Taming the Lint Monster

Part 2: Deconstructing the PC-Lint Command Line…and more

In part 1 we introduced the PC-Lint static code analysis tool, and started to look at how to configure it. In this part we will look closely at how the PC-Lint command line is formed, and some of the common PC-Lint options. We will also consider how to configure PC-Lint for a particular project configuration, and some strategies for dealing with the analysis results it generates.

First though, let's take a closer look at the command line we closed the first part with:

```
"C:\Data\PC Lint\8.00\lint-nt.exe" -i"C:\Data\PC Lint\8.00" -background -b --u
C:\Data\Code\Projects\Applications\SourceVersioner\Development\SourceVersioner_vs71_De
bug_Win32.lnt -u "C:\Data\PC Lint\8.00\std_vs71.lnt" env-vc7.lnt -t4 +ffb +linebuf
+linebuf -iC:\Data\Code\Projects\Applications\SourceVersioner\Development\Debug
c:\Data\Code\Projects\Applications\SourceVersioner\Development\Shared\FileUtils.cpp
```

If we break this command line down to its essentials, we are left with something a bit more comprehensible:

```
lint-nt.exe -i<PC-Lint folder> -background -b --u <project.lnt file> <std.lnt file> -u
env-vc7.lnt -t4 +ffb +linebuf -i<intermediate files folder> <source file>
```

Put simply, the above command line defines a single file (or "unit checkout") analysis using the configuration files `project.lnt`, `std.lnt` and `env-vc7.lnt`. It is notable that the options given in each .lnt file are applied in order – so those specified later in the command line can override directives contained within (for example) `std.lnt`. This is useful for message suppression purposes, as we can almost always override a high level option later in the command line.

This example also shows several PC-Lint options which are worthy of mention:

- **`-i`** specifies a folder for include and/or indirect files. In this particular case it is being used to specify the location of PC-Lint indirect files referenced within `std.lnt`, and any `.tlh` and `.tli` intermediate files located within the "Debug" subfolder under the project.
- **`-background`** instructs PC-Lint to run analysis at a low priority (useful if you are running several analysis tasks in parallel or are using a slow machine).
- **`-b`** suppresses the banner line
- **`--u`** instructs PC-Lint to ignore any files listed in the following `project.lnt` file (we will come back to this later). **`--u`** is usually used with `-u`.
- **`-u`** instructs PC-Lint to perform a "unit checkout" analysis - i.e. analyse a single source file or compilation unit.
- **`-t4`** tells PC-Lint that it should use a tab size of 4 spaces while parsing source files (one of the things PC-Lint can do is check indentation).
- **`+ffb`** instructs PC-Lint that it should assume ANSI compliant for loop scoping. With compilers where this behaviour can be set by project settings (notably Visual Studio .NET 2002 onwards), the alternate option (`-ffb`) may be required on specific projects.
- Each occurrence of **`+linebuf`** doubles the size of the PC-Lint input buffer (the default size is 620 characters).

Configuring PC-Lint for Specific Project Configurations

The file `SourceVersioner_vs71_Debug_Win32.lnt` is a project specific indirect file - an indirect file which specifies the PC-Lint options for a particular project configuration and platform. I usually refer to these as "project.lnt files" for convenience.

A quick glance at the contents of such a file reveals that it contains preprocessor directives, the locations of additional include folders and a list of the files in the project:

```
// Generated by Visual Lint 2.0.0.91 from file: SourceVersioner_vs71.vcproj
// -dConfiguration=Release|Win32
//
-D_UNICODE;UNICODE            // CharacterSet = "1"
-DWIN32;NDEBUG;_CONSOLE       // PreprocessorDefinitions = "WIN32;NDEBUG;_CONSOLE"
-D_CPPRTTI                    // RuntimeTypeInfo = "TRUE"
-D_MT                         // RuntimeLibrary = "0"

-i"..\Include"

SourceVersioner.cpp           // RelativePath = "SourceVersioner.cpp"
SourceVersionerImpl.cpp       // RelativePath = "SourceVersionerImpl.cpp"
stdafx.cpp                    // RelativePath = "stdafx.cpp"
Shared\FileUtils.cpp          // RelativePath = "Shared\FileUtils.cpp"
Shared\FileVersioner.cpp      // RelativePath = "Shared\FileVersioner.cpp"
Shared\PathUtils.cpp          // RelativePath = "Shared\PathUtils.cpp"
Shared\ProjectConfiguration.cpp
                              // RelativePath = "Shared\ProjectConfiguration.cpp"
Shared\ProjectFileReader.cpp  // RelativePath = "Shared\ProjectFileReader.cpp"
Shared\SolutionFileReader.cpp // RelativePath = "Shared\SolutionFileReader.cpp"
Shared\SplitPath.cpp          // RelativePath = "Shared\SplitPath.cpp"
Shared\StringUtils.cpp        // RelativePath = "Shared\StringUtils.cpp"
Shared\XmlUtils.cpp           // RelativePath = "Shared\XmlUtils.cpp"
```

These indirect files are **critically** important in ensuring that PC-Lint uses the same analysis configuration as the compiler itself. Any mismatch in preprocessor directives or include paths is likely to result in a deluge of angry (and very misleading) error messages.

It should be obvious by now that configuring PC-Lint for a given project can be a very tricky business.

If you are using Visual C++, PC-Lint can read Visual C++ project files to generate `project.lnt` files directly. For example, to generate a `project.lnt` file for the `Unicode Debug` configuration and `Win32` platform of project `CoreLib` you could use the command line:

```
lint-nt.exe CoreLib.vcproj +d"Win32|Unicode Debug" >CoreLib_Win32_Unicode_Debug.lnt
```

Unfortunately, current versions of PC-Lint cannot correctly handle Visual C++ projects which specify include folder or preprocessor symbol specifications using Visual Studio environment variables or inherited property sheet (`.vsprops`) files. If you have projects structured in this way, you will have to either write the .lnt files manually or use a third party tool to do so.

Furthermore this technique is limited to Visual C++ project files; for other compilers you will have to write the `project.lnt` files yourself.

Whole Project Analysis

The list of files shown in the example above is used in "whole project analysis", which involves analysing all of the files in a project together. This method requires significantly more memory than analysing each source file individually, but has the advantage of allowing PC-Lint to identify issues (e.g. unreferenced functions) which only become apparent in a wider context than an individual compilation unit.

If the following command line is used, PC-Lint will perform such an analysis on the entire project:

```
"C:\Data\PC Lint\8.00\lint-nt.exe" -i"C:\Data\PC Lint\8.00" -background -b "C:\Data\PC
Lint\8.00\std_vs71.lnt"  env-vc7.lnt -t4 +ffb +linebuf -
iC:\Data\Code\Projects\Applications\SourceVersioner\Development\Release
C:\Data\Code\Projects\Applications\SourceVersioner\Development\SourceVersioner_vs71_Release_Win32
.lnt
```

By comparison with the previous example, the `-u` and `--u` options are not used and a specific source file need not be specified since the `SourceVersioner_vs71_Release_Win32.lnt` file contains a list of files to analyse.

A similar method can be used to check all files in a complete solution or workspace (much more useful, on the face of it). However, if you intend to do so please bear in mind that:

- You will have to generate a top level "solution.lnt" file which dynamically sets the preprocessor directives and additional include folders for every file in the solution – and this can easily be an order or magnitude harder than getting them right for a single project.
- The analysis runs in a single threaded process, so you will not have representative results for quite some time if the codebase is large.

In practice I find that single file analysis usually gets you 95% of the way. It is however worth running whole project analysis from time to time to identify any deadwood which has crept in.

PC-Lint Messages and Categories

PC-Lint organises its messages into five categories of varying severity:

- Elective Notes e.g. 953 ("Variable could be declared as const")
- Informational - e.g. 1924 ("C-Style cast)
- Warnings - e.g 534 ("Ignoring return value of function") or 1401 ("Member symbol not initialised by constructor")
- Errors – e.g. 1083 ("1083 - Ambiguous conversion between 2nd and 3rd operands of conditional operator")
- Fatal Errors – e.g. 322 ("322 - Unable to open include file")

Conveniently, PC-Lint has a **`-w`** option which allows the warning level to be set globally. For example, **`-w3`** (the default) will enable only messages of level "Warning" and above – so all Elective Notes and Informational Messages will be suppressed. Similarly, **`–w4`** will enable all messages except Elective Notes.

If you need to enable a specific message below the current warning level, you can simply add a `+e` directive for the message in question after the `–w` option. This brings us to a very useful way of detecting only specific issues - for example, if you wanted to detect any unused include files in a project, adding the options **`–w0 +e766`** to your command line (after all other .lnt files) will enable **only** message 766 ("Header file not used in module").

The following are examples of the sort of issues I specifically look for the first time I analyse a third party codebase (interestingly, very few of them are warnings – most are actually informational):

- 429 Custodial pointer 'Symbol' (Location) has not been freed or returned (Warning)
- 578 Declaration of symbol 'Symbol' hides symbol 'Symbol' (Warning)
- 716 while(1) ... (Informational)
- 717 do ... while(0) (Informational)
- 777 Testing float's for equality (Informational)
- 795 Conceivable division by 0 (Informational)
- 801 Use of goto is deprecated (Informational)
- 825 Control flows into case/default without -fallthrough comment (Informational)
- 1506 Call to virtual function 'Symbol' within a constructor or destructor (Warning)
- 1725 Class member 'Symbol' is a reference (Informational)
- 1735 Virtual function 'Symbol' has default parameter (Informational)
- 1773 Attempt to cast away const (or volatile) (Informational)

If you are running PC-Lint on a codebase for the first time with a (consequently) relaxed warning policy, it may just be worth turning on any issues you are specifically concerned about (using a +e directive in options.lnt) to see if any of them manifest themselves.

Common Analysis Failures

Some errors are invariably fatal to the analysis in nature. The ones you are most likely to see are:

- **Fatal Error 314: Previously used .lnt file**

  Each .lnt file can only be used **once** in the command line – directly or by inference. If you see this error, check your configuration files and command line for repeated `.lnt` files.

- **Fatal Error 307: Can't open indirect file**

  This error indicates that PC-Lint was unable to locate a .lnt file on its search path. Check that the file exists in the correct location and that PC-Lint has a `-i` directive locating its folder.

- **Fatal Error 322/Error 7: Unable to open include file**

  If an include file cannot be found you will receive fatal error 322 and analysis will abort, unless error 322 has been suppressed - in which case error 7 will be raised instead and analysis will continue. In either case, check your include paths and pre-processor symbol definitions – particularly at project configuration level.

- **Error 91: Line exceeds Integer characters (use +linebuf)**

  PC-Lint has a fixed line buffer of 620 characters. If you see this message, add a `+linebuf` directive (which doubles the size of the buffer) to the command line and try again. Repeat the process until the error disappears...

- **Error 303: String too long (try +macros)**

  As per error 91, but this time in macro definitions. Add `+macrobuf` directives until it goes away.

Analysis Speed

The speed of a PC-Lint analysis is influenced by many factors, of which the most significant are probably the structure of the codebase being analysed and the performance of the system the analysis is being run on. For a large

codebase it is not unknown for a complete analysis run to take several hours – and needless to say, this can potentially pose problems or even act as a disincentive to run the analysis at all.

As PC-Lint operates directly on source files (unlike the .NET assembly analysis tool FxCop, for example), it must read and process include files; therefore the structure of include dependencies in your project can be **very** significant in terms of analysis time. Reduce your include dependencies (which you already do routinely, I assume…?) and the analysis time is likely to be reduced. Similarly, a reasonably specified modern system (e.g. a Core 2 or Athlon X2 machine with 7200rpm disks and 2GB or more memory) will crunch through your codebase in a far shorter time than that cruddy old P4 box that was co-opted as a build server 4 years ago.

As current PC-Lint versions are entirely single threaded, adding more CPU cores will not reduce the analysis time unless you somehow run multiple analysis tasks simultaneously. On multicore systems it is entirely practical to do this (single file analysis is particularly amenable to parallelisation), and it can reduce analysis times quite significantly in some cases.

One interesting recent development is that PC-lint 9.0 (released last Autumn) add options for precompiled and bypass header support - which allow the contents of system include files to be preprocessed and cached in a similar way to the Visual C++ precompiled header compilation mechanism. The tests I have run suggest that these options can potentially cut analysis times by a factor of at least 3 to 4, although with Visual C++ headers activating them can cause extraneous analysis errors (at least with PC-Lint 9.00b) – which no doubt will be resolved in a future update from Gimpel.

Overall, although techniques such as parallelisation and precompiled headers can significantly reduce your analysis time, I would strongly recommend that you take a close look at the structure of your codebase and the performance of the system the analysis is being run on first.

Tuning out issues in libraries

Regardless of which libraries you use in your projects, you are likely to encounter PC-Lint issues in class or macro definitions within those libraries. When you don't have any control over the implementation of the libraries concerned, such "noise" can be irritating to say the least (athough it could of course be indicative of a real problem with the library implementation you should be aware of…).

Fortunately, using some of PC-Lint's error suppression directives it is in most cases relatively easy to write lint directives to prevent this happening, for example:

```
-emacro(1924, MAKE_HRESULT)
-esym(1932, ATL::CAtlExeModuleT<*>)
-etype(1746, boost::shared_ptr<*>)
```

The directives `-emacro` , `-esym` and `-etype` suppress issues in macros, named symbols, and instances of objects or values of the given type respectively (note the availability of wildcards here). PC-Lint has quite a number of such options for fine-grained control of warning policy, and it is well worth getting to know how to use them effectively. By the way, you can also use them in the code directly by prefixing them with a `//lint` comment, for example:

```
//lint -esym(1712, CoreLib::CSomeClass)
```

A special mention should be reserved for the "big hammer of last resort" for noisy library header files. It looks something like this:

```
//lint –save –w0
#include "NoisyHeader.h"
//lint -restore
```

The meaning of the above should be fairly self-explanatory – the header in question has so much wrong with it that we're just going to disable any issues from it for now. Of course, if you find yourself doing this as anything but a short term measure you really should be asking whether the header in question code has a long term place in your codebase...

Incidentally, over the past few years I have amassed significant collection of such "tuning" directives for issues within the Win32 libraries (Win32 API, ATL, WTL and MFC) which have been collected into specific indirect files so we don't have to specify them directly in each project or in our global warning policy. You can download them from [2] if needed.

Turning Down the Volume - How to cope with a deluge of analysis results

I am pretty sure that most developers who run PC-Lint do so using either pre-configured scripts or some form of rudimentary integration into whichever IDE they are using.

Regardless of the method used, the issue of how to deal with the volume of analysis results it can produce remains.

Common strategies include turning off all but the most critical issues (with the intention of gradually enabling others as the issues are addressed - not that that always happens of course), grepping the raw analysis results, exporting to a spreadsheet and filtering there....the list goes on.

Which method you use really doesn't matter. What matters is that you find an approach which works well for you and your organisation, and continue to refine it as appropriate (the same applies to your warning policy, of course).

(Free) Tools Which May Help

While these techniques can be very efficient ways to spot specific issues on your "watch list", they may not give you much of a feel for the state a codebase as a whole is in. For that, a tool such as Ralph Holly's *Aloa* [3] or *LintProject* [4] can be helpful.

Aloa ("A Lint Output Analyser") takes advantage of the fact that PC-Lint can be configured to produce XML output, and interprets the results to produce a text based summary of the issues encountered:

```
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

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
```

If you are looking for a way to get to grips with an unfamiliar codebase, *Aloa* is well worth checking out.

LintProject (a tool I wrote several years ago and released as freeware via codeproject.com) takes a different approach: it runs a simple unit checkout analysis on complete Visual C++ solutions and produces simple HTML reports of the results, organised by project and file.

I wasn't aware of Aloa at the time, and we needed a simple way to analyse the codebase of our main product - 80 projects comprising about 250,000 lines of code. Obviously, a manual per file or per project analysis was not going to work, and we needed a way to analyse a complete solution in one hit. Since PC-Lint does not provide a way to do this "out of the box" (and neither does *Aloa*, unfortunately) that meant writing something new.

Like Aloa, *LintProject* is a simple command line tool which runs PC-Lint and captures its output. Unlike Aloa, *LintProject* runs on complete Visual C++ solutions or workspaces (whichever terminology takes your fancy) and produces simple HTML reports describing the number of issues found in each project and file. Although the output it

produces is far less detailed than Aloa, it is a useful way to get a feel for the analysis state of a complete Visual C++ solution.

Conclusion

Analysis tools such as PC-Lint can uncover **real** problems in your codebase. Unfortunately, far too many developers are either blissfully aware of the capabilities of code analysis tools or sceptical of their usefulness (sadly many even see compiler warnings as unwanted "noise" and would rather turn down the warning level rather than fix the root causes).

As a result advocating the use of such tools on a project can be a bit of an uphill struggle - unless the project is failing, in which case they may well grasp at anything for a while.

However, in our experience it is **definitely** worth investing the time to integrate the use of such a tool into your development process. Although you can expect to be warned of significant numbers of issues in your code when you first analyse it, if you persevere the analysis tool is likely to repay the investment you put into it many times over by assisting you in increasing the quality of your code.

Like any tool, if you take the time to learn how to use it effectively you will be more likely to reap the best results.

[1] http://www.gimpel.com

[2] http://www.riverblade.co.uk/products/visual_lint/downloads/PcLintConfigFiles.zip

[3] http://www.ddj.com/cpp/184401810

[4] http://www.codeproject.com/KB/applications/lintproject.aspx