

Modern C++ Programming

11. DEBUGGING AND TOOLS

Federico Busato

University of Verona, Dept. of Computer Science
2018, v1.0



Agenda

- **Debugging**
 - Assertion
 - Execution debugging
 - Memory debugging
 - Clang sanitizer
 - Demangling
- **CMake**
- **Code Checking and Analysis**
 - Compiler warning
 - Static analyzer
- **Code Quality (Linter)**
- **Code Testing**
 - ctest
 - catch-lib
 - Code coverage
- **Code Commenting (Doxygen)**
- **Code Statistics**
 - Count lines of code
 - Cyclomatic complexity
- **Other Tools**
 - Code formatting
 - Assembly explorer

Debugging

Assertions

Assertions are intended to be used as a means of detecting programming bugs. Assertions represent an *invariant* in the code

Error/Exceptions can indicate “exceptional” conditions (invalid user input, missing files, etc.)

- **Exceptions** are more robust but slower
- **Error** are fast but difficult to handle in complex programs

```
#include <cassert> // <-- needed
int sqrt(int value) {
    int ret = sqrt_internal(value);
    assert(ret >= 0 && (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
```

How to compile for debugging:

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

-g Enable debugging

- stores the *symbol table information* in the executable
- 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)

How to run the debugger:

```
gdb --args ./program <args...>
```

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

| Command | Abbr. | Description |
|--|----------------|--|
| <code>run</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 <program_point></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 |

| Command | Abbr. | Description |
|---|-----------------|---|
| <code>list</code> | <code>l</code> | print code |
| <code>list <function or #start,#end></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 <full></code> | <code>bt</code> | print values of local variables |
| <code>help [<command>]</code> | <code>h</code> | show help about command |
| <code>info <args/breakpoints/ watchpoints/registers/local></code> | | show information about program arguments/breakpoints/watchpoints/ registers/local variables |

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

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

The debugger automatically stops when:

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

Full story:

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

[gdb reference card V5 link](#)



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

Website: <https://valgrind.org>

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 use 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...>
```

Address Sanitizer is a memory error detector (out-of-bounds, use-after-free, etc.). It relies on code instrumentation at compile-time. Similar to valgrind but faster (2X slowdown)

Website:

<https://clang.llvm.org/docs/AddressSanitizer.html>

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

`-O1` disable inlining

`-g` generate symbol table

Memory Sanitizer is detector of uninitialized reads. It relies on code instrumentation at compile-time. Similar to valgrind but faster (3X slowdown)

Website: <https://clang.llvm.org/docs/MemorySanitizer.html>

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

LeakSanitizer is a run-time memory leak detector. It relies on code instrumentation at compile-time. Similar to valgrind but faster

Website: clang.llvm.org/docs/LeakSanitizer.html

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

UndefinedBehaviorSanitizer is a undefined behavior detector (signed integer overflow, enumerated not in range, etc.). It relies on code instrumentation at compile-time. Not included in valgrind

Website: clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

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


Demangling

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

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>

How to Debug Common Errors

Segmentation fault/assertion fail

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

Double free or corruption

- valgrind

Infinite execution

- gdb + (CTRL + C)

Incorrect results

- valgrind + assertion + gdb + UndefinedBehaviorSanitizer

CMake



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

Website: <https://cmake.org>

CMake is used to control the software compilation process using simple platform and compiler independent configuration files, and *generate* native makefiles 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.

CMakeLists.txt minimal example:

```
PROJECT(my_project)           # project name
CMAKE_MINIMUM_REQUIRED(VERSION 3.5) # minimum version

ADD_EXECUTABLE(out_program program.cpp) # compile command
```

```
$cmake .      # CMakeLists.txt directory
$make        # makefile automatically generated
Scanning dependencies of target out_program
[100%] Building CXX object CMakeFiles/out_program.dir/program.cpp.o
Linking CXX executable out_program
[100%] Built target out_program
```

```
PROJECT(my_project)                # project name
CMAKE_MINIMUM_REQUIRED(VERSION 3.5) # minimum version

# verify if a compiler flag is supported
CHECK_CXX_COMPILER_FLAG("-std=c++14" COMPILER_SUPPORTS_CXX14)
IF (COMPILER_SUPPORTS_CXX14)      # if statement
    ADD_COMPILE_OPTIONS("-std=c++14") # if supported add the flag
ELSE()                             # else statement
    # if not supported exit and print message
    MESSAGE(FATAL_ERROR "Compiler ${CMAKE_CXX_COMPILER} has no C++14"
            "support.")
ENDIF()                             # end if statement

# indicate include directory
INCLUDE_DIRECTORIES("${PROJECT_SOURCE_DIR}/include")
# find all .cpp file in src/ directory
FILE(GLOB_RECURSE SRCS "${PROJECT_SOURCE_DIR}/src/*.cpp")

ADD_EXECUTABLE(out_program ${SRCS}) # compile all *.cpp file
```

```
PROJECT(my_project)                # project name
CMAKE_MINIMUM_REQUIRED(VERSION 3.5) # minimum version

IF (CMAKE_BUILD_TYPE STREQUAL "Debug") # "Debug" mode
  ADD_COMPILE_OPTIONS("-g")
  ADD_COMPILE_OPTIONS("-O1")
  IF (CMAKE_COMPILER_IS_GNUCXX)        # if compiler is gcc
    ADD_COMPILE_OPTIONS("-ggdb3")
  ELSEIF (CMAKE_CXX_COMPILER_ID EQUAL "Clang") # if compiler is clang
    ADD_COMPILE_OPTIONS("-fsanitize=address")
    ADD_COMPILE_OPTIONS("-fno-omit-frame-pointer")
  ENDIF()
ELSEIF (CMAKE_BUILD_TYPE STREQUAL "Release") # "Release" mode
  ADD_COMPILE_OPTIONS("-O2")
ENDIF()

ADD_EXECUTABLE(out_program program.cpp)
```

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

```
PROJECT(my_project)           # project name
CMAKE_MINIMUM_REQUIRED(VERSION 3.5) # minimum version

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

IF (Boost_FOUND)
    INCLUDE_DIRECTORIES("${PROJECT_SOURCE_DIR}/include"
                        Boost_INCLUDE_DIRS) # automatic variable
ELSE()
    MESSAGE(FATAL_ERROR "Boost Lib not found")
ENDIF()

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

ADD_EXECUTABLE(out_program 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 3.5) # minimum version

SET(CMAKE_EXPORT_COMPILE_COMMANDS ON) # <--

ADD_EXECUTABLE(out_program program.cpp)
```

Change compiler:

```
CC=gcc CXX=g++ cmake .
```

Useful variables:

https://cmake.org/Wiki/CMake_Useful_Variables

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**

Full list: <https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html>

Which Clang Warning Is Generating This Message?

<https://fuckingclangwarnings.com>

Static Analyzer (clang static analyzer)



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

Website: <https://clang-analyzer.llvm.org>

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 Analyzer (cppcheck/oclint)

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)

Website: <http://cppcheck.sourceforge.net/>

```
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
```

Oclint is a tool for improving quality and reducing defects by inspecting C/C++ code and looking for potential problems

Website: <https://oclint.org>

```
cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=ON .  
oclint-json-compilation-database -enable-global-analysis  
        -i <includes_dirs>
```

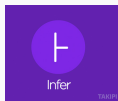
Static Analyzer (PVS-Studio/FBIinfer)



PVS-Studio is a high-quality *proprietary* (free for students) static code analyzer supporting C, C++

Website: <https://www.viva64.com/en/pvs-studio>

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



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

Website: <http://fbinfer.com>

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

Code Quality

Linters (clang-tidy)

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

Website: <https://clang.llvm.org/extra/clang-tidy>

```
$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

Linters (clang-tidy)

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-*
```

Linters (vera++)

Vera++ is tool for verification and analysis of C++ source code. It is complementary to clang-tidy: It provides weaker checkers, more oriented to syntax, then semantic

- well-formed file names
- space rules
- variable names
- etc.

Website:

<https://bitbucket.org/verateam/vera/wiki/Home>

```
vera++ --rule <rule_list> <src_file/include_file>
```

```
vera++ --profile <profile_name> <src_file/include_file>
```

Code Testing

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)

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

Alternatives: Google Test, Boost.Test, CppUnit, etc.

Website: <https://catch-lib.net>

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:

<https://github.com/catchorg/Catch2/blob/master/docs/command-line.md>

```
#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:

```
#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

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 |  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 | : #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: } |

Code Commenting

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

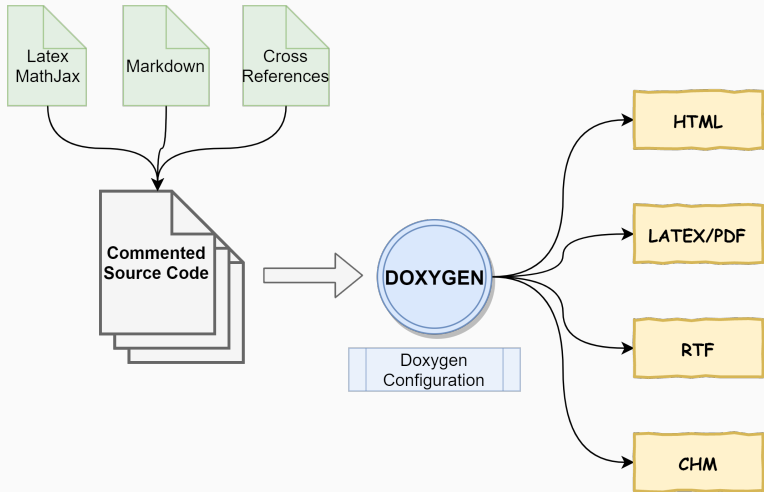
Doxygen usage

- 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>
```



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](#)

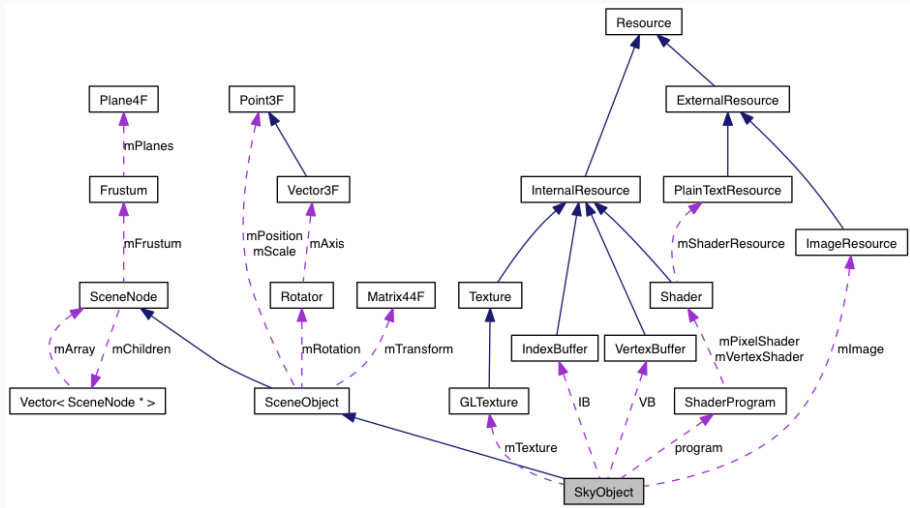
```
/**
 * @copyright MyProject
 * license BSD3, Apache, MIT, etc.
 * @author MySelf
 * @version v3.14159265359
 * @date March, 2018
 * @file
 */

/// @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>
int my_function(const T* input, T* output);

/// @brief
void related_function;
```

Code Statistics

Count Lines of Code (cloc)

Website: <https://cloc.sourceforge.net>

```
$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.

Cyclomatic Complexity Analyzer (lyzard)

Website: <https://github.com/terryyin/lyzard>

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's source code

```
$lyzard 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

Other Tools

Code Formatting (clang-format)

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

Website: clang.llvm.org/docs/ClangFormat.html

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

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

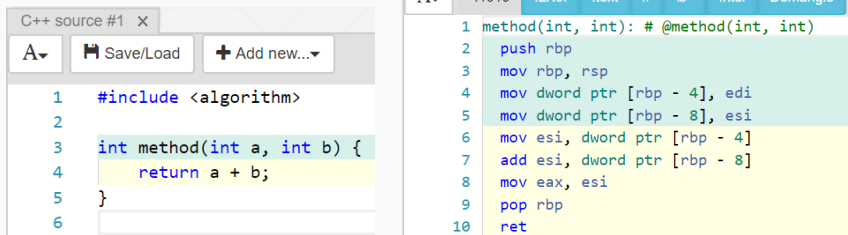
clang-format example:

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

Assembly Explorer

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

Website: <https://godbolt.org>



The screenshot displays the Compiler Explorer interface. On the left, a window titled "C++ source #1" shows 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:

```
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 feature: support multiple architectures and compilers

CppInsights See what your compiler does behind the scenes

Website: <https://cppinsights.io/>



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 }
```

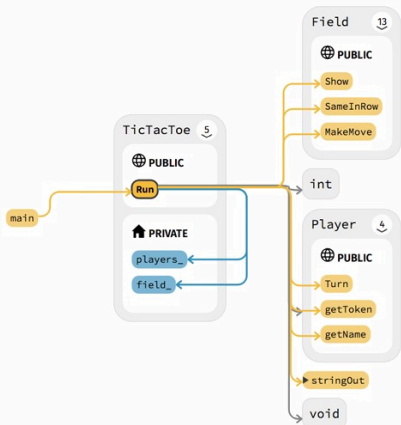
Insight:

```
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 + 10;
12
13        for( ; __begin1 != __end1; ++__begin1 )
14        {
15            const char & c = *__begin1;
16            printf("c=%c\n", static_cast<int>(c));
17        }
18    }
19 }
```


SourceTrail

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

Website: <https://www.sourcetrail.com/#intro>



```
.. TicTacToe::Start
32     return true;
33 }
34
35 void TicTacToe::Run() {
36     field_.Show();
37
38     int playerIndex = 0;
39
40     for ( int i = 0; i < 9; i++ ) {
41         Player& player = *players_[playerIndex];
42
43         field_.MakeMove( player.Turn( field_ ),
44             field_.Show());
45
46         if ( field_.SameInRow( player.getToken() )
47             io::stringOut( player.getName());
48             io::stringOut( " won!\n\n");
49             return;
50         }
51
52         playerIndex = ( playerIndex + 1 ) % 2;
53     }
54
55     io::stringOut( "Game ends in draw!\n\n");
56 }
57
58 void TicTacToe::Reset() {
```