OCLint Documentation

Release 0.6

Longyi Qi

January 11, 2013

CONTENTS

1 Overview **1**
- 1.1 Features . . . 1
- 1.2 Current Status . . . 1
- 1.3 Get Started . . . 1
- 1.4 Get Involved . . . 2

2 Introduction **3**
- 2.1 Download . . . 3
- 2.2 Building OCLint . . . 4
- 2.3 Installation . . . 5
- 2.4 Tutorial . . . 5

3 Command Line Usage **9**
- 3.1 Using oclint . . . 9
- 3.2 Using oclint-json-compilation-database . . . 11
- 3.3 Using oclint-xcodebuild . . . 14

4 Customizing OCLint **17**
- 4.1 Customizing Rules . . . 17
- 4.2 Customizing Reports . . . 18

5 Rule Index **19**
- 5.1 Basic . . . 19
- 5.2 Convention . . . 19
- 5.3 Empty . . . 20
- 5.4 Redundant . . . 20
- 5.5 Size . . . 20
- 5.6 Unused . . . 20

Bibliography **21**

CHAPTER

ONE

OVERVIEW

OCLint is a static code analysis tool for improving quality and reducing defects by inspecting C, C++ and Objective-C code and looking for potential problems like possible bugs, unused code, complicated code, redundant code, code smells, bad practices, and so on.

1.1 Features

Static code analysis is a critical technique to detect defects that aren't visible to compilers. OCLint automates this inspection process with advanced features:

- Relying on the abstract syntax tree of the source code for better accuracy and efficiency; False positives are mostly reduced to avoid useful results sinking in them.
- Dynamically loading rules into system, even in the runtime.
- Flexible and extensible configurations ensure users in customizing the behaviors of the tool.
- Command line invocation facilitates continuous integration and continuous inspection of the code while being developed, so that technical debts can be fixed early on to reduce the maintenance cost.

1.2 Current Status

OCLint is far from finished but is being continuously improved in many aspects, e.g. accuracy, performance and usability.

OCLint started as a research project (defense poster) at the University of Houston. Since then, OCLint has been rewritten and grown to be a 100% open source project. OCLint is designed to serve both academia and industry. Our goal is to make it a must-have tool for developers who program in C, C++ and Objective-C languages.

OCLint is compatible with Mac OS X and various Linux distributions.

1.3 Get Started

There is a large change that pre-compiled binaries are ready for you to download. It's also recommended to start with getting the release code and building it. You can then install and run OCLint.

1.4 Get Involved

Please also consider getting involved in the OCLint community. All kinds of contributions, like giving feedback, start a discussion, filing bug reports, request new features, best of all, submitting a patch or two, are appreciated.

CHAPTER

TWO

INTRODUCTION

2.1 Download

All release downloads can be found at OCLint website download page.

2.1.1 Choosing the Right Version

Development We keep our codebase stable by running continuous integrations on various platforms. Thus, it's safe to use the latest development version and get access to new features. However, certain things are under development and may not meet release standard.

Stable Latest official release is the most stable version, and meanwhile, is easier to get support from the entire community.

Previous Previous release is the second latest stable version, you can still get support from us. But pay attention, it will be archived when next version releases. In addition, deferring upgrading your development toolset is a type of technical debt that you should avoid.

Unsupported Choosing archived versions without supports.

This documentation is updated for OCLint version 0.6.

2.1.2 Choosing Pre-Compiled Binaries

Pre-compiled binaries are considered to ease you get started. OCLint binaries depend on very little and fundamental system standard libraries. Each should properly works on the platform it specifies. However, there are rare cases that these binaries do not suit you well. If the compiled binary doesn't work on your machine, consider building OCLint locally.

When download process is finished, see installation document for next step.

2.1.3 Choosing Source Code

All source code releases contain every module that is necessary for compilation and execution.

Different from development codebase, in which all dependencies are caught up to the latest version and debug flags are enabled, in released codebase, everything is finalized and optimized for that particular version.

Read how to build OCLint when you have source code on hand.

2.2 Building OCLint

This page presents you a shortcut of building OCLint release version.

2.2.1 System Requirements

1. See LLVM System Requirements
2. Apache Subversion
3. CMake

2.2.2 Prepare LLVM/Clang

1. **Checkout LLVM/Clang**
   - `cd oclint-scripts`
   - `./checkoutLLVMClang.sh`
   - `cd ..`
2. **Build LLVM/Clang**
   - `cd oclint-scripts`
   - `./buildClang.sh release`
   - `cd ..`

2.2.3 Build OCLint

1. `cd oclint-scripts`
2. `./buildCore.sh`
3. `./buildMetrics.sh`
4. `./buildRules.sh`
5. `./buildRelease.sh clean` (optional)
6. `./buildRelease.sh`
7. `cd ..`

2.2.4 Verify Your Build

1. `cd build/oclint-release/bin`
2. `./oclint`

You should get:

```
$ ./oclint
oclint: Not enough positional command line arguments specified!
Must specify at least 1 positional arguments: See: ./oclint -help
```

Now, let's move onto installation.

2.3 Installation

No matter you have downloaded a pre-compiled binary distribution or have built OCLint from scratch by yourself, now you should have an OCLint release with which file tree similar to this:

```
oclint-release
|-bin
|-lib
|---clang
|-----3.2
|-------include
|-------lib
|---oclint
|-----rules
```

In fact, you can execute `oclint` from `bin` directory now.

In order to easily invoke all OCLint executables, it's recommended to install OCLint to `PATH`, the environment variable that tells system which directories to search for executable files.

2.3.1 Option 1: Directly Add to PATH

You can add OCLint release folder directly to `PATH` by appending the following code block into your `.bashrc` or `.bash_profile` that is read when terminal launches.

```
OCLINT_HOME=/path/to/oclint-release
export PATH=$OCLINT_HOME/bin:$PATH
```

2.3.2 Option 2: Copy OCLint to System PATH

You can also copy OCLint to system `PATH`. There is an example that presumes `/usr/local/bin` is in the `PATH`.

1. `sudo cp bin/oclint* /usr/local/bin/` (require root permission)
2. `sudo cp -rp lib/* /usr/local/lib/` (require root permission)

Dependency libraries are required to be put into correct directory, because `oclint` executable searches `$(/path/to/bin/oclint)/../lib/clang` and `$(/path/to/bin/oclint)/../lib/oclint/rules` for required builtin headers and dynamic libraries.

2.3.3 Verify Installation

Open a new command line terminal, and simply execute `oclint`, you should get:

```
$ oclint
oclint: Not enough positional command line arguments specified!
Must specify at least 1 positional arguments: See: ./oclint -help
```

That's it – you can now move onto tutorial.

2.4 Tutorial

This tutorial walks you through the inspection of a piece of small but smelly C++ source code. By the end of this tutorial, you should be able to

- Apply OCLint on a single file
- Configure OCLint with very basic compiler flags
- Understand output

Throughout this tutorial, we will also lead you to the detail pages if you are interested in certain steps and are willing to know more.

2.4.1 Something Smells Here

Create a `sample.cpp` file with the content below:

```
int main() {
    int i = 0, j = 1;
    if (j) {
        if (i) {
            return 1;
            j = 0;
        }
    }
    else {
        // Do this later!
    }
    return 0;
}
```

2.4.2 Building Sample Code (Optional)

There is **no need** to build the code prior to run OCLint against it. However, since finding the correct arguments becomes one of the most frequently asked questions, this step is trying to help you convert your compiler flags to the ones that OCLint requires.

Note: This step, however, doesn't teach you how to find the correct compiler flags, thus, some level of knowledge about compiler flags is a prerequisite.

```
$ CC -c sample.cpp // step 1: compiling generates sample.o
$ CC -o sample sample.o // step 2: linking generates sample executable file

// Change CC to your favorite compiler that is GCC-compatible, e.g. g++ and clang++

$ ./sample // execute the binary
$ echo $? // output of a 0 means the code has been successfully built
```

We just took two sequential steps to generate the binary, step 1 compiles the code, and step 2 links. We are only interested in step 1 because that's all compiler flags you need to give to OCLint. Here in this case, the compiler flag is `-c`, and inspected source file is `sample.cpp`.

If you cannot pass through this step, don't give up, there are two helper programs oclint-json-compilation-database and oclint-xcodebuild (for Mac Xcode users) could help find the arguments for you.

2.4.3 Checking Single File

OCLint checks on single file with the following format:

```
oclint [options] <source> -- [compiler flags]
```

So, the command that applies to the sample source is

```
$ oclint sample.cpp -- -c
```

To change OCLint behavior, change the `[options]` before the source; to alter the compiler behavior, change the `[compiler flags]` after the `--` separator. A complicated example might look like this:

```
$ oclint -html -o report.html sample.cpp -- -D__STDC_CONSTANT_MACROS -D__STDC_LIMIT_MACROS -I/usr/inc
```

For detail about OCLint options and inspect multiple files, see oclint usage.

Some Thoughts

This approach works perfectly if you want to apply OCLint against one single file. The inspection process is quick, and making changes to arguments is easy.

When working on a project with a group of source files, you definitely prefer inspecting the entire project and having one report consists of all results. Well, if they share the same compiler flags, you can do

```
oclint [options]  <source0> [... <sourceN>] -- [compiler flags]
```

Now, each source file may have different compiler flags. In this case, OCLint uses the **compilation database** to know which source files to parse with what compiler flags. It can be considered as a condensed Makefile. So, you can do

```
oclint -p <build-path> [other options]  <source0> [... <sourceN>]
```

A more handy helper program that comes with OCLint is oclint-json-compilation-database.

In addition, if you are working on a Mac with Xcode as IDE, an experimental helper program oclint-xcodebuild is prepared for you.

2.4.4 Understanding Report

By applying OCLint against the above sample, we got the output like this:

```
Processing: /path/to/sample.cpp.
OCLint Report

Summary: TotalFiles=1 FilesWithViolations=1 P1=0 P2=2 P3=1

/path/to/sample.cpp:4:9: collapsible if statements P3
/path/to/sample.cpp:13:9: empty else block P2
/path/to/sample.cpp:9:17: dead code P2

[OCLint (http://oclint.org) v0.6]
```

Basically, you can find the following information in the report:

- Summary
  - total files
  - files with violations
  - number of priority 1 violations
  - number of priority 2 violations

    - number of priority 3 violations
- A list of violations
    - path to the source file
    - line number
    - column number
    - violated rule
    - priority
    - message (if any)
- OCLint information
    - website
    - release version

Read more about customizing reports.

We hope you have some feelings about OCLint, you can move on with a comprehensive usage guide. Also feel free to browse the rest content in this documentation for details, back to index or see table of contents. Thank you!

CHAPTER

THREE

COMMAND LINE USAGE

3.1 Using oclint

When you invoke OCLint, it normally does rule loading, compilation, analysis, and report generation. The options allow you to change the behavior of each step to certain ways that meet your requirement.

See all supported options in OCLint 0.6 by typing `oclint -help`:

```
USAGE: oclint [options] <source0> [... <sourceN>]

OPTIONS:
  -R=<directory>              - Add directory to rule loading path
  -fatal-assembler-warnings   - Consider warnings as error
  -help                       - Display available options (-help-hidden for more)
  -max-priority-1=<threshold> - The max allowed number of priority 1 violations
  -max-priority-2=<threshold> - The max allowed number of priority 2 violations
  -max-priority-3=<threshold> - The max allowed number of priority 3 violations
  -o=<path>                   - Write output to <path>
  -p=<string>                 - Build path
  -rc=<paramemter>=<value>    - Override the default baheviour of rules
  -stats                      - Enable statistics output from program
  Choose report type:
    -text                     - Plain text report
    -html                     - HTML formatted report
  -version                    - Display the version of this program

-p <build-path> is used to read a compile command database.

    For example, it can be a CMake build directory in which a file named
    compile_commands.json exists (use -DCMAKE_EXPORT_COMPILE_COMMANDS=ON
    CMake option to get this output). When no build path is specified,
    a search for compile_commands.json will be attempted through all
    parent paths of the first input file . See:
    http://clang.llvm.org/docs/HowToSetupToolingForLLVM.html for an
    example of setting up Clang Tooling on a source tree.

<source0> ... specify the paths of source files. These paths are
    looked up in the compile command database. If the path of a file is
    absolute, it needs to point into CMake's source tree. If the path is
    relative, the current working directory needs to be in the CMake
    source tree and the file must be in a subdirectory of the current
    working directory. "./" prefixes in the relative files will be
    automatically removed, but the rest of a relative path must be a
    suffix of a path in the compile command database.
```

```
For more information, please visit http://oclint.org
```

3.1.1 Rule Loading Options

-R <directory> Rule loading path can be changed by using `-R` option. Multiple rule loading paths can be specified to load rules from more than one directories. By default, OCLint searches `$(/path/to/bin/oclint)/../lib/oclint/rules` for the dyanmic libraries that contain rules.

-rc <paramemter>=<value> Certain rules have threshold to decide whether to emit violations. These thresholds can be changed by `-rc` option with a key-value pair.

More detail on changing the behavior in rules loading process during runtime can be found in customizing rules page.

3.1.2 Compilation Options

OCLint needs to know the specific compiler options for the sources that are being inspected. There are two alternatives, specifying the compiler options or using compile commands database.

Giving Compiler Options

Compiler options can be given directly to OCLint for compilation process. It's straight forward that, after all OCLint options and sources, append `--` separator followed by all compiler options:

```
oclint [oclint options] <source0> [... <sourceN>] -- [compiler options]
```

For example, if you are compiling a file named by following command:

```
clang -x objective-c -arch armv7 -std=gnu99 -fobjc-arc -O0 -isysroot /Applications/Xcode.app/Contents
```

(Wow, it's longer than expectation.)

Then when you analyze this code, your OCLint command will be:

```
oclint [oclint options] RPActivityIndicatorManager.m -- -x objective-c -arch armv7 -std=gnu99 -fobjc-
```

Compile Commands Database

-p <build-path> Choose the build directory in which a file named compile_commands.json exists. When no build path is specified, a search for compile_commands.json will be attempted through all parent paths of the first input file.

OCLint requires this compilation database to understand specific build options for each file. Currently it supports `compile_commands.json` file. See oclint-json-compilation-database for detail. If you are working with Xcode, oclint-xcodebuild can generate the required compile_database.json file for you with your little help.

3.1.3 Inspection Options

Of course, specify all the source files you want to inspect. Multiple files can be analyzed with one invocation.

3.1.4 Report Options

-o <path> Instead of piping output to console, `-o` will redirect the report to the <path> you specified.

-text Use plain text report, this is the default

-html Use HTML report for better readability

See customizing reports for detail.

3.1.5 Exit Status Options

-max-priority-1 <threshold> The max allowed number of priority 1 violations

-max-priority-2 <threshold> The max allowed number of priority 2 violations

-max-priority-3 <threshold> The max allowed number of priority 3 violations

This option helps in continuous integration and other build systems. When the number of violations in one of these priorities is larger than the maximum tolerance, OCLint will return with an exit status code other than 0 (code zero means normal termination) to notify a high volume of violations. By default, less than 20 priority 3 violations are allowed, 10 violations is maximum for priority 2, and no priority 1 violation can be tolerated. Too many violations result in bad code quality, if that happens, OCLint return with an exit code of 3.

OCLint returns with one of the four exit codes below

- **0** - SUCCESS
- **1** - RULE_NOT_FOUND
- **2** - ERROR_WHILE_PROCESSING
- **3** - VIOLATIONS_EXCEED_THRESHOLD

3.1.6 Other Options

-version Show version information about OCLint, LLVM and some environment variables.

3.2 Using oclint-json-compilation-database

OCLint needs a compilation database to figure out the compiler options for parsing each file, so that it can run more accurate analysis on an intermediate representation of the source code. This document provides instructions about generating `compile_commands.json` (JSON Compilation Database), and use `oclint-json-compilation-database` for code analysis.

3.2.1 JSON Compilation Database

A JSON Compilation Database, file name `compile_commands.json`, maintains a list of source code files with related build options. For each source file, working directory and command for compiling the source code are explicitly given. For example:

```
[
    {
      "directory": "/Projects/oclint/build",
      "command": "/Projects/llvm/bin/clang++   -D__STDC_LIMIT_MACROS -D__STDC_CONSTANT_MACROS  -O0 -g
      "file": "/Projects/oclint/main.cpp"
```

```
    },
    {
      "directory": "/Projects/oclint/build/impl/core",
      "command": "/Projects/llvm/bin/clang++   -D__STDC_LIMIT_MACROS -D__STDC_CONSTANT_MACROS  -O0 -g
      "file": "/Projects/oclint/impl/core/Violation.cpp"
    },
    {
      "directory": "/Projects/oclint/build/impl/core",
      "command": "/Projects/llvm/bin/clang++   -D__STDC_LIMIT_MACROS -D__STDC_CONSTANT_MACROS  -O0 -g
      "file": "/Projects/oclint/impl/core/ViolationSet.cpp"
    }
]
```

See JSON Compilation Database Format Specification with more precise defination.

3.2.2 Generating JSON Compilation Database

There are three approaches for generating JSON Compilation Database - writing your own, using CMake, and using OCLint xcodebuild helper program.

Writing Your Own

You can follow the format defined in JSON Compilation Database Format Specification, and write your own `compile_commands.json` file. It is convenient when you have a few sources to inspect.

You certainly need some tools' help if you have a large project.

Using CMake

CMake is a cross-platform build system. It can also help generate the required `compile_commands.json` compilation database.

Read CMake Documentation about how to use CMake as your build system.

You need to tell CMake that you are expecting `compile_commands.json` to be generated, so that when CMake converts its `CMakeLists.txt` to regular `Makefile`, it outputs `compile_commands.json` file for you along the way. To do this, add one extra option to you CMake command, for example:

```
cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=ON path/to/source-root
```

As a result, the `compile_commands.json` file is generated in current folder.

You can leave it here in the build directory, and use `-p` option for `oclint` to specify this file.

But in order to use `oclint-json-compilation-database`, it's required to copy or link this file to your source directory, for example:

```
ln -s `pwd`/compile_commands.json /path/to/source-root
```

Using OCLint xcodebuild Helper Program

If you are a Xcode user, then we have a helper program that extracts adequate compiler options and convert them into `compile_commands.json` file. Read how to use oclint-xcodebuild for instructions.

3.2.3 oclint-json-compilation-database Usage

Since you have `compile_commands.json` file copied or linked, you can now simply run `oclint-json-compilation-database` in your source directory. It reads the `compile_commands.json` file, get the list of all source files, and pass them to `oclint` for analysis.

See the usage by typing `oclint-json-complication-database -help`:

```
usage: oclint-json-compilation-database [-h] [-v] [-i INCLUDES] [-e EXCLUDES]
                                        [oclint_args [oclint_args ...]]

OCLint for JSON Compilation Database (compile_commands.json)

positional arguments:
  oclint_args           arguments that are passed to OCLint invocation

optional arguments:
  -h, --help            show this help message and exit
  -v                    show invocation command with arguments
  -i INCLUDES, -include INCLUDES, --include INCLUDES
                        extract files matching pattern
  -e EXCLUDES, -exclude EXCLUDES, --exclude EXCLUDES
                        remove files matching pattern
```

Filter Options

-i INCLUDES, -include INCLUDES, –include INCLUDES Extract files matching pattern from `compile_commands.json` or prior matching result

-e EXCLUDES, -exclude EXCLUDES, –exclude EXCLUDES Remove files matching pattern from `compile_commands.json` or prior matching result

Sometimes, you may be interested in a subset of entire codebase defined in `compile_commands.json`, and just want to inspect these sources. To do that, you can use filter options to get this subset. Since `oclint-json-compilation-database` is written in Python, so the matching pattern needs to follow Python regular expression syntax. In addition, multiple filters can be chained to get the file set you need for analysis.

OCLint Options

Remember there are many options that you can use to change the behavior of OCLint itself? Sure, you can ask `oclint-json-compilation-database` to pass through these options when it invokes `oclint` under the hook.

Since you have all compiler options in `compile_commands.json` file, so this time you don't need to tell `oclint` about them. But by following the same idea, now, these OCLint options can be given directly to `oclint-json-compilation-database` by appending `--` separator followed by all OCLint options:

```
oclint-json-compilation-database [<filter0> ... <filterN>] -- [oclint options]
```

Debug Options

-v show invocation command with arguments

Debug options are used for you to see the final `oclint` invocation command according to your settings of all filters and OCLint options. If you run the generated `oclint` command directly in the console, you should get the identical result as using `oclint-json-compilation-database`.

3.3 Using oclint-xcodebuild

OCLint recognizes a file called `compile_commands.json` to figure out the compiler options for parsing each file. By having a `compile_commands.json` file, analyzing projects with a large codebase becomes easier. Since all compiler options are implicitly configured in Xcode workspace/project settings, and you can see what actually happens when you invoke `xcodebuild` in terminal, this helper program then try to extract adequate compiler options, convert them into JSON Compilation Database format, and save it into `compile_commands.json` file.

See Also:

To gain some context about JSON Compilation Database and `compile_commands.json`, read using oclint-json-compilation-database.

3.3.1 Run xcodebuild

First of all, you need to help yourself a little bit. `xcodebuild` is a command line tool that Apple provides along with your Xcode application to let you build Xcode workspace/project in the terminal.

Read Apple's official xcodebuild Manual Page from Mac Developer Library is a good starting point if you haven't done any tasks about `xcodebuild`.

It is a quite simple task to some people by figuring out the correct options for `xcodebuild`. However, some people may feel it's not intuitive, so be patient, and take you time. You may find many online tutorials and blog posts that may help.

Okay, we believe you are quite confidence of running `xcodebuild` to build your Xcode project. Now, there are two things please pay attention:

- You need to save the `xcodebuild` output to a log file, by convention, name it `xcodebuild.log`. You can use `xcodebuild <options> | tee xcodebuild.log` to pipe every line of the output to `xcodebuild.log` file.
- If a source file has been built by Xcode, and it's not modified since last build, then it might not be built again when you invoke `xcodebuild`. In other words, if it happens, this file won't be shown in the log. So you won't see it in `compile_commands.json`. To avoid that, use clean build by removing all build products and intermediate files from the build directory. (Hope you understand what *clean build* means after spending the time on learning `xcodebuild`.)

3.3.2 Run oclint-xcodebuild

So a `xcodebuild.log` file should be there properly on the root folder of your Xcode project. Simply run

```
oclint-xcodebuild
```

The `compile_commands.json` will be generated to the same folder.

3.3.3 What's next

Now you have `compile_command.json` file for your Xcode project, you can move onto use oclint-json-compilation-database and use OCLint to inspect your codebase.

Note: `oclint-xcodebuild` is still an experimental project. The success of it depends various things, e.g. Mac OS X version, the Xcode version and project settings. However, since developers who use Xcode are familiar with Apple's manner of supporting only the latest version and one previous version, so `oclint-xcodebuild` tries to follow this convention. Your feedback is warmly welcome to help improve this helper program.

CHAPTER

FOUR

CUSTOMIZING OCLINT

4.1 Customizing Rules

OCLint is a rule based code analysis tool. Rule system is designed to be very extensible and flexible. So it's easy to customize rules in many ways, like making particular ruleset for certain projects, categorize them and load them from different locations, change the behavior of rules during runtime. You can even easily write your own custom rules, and plug them into OCLint system without rebuilding OCLint. You can find how to customize rules in this document, and how to custom rules in another page.

4.1.1 Selecting Rules for Inspection

By default, rules will be loaded from `$(/path/to/bin/oclint)/../lib/oclint/rules` directory, we name it **rule search path**. You can change the rule search path with `-R <directory>` option. Multiple locations can be specified, and rules in all these directories will be loaded.

Rule directory consists of a group of dynamic libraries, with extension `.so` in Linux and `.dylib` in Mac OS X. You can easily copy a subset of these rules into another folder, and use this new ruleset for particular projects. We call this *plug-and-use* rule loading mechanism - all dynamic libraries are loaded when OCLint searches through rule loading paths. So, when certain rules do not suite your project, simply remove them; and dragging and dropping new rules into the rule loading paths makes them immediate available for use.

4.1.2 Rule Thresholds

Certain rules emit violations only when the metrics of the source code exceed the thresholds. For example, according to [McCabe76], a reasonable cyclomatic complexity number for a method should be less than 10. So, default threshold in CyclomaticComplexityRule is set to 10. However, this value may not be the best for your project, you might either be okay with a more complicated codebase or you decide to push your team hard with a more restrict threshold. Then you can change this threshold by using `-rc CYCLOMATIC_COMPLEXITY=<new_value>` option.

Here is a list of all thresholds defined in OCLint 0.6:

CYCLOMATIC_COMPLEXITY Cyclomatic complexity of a method, default value is 10

LONG_CLASS Number of lines for a C++ class or Objective-C interface, category, protocol, and implementation, default value is 1000

LONG_LINE Number of characters for one line of code, default value is 100

LONG_METHOD Number of lines for a method or function, default value is 50

MINIMUM_CASES_IN_SWITCH Count of case statements in a switch statement, default value is 3

NPATH_COMPLEXITY NPath complexity of a method, default value is 200

NCSS_METHOD Number of non-commenting source statements of a method, default value is 30

NESTED_BLOCK_DEPTH Depth of a block or compound statement, default value is 5

4.2 Customizing Reports

OCLint currently supports plain text report and HTML report.

Report module will be extracted from core module to make it easier for developing custom report renderers. In addition, some report types are under development for largely used continuous integration system, and they are high priority tasks in the next release cycle (version 0.7).

4.2.1 Report Options

Usually you don't need to explicit specify `-text` flag as plain text report is the default. In order to see HTML report, add `-html` to `oclint` command.

Since browser is a good place to view HTML report. HTML report is better to be redirected to a file instead of console. It's easy to achieve this with `-o <path>` option.

4.2.2 Plain Text Report

Plain text report is designed for direct console output.

Report starts with the summary of the inspection, it consists of number of total files, number of files with violations, and number of violations of three priorities.

The output of each violation is formatted similar to compiler errors. Each line in the report indicates a violation. It starts with the file name, line, and column of the source code that the violation stands for. The name of the violated rule is printed after, followed by the priority of the rule. OCLint also outputs descriptions if current violation has any.

4.2.3 HTML Report

HTML reporter is browser friendly with better readability. The entire report follows the same order as the text report, but with a nicer structure of sections, paragraphs, and annotations. Violations are highlighted with different colors according to the priority.

CHAPTER

FIVE

RULE INDEX

OCLint 0.6 includes 42 rules.

5.1 Basic

- BitwiseOperatorInConditional
- BrokenOddnessCheck
- CollapsibleIfStatements
- ConstantConditionalOperator
- ConstantIfExpression
- DeadCode
- DoubleNegative
- ForLoopShouldBeWhileLoop
- GotoStatement
- MultipleUnaryOperator
- ReturnFromFinallyBlock
- ThrowExceptionFromFinallyBlock

5.2 Convention

- DefaultLabelNotLastInSwitchStatement
- InvertedLogic
- MissingBreakInSwitchStatement
- NonCaseLabelInSwitchStatement
- ParameterReassignment
- SwitchStatementsShouldHaveDefault
- TooFewBranchesInSwitchStatement

5.3 Empty

- EmptyCatchStatement
- EmptyDoWhileStatement
- EmptyElseBlock
- EmptyFinallyStatement
- EmptyForStatement
- EmptyIfStatement
- EmptySwitchStatement
- EmptyTryStatement
- EmptyWhileStatement

5.4 Redundant

- RedundantConditionalOperator
- RedundantIfStatement
- RedundantLocalVariable
- UnnecessaryElseStatement
- UselessParentheses

5.5 Size

- CyclomaticComplexity
- LongClass
- LongLine
- LongMethod
- NcssMethodCount
- NestedBlockDepth
- NPathComplexity

5.6 Unused

- UnusedLocalVariable
- UnusedMethodParameter

BIBLIOGRAPHY

[McCabe76] McCabe, T. J. (1976). A Complexity Measure. IEEE Transactions on Software Engineering, SE-2(4), 308-320. doi:10.1109/TSE.1976.233837