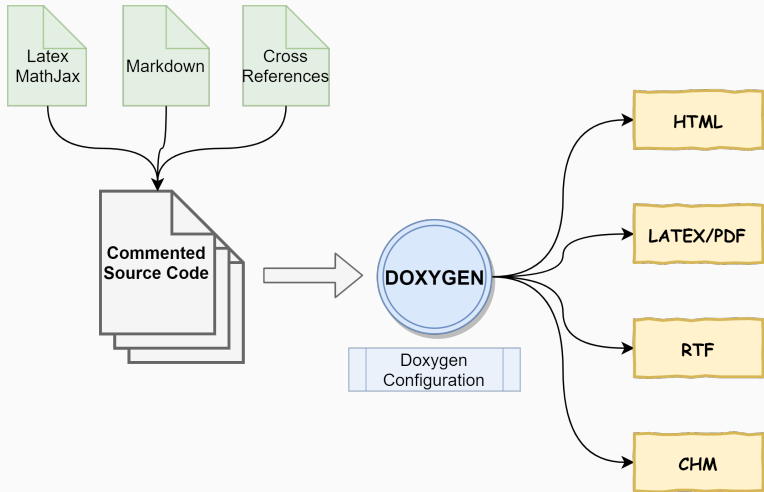
Doxygen provides support for:

- **Latex/MathJax** Insert latex math `$<code>$`

- **Markdown** (Markdown Cheatsheet link) Italic text `*<code>*`, bold text `**<code>**`, table, list, etc.

- **Automatic cross references** Between functions, variables, etc.

- **Specific highlight** Code `` `<code>` ``, parameter `@param <param>`

Doxygen guidelines:

- Include in every file **copyright**, **author**, **date**, **version**

- Comment `namespaces` and `classes`

- Comment `template` parameters

- Distinguish `input` and `output` parameters

- Call/Hierarchy graph can be useful in large projects
  (should include `graphviz`)
  `HAVE_DOT = YES`
  `GRAPHICAL_HIERARCHY = YES`
  `CALL_GRAPH = YES`
  `CALLER_GRAPH = YES`

μOS++ Doxygen style guide link

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

/// @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",
 *          Latex/MathJax: $\sqrt a$
 * @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;
```

61/79

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++

Teaching the art of great documentation, by Google

Code Statistics

Count Lines of Code - `cloc`

<u>`cloc`</u> 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.

<u>Lizard</u> 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
- `param`: Parameter count of functions

`CCN = 3`
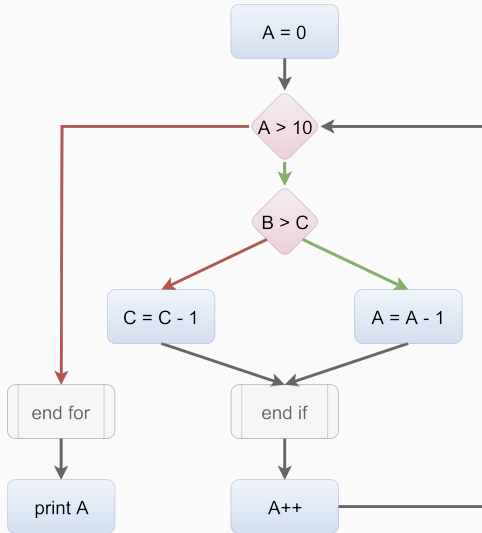
CC	Risk Evaluation
1-10	a simple program, *without much risk*
11-20	more complex, *moderate risk*
21-50	complex, *high risk*
> 50	untestable program, *very high risk*

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

References:
www.microsoftpressstore.com/store/code-complete-9780735619678
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>
```
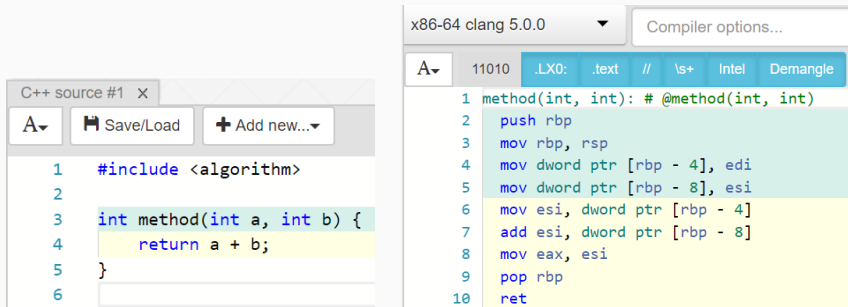
clang-format searches the configuration file `.clang-format` file
located in the closest parent directory of the input file

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.

Key feature: support multiple architectures and compilers

CppInsights See what your compiler does behind the scenes

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 }
```

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

```python
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%
```

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

Local Code Search - `ripgrep`

`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.

Type a code snippet or function   search

Search over 20 billion lines of code from 7,000,000 projects
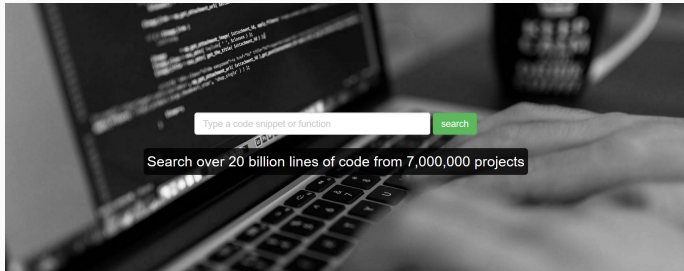
grep.app searches across a half million GitHub repos

// grep.app

Search across a half million git repos

🔍 Search

☐ Case sensitive   ☐ Regular expression   ☐ Whole words

Sourcetrail is an interactive code explorer that simplifies
navigation in complex source code

<u>**`Quick-benchmark`**</u> 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
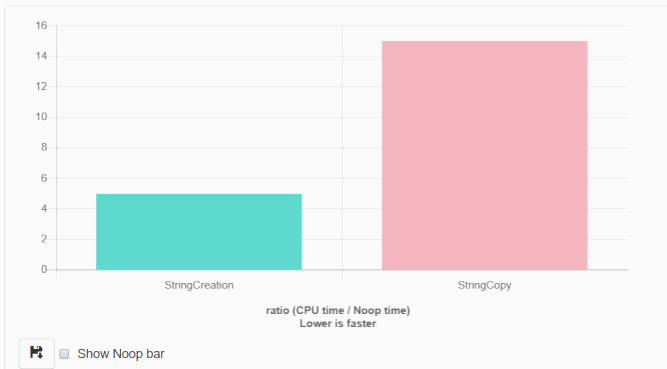
Font for Coding

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

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

Some examples:

- `JetBrain Mono`
- `Fira Code`
- `Microsoft Cascadia`
- `Consolas Ligaturized`