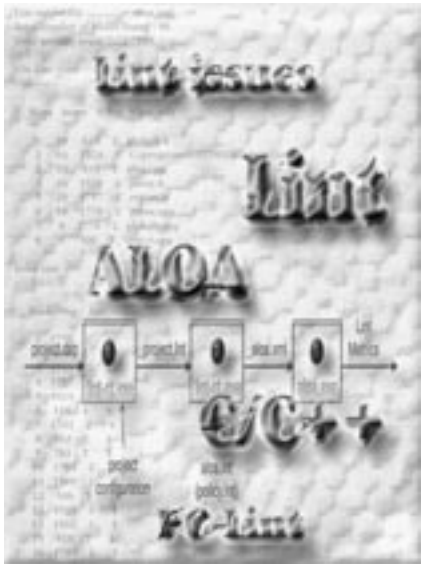


Ralf Holly

Lint Metrics & ALOA

Making a powerful tool even more powerful

Almost every seasoned C/C++ developer knows about Lint and the fact that it can help to greatly improve software quality. Lint statically analyzes your source code and generates detailed lists of warnings about potential bugs, classic mistakes, and portability pitfalls. By going through the list of Lint warnings (known as "Lint issues"), you can not only correct the problems Lint complains about, but by reviewing your own code, quite often find unrelated defects. This is yet more proof of the value of code reviews, even if they are "only" conducted by Lint and the author of the code.

Lint aims at finding bugs early in the development cycle, before testing starts. Bugs found at this stage are typically 5–10 times cheaper to fix than bugs found during system testing [1]. If used from the outset, a tool such as Lint would ideally help you achieve a constant warning count of 0. However, in most cases, Lint is introduced into ongoing projects, in which case, you can easily get drowned in a deluge of Lint warnings.

In such settings, project managers usually do not have enough resources to clean up all modules, nor do they want to risk making code changes late in the process [2]. In ongoing projects, project managers would normally rather get a Lint quality overview of the project—a compilation of the most frequently encountered Lint warnings, a sorted list of the most troublesome modules, and an overall "Lint score" that allows easy tracking of the code quality between builds. This is where ALOA comes into play.

Ralf Holly is the principal of PERA Software Solutions and can be contacted at rholly@pera-software.com.

Hawaiian Linting

ALOA (short for A Lint Output Analyzer) is a tool that processes output generated by PC-lint (Gimpel Software's Lint implementation; see http://www.gimpel.com/) and computes various useful metrics that give a quick overview of the internal quality of any C/C++ project. Furthermore, it shows which kind of Lint issues are most frequently encountered and highlights issue-laden modules. The metrics produced by ALOA are useful for tracking a project's Lint compliance and for fine-tuning Lint policies. The source code and related files for ALOA are available at http://www.cuj.com/code/.

ALOA takes the output from a project-wide Lint session and computes statistics and metrics like:

- Overall Lint score. This is a weighted sum of the severity of all Lint issues encountered [3]. This metric is particularly useful for monitoring how the Lint quality changes between builds.
- File list. A list of modules, sorted by the weighted severity sum of the Lint issues contained. The main purpose of this list is to highlight troublesome modules. Modules from the top of the list are typically reviewed and cleaned up first.
- Issue list. This list shows which Lint issues are encountered most frequently in a project. It is a great tool for fine tuning the Lint policy because it shows which programming constructs/styles are typical for a project and, hence, may be suppressed globally by disabling the corresponding warnings in the Lint policy file.

ALOA is free software, licensed under the terms of the GPL [4], and assumes that you have a PC-lint license. PC-lint is a powerful commercial version of Lint; however, I suppose that it is not too difficult to adapt ALOA to support other Lint implementations.

True to Microsoft's "eat your own dog food" approach, ALOA has been used on itself. Figure 1 is output from an earlier version. Today, of course, ALOA is free of Lint warnings at warning level 3.

Using ALOA

Again, ALOA needs the output from a project-wide Lint session as input. This multistaged process—step 1, obtaining project settings; step 2, running Lint; step 3, running ALOA—is driven by a batch file called "dsplinter.bat," which I wrote for Visual C++ projects (see Figure 2).

In the first step, dsplinter forwards a user-provided Visual C++ project file (.dsp) to PC-lint (lint-nt.exe). The output from this stage is a PC-lint indirect file (_project.lnt) that contains a list of all files that belong to this project as well as predefined preprocessor symbols, and includes path settings in a format that is recognized by PC-lint.

Next, _project.lnt is fed to PC-lint again, together with another indirect file called "aloa.lnt." aloa.lnt contains output formatting settings that ensure that PC-lint generates messages in a format that is understood by ALOA. aloa.lnt includes the project policy (policy.lnt) by reference. The project policy determines the PC-lint warning level; that is, it defines which warnings are reported and which warnings are suppressed. dsplinter assumes that the project policy is located in the same directory as the Visual C++ .dsp file. During the project-wide Lint session, all output is collected in a file named "_aloa.xml."

At step 3 (running ALOA), _aloa.xml is passed to ALOA, which generates the Lint metrics and prints them to `stdout`. Typically, the output is redirected to a file on the command line, such that the results can be put under revision control and compared with results from previous runs.

The command-line interface of dsplinter looks like this:

```
dsplinter <dspfile> (<config> | -default) [<file> | -aloa]
```

The first parameter is the fully qualified filename of the Visual C++ project that you want to Lint. Next comes the actual configuration within your .dsp file. You can either select a particular configuration (for example, "myproject - Win32 Release") or pass `-default` to select the default configuration. The third parameter specifies the mode of operation. If omitted, the whole project gets Linted and all original PC-lint messages go to `stdout`. If you pass a fully qualified name of a project member file instead, only this module gets Linted [5]. To run dsplinter in ALOA mode (the focus of this article), you have to pass `-aloa`.

The biggest benefit of using dsplinter to drive the process is that it extracts the Visual C++ project settings on the fly (step 1; obtaining project settings). Therefore, you don't need to maintain a separate _project.lnt manually in parallel to your .dsp file. This is possibly because PC-lint directly supports .dsp to .lnt conversion for Visual C++ projects. But even if you are not using PC-lint or you are not working on Visual C++ projects, it shouldn't be too difficult to write a Perl script that extracts project parameters from your IDE vendor's project file.

Implementation

ALOA's code was developed under Visual C++ 6.0 and comprises only 400 lines of uncommented source code. This leanness is possible because on the one hand ALOA doesn't come with a fancy GUI and on the other hand, STL offers most of what ALOA needs.

ALOA's prebuilt executable, the Visual C++ project file and source code, as well as dsplinter can be downloaded from http://www.pera-software.com/aloa.htm.

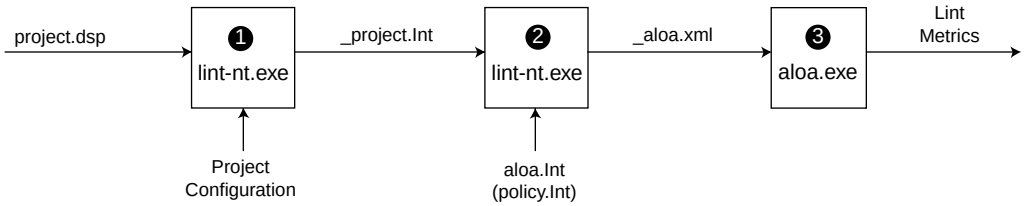
Two classes, `File` and `Issue`, are essential for understanding ALOA's design. `File` encapsulates a source-code module that has been Linted. `File`'s attributes are `m_filename` (the name of the file), `m_severityScore` (the weighted severity sum of this file), and `m_severestIssueNumber` (the issue number of the severest issue in this file).

As its name suggests, `Issue` represents a Lint issue. An `Issue` has an issue number (`m_number`), a severity value (`m_severity`), and a counter that tallies how many times this particular issue number has been encountered in the whole project.

Figure 1: ***Typical ALOA output.***

```
Lint output file ........... : _aloa.xml
Total number of issues found : 96
Total severity score ....... : 245

File List
---------

     Rank   Score   MMsg  MSev  File

       1      81     618    3   globals.h
       2      64    1024    4   C:\progs\msvc\VC98\Include\vector
       3      24     818    2   aloa.cpp
       4      24    1510    3   parse.h
       5      20    1717    2   report.h
       6      18    1776    2   parse.cpp
       7       8    1776    2   globals.cpp
       8       6     506    3   report.cpp

Issue List
----------

     Rank  Msg    Sev   Count

       1   1776    2     20
       2    618    3     15
       3   1712    2     12
       4   1717    2     10
       5   1024    4      8
       6    118    4      8
       7   1702    2      4
       8    762    2      4
       9    783    2      3
      10   1764    2      3
      11   1509    3      2
      12    506    3      2
      13   1510    3      1
      14   1512    3      1
      15    818    2      1
      16   1727    2      1
      17   1746    2      1

Legend
------

Severity level   1 : Elective note
Severity level   2 : Informational
Severity level   3 : Warning
Severity level   4 : Syntax error
Severity level 999 : PC-lint error

Score  Severity score
MMsg   Number of issue with the highest severity encountered
MSev   Severity level of the severest issue encountered
Msg    Lint issue number
Sev    Issue severity level
Count  Total number of occurrences of issue in the whole project
```

Figure 2: ***Diagram of the dsplinter.bat batch file.***



Both the File and Issue classes maintain a vector of pointers to each other. This cross referencing makes it fairly easy to answer questions such as, "What Issues are encountered in this File?" or "Which Files contain this Issue?" later on when generating metrics. Listing 1 is the definition of File and Issue.

Even though dsplinter (or, more precisely, aloa.lnt) instructs PC-lint to produce well-formed XML output that can be viewed with any XML viewer like XMLSpy, ALOA doesn't use a full-fledged XML parser. For simplicity, ALOA comes with a primitive, though efficient, parser that is only capable of extracting XML attribute values. Don't be surprised if you discover that it doesn't support DTDs, validation, XML comments, and the like.

After main (see Listing 2) has validated the command-line arguments, it invokes the parser and registers a callback function (onNewIssueHandler; see Listing 2). onNewIssueHandler is called back by the parser every time a new Lint issue is encountered. The arguments of onNewIssueHandler are the name of the file where the issue was found and the issue number.

The main job of onNewIssueHandler is to maintain an std::map of File objects (gFileMap) and an std::map of Issue objects (gIssueMap). The former uses the filename of the source-code module as a key, whereas the latter uses the Lint issue number:

```
typedef std::map<std::string, File*> FILE_MAP;
extern FILE_MAP gFileMap;

typedef std::map<int, Issue*> ISSUE_MAP;
extern ISSUE_MAP gIssueMap;
```

The first thing onNewIssueHandler does is update global information such as the project's total number of issues (gIssuesCount) and the total severity score (gSeverityScore).

Next, onNewIssueHandler checks whether this particular filename has already been registered with gFileMap. If "yes," the corresponding File object is retrieved from the map; otherwise, a new File object is created. The same thing happens with Issue objects. First they are checked as to whether a Lint Issue with the given issue number already exists in gIssueMap; if not, a new Issue object is created and stored in the map.

The last thing onNewIssueHandler does is cross-registration of File and Issue objects by calling File::addIssue and Issue::addFile, respectively. This happens regardless of how the File and Issue objects were obtained (either by creating them from scratch or by looking up existing objects in the maps).

Obtaining Metrics

Once the parsing phase is over, main invokes a function called buildMetricsLists (see Listing 2). Since most of the metrics have already been computed on the fly during the parsing phase, the only task left for this function is to create sorted lists of all Files and Issues encountered in the project.

Since it is not possible to sort the std::maps directly (they have their own key-dependent ordering), the contained File and Issue objects are first copied to dedicated std::vectors (gFileList and gIssueList) before calling std::sort. To be able to use sort, two global versions of operator<() need to be defined:

```
inline bool operator<(const File& lhs, const File&  rhs) {
    // Sort by severity score, then filename
    if (lhs.m_severityScore == rhs.m_severityScore)
        return lhs.m_filename < rhs.m_filename;
    return lhs.m_severityScore > rhs.m_severityScore;
}

inline bool operator<(const Issue& lhs, const Issue& rhs) {
    // Sort by count, then by severity
    if (lhs.m_count == rhs.m_count)
        return lhs.m_severity > rhs.m_severity;
    return lhs.m_count > rhs.m_count;
}
```

After the lists have been built up in memory, the only job left to do is present the metrics to the user. This is done by invoking reportMetrics.

Listing 1

```
// Encapsulates a lint issue (ie. warning, error)
class Issue {
public:
    Issue(int number, int severity) :
        m_number(number),
        m_severity(severity),
        m_count(0) {}
    // Register a file with this issue
    void addFile(const File* pFile) {
        assert(pFile != NULL);
        m_files.push_back(pFile);
        ++m_count;
    }
    int getNumber() const       { return m_number;   }
    int getSeverity() const     { return m_severity; }
    int getCount() const        { return m_count;    }
private:
    friend bool operator<(const Issue& lhs, const Issue& rhs);
    typedef std::vector<const File*> FileList;
    int m_number;          // Lint issue number
    int m_severity;        // The severity level of this lint issue
    int m_count;           // Total number of occurrences of this lint issue
    FileList m_files;     // List of all files that contain this lint issue
};
// Encapsulates a source code file with possibly many lint issues
class File {
public:
    File(const std::string& filename) :
        m_filename(filename),
        m_severityScore(0),
        m_severestIssueNumber(UNUSED_ISSUE_NUMBER) {}
    // Registers a lint issue with this file
    void addIssue(const Issue* pIssue) {
        assert(pIssue != NULL);
        m_issues.push_back(pIssue);
        int issueNumber = pIssue->getNumber();
        int severity = getSeverity(issueNumber);
        if (   m_severestIssueNumber == UNUSED_ISSUE_NUMBER
            || severity > getSeverity(m_severestIssueNumber) ) {
            m_severestIssueNumber = issueNumber;
        }
        m_severityScore += severity;
    }
    const std::string& getFilename() const   { return m_filename;           }
    int getSeverityScore() const             { return m_severityScore;      }
    int getSeverestIssueNumber() const       { return m_severestIssueNumber; }
private:
    friend bool operator<(const File& lhs, const File& rhs);
    typedef std::vector<const Issue*> IssueList;
    std::string m_filename;     // The name of this source code module
    int m_severityScore;        // The accumulated severity score
    int m_severestIssueNumber;  // The issue number with the highest severity
    IssueList m_issues;     // List of all Lint issues contained in this file
};
```

Listing 2

```
static void onNewIssueHandler(const char* pFilename, int number) {
    int severity = getSeverity(number);
    // Update global metrics
    ++gIssuesCount;
    gSeverityScore += severity;
    // Obtain file object
    string filename(pFilename);
    File* pFile = 0;
    FILE_MAP::iterator iterFile = gFileMap.find(filename);

    // If unknown filename, create new file object
    if (iterFile == gFileMap.end()) {
        pFile = new File(filename);
        bool wasInserted = gFileMap.insert(make_pair(filename, pFile)).second;
        assert(wasInserted);
    // If known filename, retrieve existing file object
    } else {
        pFile = (*iterFile).second;
    }
    // Obtain issue object
    Issue* pIssue = 0;
    ISSUE_MAP::iterator iterIssue = gIssueMap.find(number);
    // If unknown issue, create new issue object
    if (iterIssue == gIssueMap.end()) {
        pIssue = new Issue(number, severity);
        bool wasInserted = gIssueMap.insert(make_pair(number, pIssue)).second;
        assert(wasInserted);
    // If known issue, retrieve existing issue object
    } else {
        pIssue = (*iterIssue).second;
    }
    // Update file/issue metrics
    pFile->addIssue(pIssue);
    pIssue->addFile(pFile);
}
static void buildMetricsLists() {
    // Create sorted file list
    FILE_MAP::iterator iterFile = gFileMap.begin();
    for (; iterFile != gFileMap.end(); ++iterFile) {
        gFileList.push_back(*(*iterFile).second);
    }
    sort(gFileList.begin(), gFileList.end());
    // Create sorted issue list
    ISSUE_MAP::iterator iterIssue = gIssueMap.begin();
    for (; iterIssue != gIssueMap.end(); ++iterIssue) {
        gIssueList.push_back(*(*iterIssue).second);
    }
    sort(gIssueList.begin(), gIssueList.end());
}
int main(int argc, const char* const argv[]) {
    try {
        initGlobals();
        scanCommandLine(argc, argv);
        parseLintOutputFile(gpLintOutputFile, &onNewIssueHandler);
        buildMetricsLists();
        reportMetrics();
    } catch (const ParseFileNotFoundError& e) {
        reportFatalError("Cannot access " + e.getFilename());
    } catch (const ParseMalformedLineError& e) {
        ostringstream s;
        s << "Malformed file: " << e.getFilename() << ",
                                          line: " << e.getLineNo();
        reportFatalError(s.str());
    } catch (...) {
        reportFatalError("Unspecified fatal error");
    }
    return EXIT_SUCCESS;
}
```

Conclusion

ALOA is a simple yet powerful tool, particularly for introducing PC-lint to ongoing C/C++ projects. It quickly gives an overview of all encountered Lint issues, helps monitor a project's Lint quality over the course of time, and pinpoints troublesome modules.

I have successfully used ALOA on various projects, ranging from extremely resource-constrained embedded systems to GUI-based desktop systems. In all cases, PC-lint and ALOA have jointly helped reducing development costs.

It is impossible for me to quantify the benefits of using ALOA, but I'm convinced it is another—rather lightweight and inexpensive—step towards better software quality.

Acknowledgment

I would like to thank Peter Most for his support in reviewing this article and source code.

References

[1] McConnell, Steve. *Code Complete*, Microsoft Press, 1993.

[2] Obviously, most of what Lint warns about are not really bugs. Rather, Lint acts like a mentor and asks the developer questions such as, "Are you really sure about what you are doing?"

[3] Not all Lint issues are considered equally bad. For instance, ALOA differentiates between Lint fatal errors (999 severity points), syntax errors (4 severity points), warnings (3 severity points), informational messages (2 severity points), and elective notes (1 severity point).

[4] http://www.gnu.org/licenses/gpl.html/.

[5] With the help of dsplinter, you can easily integrate PC-lint with Visual Studio. This lets you Lint single modules from within Visual Studio and to quickly navigate between Lint messages. Refer to ALOA's readme file for details. ❑