Taming the Lint Monster

Part 1: A personal perspective of the PC-Lint code analysis tool

An All Too Common Story

It's such a common story. Partway through a project, the company starts to become anxious about the number of defects that are being identified in the product, and how long they are taking to fix. Even worse, *customers are beginning to notice.*

Something must be done. Additional resources are thrown at the problem, but somehow it doesn't ever seem to be enough. The codebase is large, complex and hard to understand and maintain. It is - for all intents and purposes - a "Big Ball of Mud" (not that anyone in the company would know such a term; after all - they are far too busy firefighting to read tech blogs and keep up to date with current trends in software development).

After several months of throwing additional firefighters at the problem, someone has the bright idea™ to find out just how much hidden nastiness is lurking in the code base waiting for the right moment to let loose it's wrath on the unsuspecting team.

An appropriate tool is identified and procured, and then the real fun starts - actually *using* it.

Inevitably, it never quite turns out the way the team (or their managers) expect. Not only does it turn out to be an absolute nightmare to configure and use (after all you never appreciate how much work compiler project files can save you from until you have to maintain something comparable yourself), but when the team do *finally* get it working to their satisfaction the results it produces are so volumous that nobody quite knows what to do with them. Worse, they contain some **really bad news** ™.

As all too often happens, dealing with the issues the tool raises is deemed to be a) too expensive, b) too risky and c) not as much fun as writing new copy-paste code (though nobody is ever quite honest enough to admit to the latter).

The team conveniently forget about the whole experience and go back to compiling at warning level 3 as they always have done. The installation disk for the offending tool is quietly hidden away in a desk draw and forgotten...and of course, the Big Ball of Mud grows ever bigger until the *inevitable "lets just re-write it in language "X"* event a year or two later. With an eye on what language "X" would look like on everyone's CV, of course...

But it sure did seem like a good idea at the time.

Meeting PC-Lint in a Dark Alley

My personal experience of PC-Lint started around 1996. At the time, I was leading a small team (actually, there were three of us...) developing virtual instrument software for a Windows NT 3.51 based automatic test system to support a military aircraft. My team was just one of several on the project, and although we had a good team we were badly under-resourced and constantly on edge – partly because of (seemingly politically motivated) interference by our prime contractor.

It was challenging, but highly stressful work (our Software Manager leaving the project due to stress was testament enough to that). Nevertheless, we were making good progress - our code was working and we were back on schedule (or at least we were after re-estimating everything from the ground up with three times the original estimate...). On the surface, the worst was behind us, and our system was actually starting to look pretty good.

It was at that point that someone had the bright idea of applying code analysis tools to the codebase to see if any potential nasties were lurking within. A demonstration by McCabe was arranged, and although it looked very impressive - especially on the (as it seemed at the time) huge monitor they brought (ours were tiny 14" things then) - but the price was well beyond our budget - especially at such a late stage of the project.

Enter Gimpel PC-Lint [1], a C/C++ static code analysis tool published by Gimpel Software (then at version 6, if I remember rightly). This was a tool I was not aware of at the time but which my co-team leader recommended.

When it arrived we had of course to learn how to use it. At the time we were using a mixture of Visual C++ 1.52 and 4.2, so the "integration" consisted of adding a custom tool to run an analysis on a single file and display the results in the output window. Rather amazingly, that is still the way it is usually done today.

Needless to say the results seemed cryptic and verbose, so we did what most teams in this situation do - turn off most of the PC-Lint messages, leaving only those which we thought might indicate a serious problem. That is of course an entirely reasonable approach, but it does of course carry the risk that in doing so you may inadvertently mask something very significant.

The next thing that happened was that our code review policy was amended to include a requirement that each lint issue remaining in the codebase be justified by the developer responsible for it. Suddenly code reviews became more of a challenge, so in that respect the process worked well. We did, however only use a very small subset of PC-Lint's capability - and the warning policy remained (to my mind, looking back) far too lax for long term use.

Up Close and Personal with PC-Lint

PC-Lint (along with its Unix/Linux cousin Flexelint) is a C/C++ static code analysis tool published by Gimpel Software [1].

Although there are a number of other similar tools on the market (e.g. PreFAST, Klocwork Insight, Parasoft and QA C++) most are part of a much larger integrated toolset with an enterprise price tag to match. The only open source contender I'm aware of (Splint [2]) is limited to C and its authors unfortunately seem to have no intention of adding C++ support (however, as the source code is freely available, I will happily leave that as an exercise for the reader...).

PC-Lint is a command line tool with a range of options rivalling those of a full featured C++ compiler. All of that configurability comes at a price of course - complexity and and all it entails. It is far from easy to use, and the analysis results it produces can also be verbose and cryptic to say the least. Nevertheless, PC-Lint is **very** thorough, and more than capable of exposing potentially serious hidden flaws in your codebase.

At its absolute simplest, a PC-Lint command line to analyse a single file and output the results to the console looks something like this:

```
lint-nt std.lnt <filename>
```

where `lint-nt.exe` is the PC-Lint executable and `std.lnt` is a configuration file (Gimpel call them "indirect files") describing the compiler and framework configuration (preprocessor symbols etc.), include paths and warning policy.

`std.lnt` usually consists of a set of references to other indirect files, together with a handful of options and a set of include folder specifications, for example:

```
// Standard PC-Lint configuration options for Visual Studio 2005
// Generated by Visual Lint version 1.5.9.79 at Tuesday, April 08, 2008 16:40:13
//
au-sm123.lnt   // Effective C++ 3rd Edition policy
co-msc80.lnt   // Visual Studio 2005 compiler definitions
lib-mfc.lnt    // MFC library definitions
lib-stl.lnt    // Standard Template library definitions
lib-w32.lnt    // Win32 API definitions
lib-wnt.lnt    // Windows NT API definitions
lib-atl.lnt    // ATL definitions
riverblade.lnt // Other library tuning
options.lnt    // Warning policy
-si4 -sp4      // Integers and pointers are 4 bytes

// Include definitions for platform 'Win32'
-i"C:\Program Files\Microsoft SDKs\Windows\v6.0\Include"
-i"C:\Program Files\Microsoft SDKs\Windows\v6.0\Include\gl"
-i"C:\Program Files\Microsoft Visual Studio 8\VC\include"
-i"C:\Program Files\Microsoft Visual Studio 8\VC\atlmfc\include"
-i"C:\Program Files\Microsoft Visual Studio 8\SDK\v2.0\include"
```

Of particular note is `options.lnt`, which usually defines the warning policy, together with additional issue suppression options. Our own warning policy is actually pretty simple - it consists of the full set of Scott Meyers recommendations (activated by the inclusion of `au-sm123.lnt` in `std.lnt`), with a handful of issues suppressed by `-e` directives:

```
-e27     // Avoid "Illegal character" errors on VS2005 .tlh and .tli files)
-e537    // (Warning -- Repeated include file)
-e655    // (Warning -- bit-wise operation uses (compatible) enum's)
-e730    // (Info -- Boolean argument to function)

-e783    // (Info -- Line does not end with new-line)
-e944    // (Note -- Left / Right argument for operator always evaluates to False)
-e1550   // (Warning -- exception thrown by function is not on throw-list of function)
-e1774   // (Info -- Could use dynamic_cast to downcast ptr to polymorphic type)
-e1904   // (Note -- Old-style C comment -- Effective C++ #4)
```

Such an aggressive warning policy is however not well suited for use with a codebase which has not been analysed before. In such cases, replacing `au-sm123.lnt` with a less stringent alternative is often advisable until the major issues in the codebase have been dealt with.

If you use Microsoft Visual C++ you can download sample `std.lnt` and `options.lnt` files for all Visual Studio versions from Visual C++ 6.0 to Visual Studio 2008 and eMbedded Visual C++ 4.0 from the Riverblade website. [3]

When PC-Lint is run on a source file, the results are by default presented in textual form, as this example shows:

```
--- Module: CJFlatHeaderCtrl.cpp
}
CJFlatHeaderCtrl.cpp(160): error 1401:
       (Warning -- member 'CCJFlatHeaderCtrl::m_bSortAsc'
(line 146, file ..\Include\CJFlatHeaderCtrl.h) not initialized by constructor)
```

```
}
CJFlatHeaderCtrl.cpp(166): error 1740:
      (Info -- pointer member'CCJFlatHeaderCtrl::m_pParentWnd'
(line 150, file ..\Include\CJFlatHeaderCtrl.h)
      not directly freed or zero'ed by destructor

-- Effective C++ #6)

--- Global Wrap-up
error 900: (Note -- Successful completion, 2 messages produced)
```

Message Formatting and Presentation

The format of the analysis results can be configured using a suitable indirect file. With the appropriate configuration for a given development environment, most development environments can understand enough to provide "double click to go to issue location" functionality for analysis results piped to its output window.

Gimpel provide a number of such environment options files in the PC-Lint installation (for example `env-vc8.lnt` for Visual Studio 2005), and others can be downloaded from the support page at http://www.gimpel.com/html/ptch80.htm (PC-Lint 8.00) or http://www.gimpel.com/html/ptch90.htm (PC-Lint 9.00).

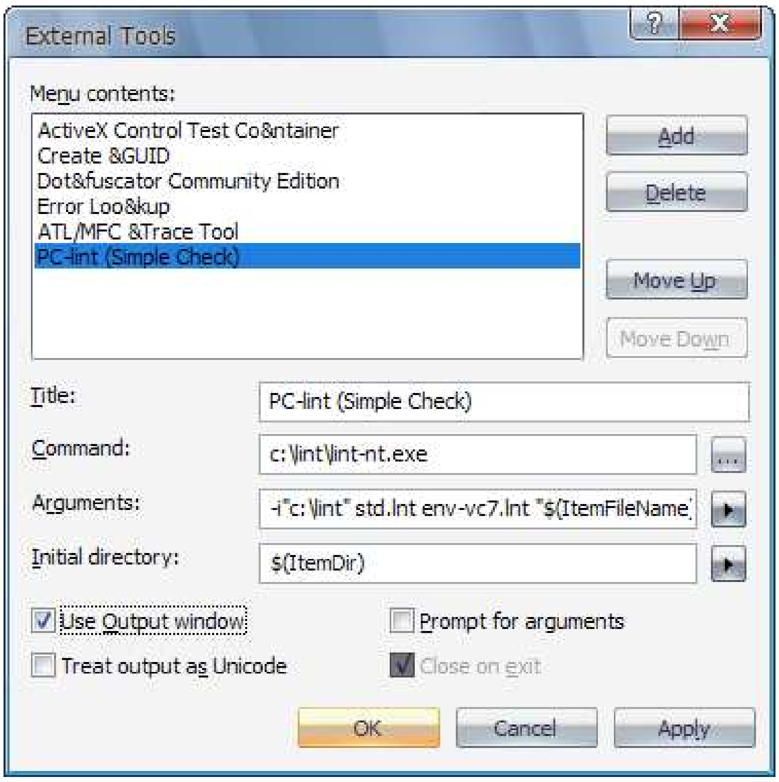
A special mention must be made of `env-xml.lnt`, which allows PC-Lint to easily generate XML output such as:

```
<?xml version="1.0" ?>
<doc>
  <message>
    <file>fileb.cpp</file>
    <line>2</line> <type>Info</type>
    <code>753</code>
    <desc>local class 'X' (line 2, file fileb.cpp) not referenced</desc>
    </message>
  <message>
    <file>fileb.cpp</file>
    <line>4</line>
    <type>Info</type>
    <code>754</code>
    <desc>local structure member 'X::a' (line 4, file fileb.cpp) not referenced</desc>
  </message>
</doc>
```

If you intend to post-process the raw analysis results, the usefulness of this should be obvious!

(Very) Basic IDE Integration

Incidentally, some of the the `env-*.lnt` files also contain instructions on how to perform a basic integration of PC-Lint within the corresponding IDE. This usually takes the form of a custom tool, as you can see in this screenshot of an example for Visual Studio:



External Tools

Menu contents:

- ActiveX Control Test Co&ntainer
- Create &GUID
- Dot&fuscator Community Edition
- Error Loo&kup
- ATL/MFC &Trace Tool
- PC-lint (Simple Check)

Title: PC-lint (Simple Check)

Command: c:\lint\lint-nt.exe

Arguments: -i"c:\lint" std.lnt env-vc7.lnt "$(ItemFileName)

Initial directory: $(ItemDir)

[✓] Use Output window
[ ] Prompt for arguments
[ ] Treat output as Unicode
[✓] Close on exit

Although such an integration is how it is often done, it can be less than ideal – the operation is modal (which you will **really** feel if the analysis takes more than a few seconds), and the analysis results arrive in a daunting blob of monospaced text. Every developer has their own strategies for dealing with those issues, of course – although I suspect that the industry standard is probably extended coffee breaks and lots of grepping….

To Conclude Part 1…

PC-Lint is a **very** capable tool, but one in which you really need to invest the time and effort to learn how to use it effectively if you want to reap the real benefits it can yield (if you are looking for a "quick fix" for a dirty codebase, look away now…).

Although basic PC-Lint configurations are usually pretty straightforward, things can get rather complex very quickly. To close this first part of the article, take a look at this example command line generated by Visual Lint:

```
"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 -
iC:\Data\Code\Projects\Applications\SourceVersioner\Development\Debug
c:\Data\Code\Projects\Applications\SourceVersioner\Development\Shared\FileUtils.cpp
```

Even though this command line could be shortened by using relative paths it is still not really something you would want to have to type very often from memory, is it?

In part 2 (“Deconstructing the PC-Lint command line”) we will look in detail at how this command line is formed, and some of the common PC-Lint options. We will also discuss how to configure PC-Lint for a particular project configuration, and consider some strategies for dealing with the analysis results it generates.

References

[1] http://www.gimpel.com
[2] http://www.splint.org
[3] http://www.riverblade.co.uk/products/visual_lint/downloads/PcLintConfigFiles.zip