Use Lint for Static Code Analysis

Language specifications, including those for C and C++, are often loosely written. A static analysis tool called lint can help you find dangerous and non-portable constructs in your code before your compiler turns them into run-time bugs.

Anyone who has written a program has had to debug code. In many cases, after staring at the code for hours, we finally realize that we have done something stupid like read an uninitialized variable, or index beyond the end of an array. With experience, the incidence of these “doh!” bugs decreases; but they can still be extremely frustrating and costly.

What we need at such times is an infinitely patient, anal-retentive expert to inspect every inch of our code. This expert must be more thorough than any compiler, and should report back in a completely dispassionate way everything that is potentially wrong with our code. A development tool called lint is the closest you can get to this ideal.

Lint is a tool similar to a compiler in that it parses C or C++ source files. It checks these files for syntactical correctness. To lint a file called foo.c, one typically just types:

```
lint foo.c
```

at the command line. Naturally, though, there are also many optional command line switches.

Standard features

Whereas a compiler concerns itself primarily with code generation, lint is completely devoted to checking your code for a myriad of possible defects. The key word here is possible. Just because lint flags a section of your code for review, it doesn’t necessarily mean a problem will occur when you compile that code with your particular compiler.

Lint is designed to be compiler-agnostic and is, in fact, frequently in the business of focusing your attention on parts of the code that might result in different behavior depending on the specific compiler used.

The specific list of problems that lint checks varies by implementation and version of the tool. However, most flavors of lint will at least check for the following:

- Possible indexing beyond array bounds
- De-referencing of null pointers

- Suspicious assignments (such as if (a = b))
- Mismatches in variable types (such as foo declared as a double in one file and used as a long in another)
- Potentially dangerous data type combinations
- Unused variables
- Unreachable code
- Header files included multiple times and/or unnecessarily
- Non-portable constructs

Lint checks so many things, in fact, that it's common for the tool to initially produce as many errors and warnings as there are lines of code in the source file that's input. This is a big turn-off for many potential users, since their attitude tends to be "this tool is so picky it's ridiculous." However, working through each warning and correcting it can be a rewarding exercise. As an example, consider this seemingly innocuous code:

```
main(void)
{
      int i;
      for (i = 0; i < 10l; i++)
      {
            printf("%d\n",i);
      }
}
```

What will be the last number printed when you run this program? If you answered "100," you're wrong. This is a perfectly valid program, and will compile without warning on most any compiler. However, lint will complain about something. If you can't see the problem via visual inspection, then I suggest you cut and paste the code above and run it through your favorite debugger. Note, and a big hint here: do not simply type this program into your editor. Observe how long it takes you to find this problem–and then ask yourself whether wading through lint warnings isn't so bad.

Getting the lint out

What sort of real world benefits can you expect from addressing all of the warnings produced by lint? My experiences on a typical project involving a small microcontroller (total code size below 32KB) included the following:

- Lint found two or three outright bugs–before I had even started testing the code
- I learned something about the C language each time I ran lint

- My final code was cleaner because lint informed me of unused variables, macros, and header file includes that could be safely removed
- I was better informed of potential portability issues

Given the above, it will probably not surprise you to learn that organizations that are really serious about code quality often insist not only that all code compile without warnings (which is relatively trivial to achieve) but also that it be "lint free"-that is, generate no warnings with lint. This is a much more difficult criteria to achieve.

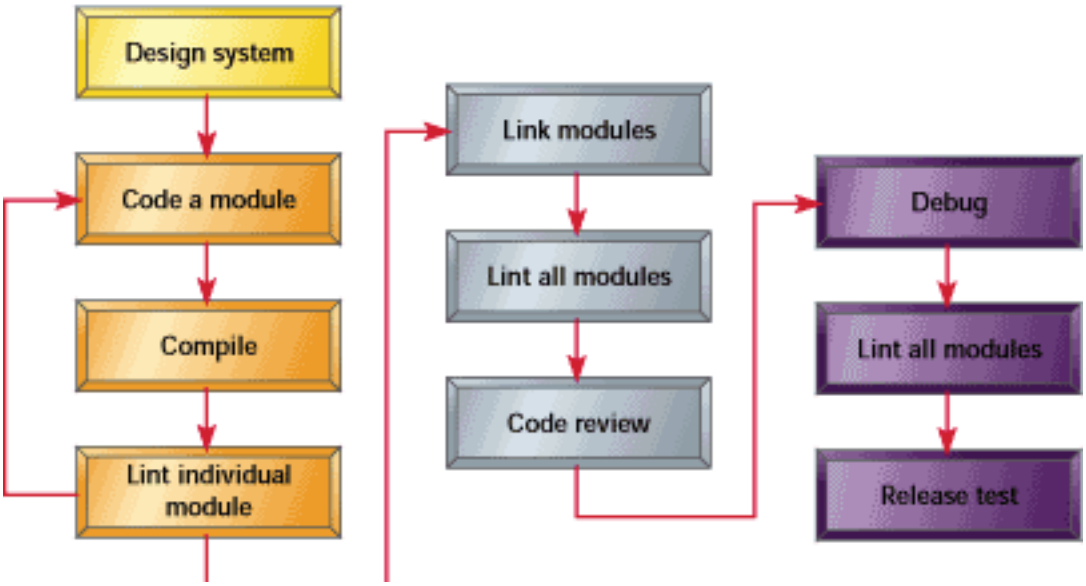

Figure 1: How lint fits into the development process

It's worth looking at where lint fits into the development process. My general design flow is shown in Figure 1. Once I have code that compiles, I lint it. If the code gets through lint okay, it's highly unlikely that I'll be embarrassed in the code review. During the debug phase, it's normal for changes to be made to the code. However, once the code is debugged, and before it is passed to the release test, I normally lint the code again. Why? I'm always amazed at the number of sloppy coding constructs that occur when code is being debugged. Lint is a great tool for identifying that ratty piece of code that was put in there to help debug something and then promptly forgotten.

Sources of lint

Lint is a standard tool on most Linux or Unix development systems. In the PC realm, however, you often have to go out and buy lint, or find a free or shareware version. If you do buy lint, rest assured it will likely be the best money you have ever spent in your embedded career. Most lint variants are relatively inexpensive (less than $1,000) and worth every penny.

Incidentally, you may be wondering how well lint handles all those nasty little compiler extensions that are so common in embedded development. This is an area where the commercial programs outshine the free and shareware offerings. In

particular, some versions of lint allow considerable customization of lint's rules, such that all those extensions are correctly handled. In some cases, the compiler definitions are even supplied by the lint vendor. In others, you may be able to get them from the compiler vendor.

If you don't have access to lint, but are using the GNU tools (on Unix or a PC), simply use gcc's -Wall flag to achieve about 80% of the same functionality.

So, for all the neophytes out there, get yourself a copy of lint, and use it. If nothing else, your boss will be impressed with the maturity of your code. For all the experienced hacks who aren't using lint–watch out! The new guys who are using it might show you up.

Further reading

Wikipedia – Lint

Darwin, Ian F. Checking C Programs with Lint. Sebastopol, CA: O'Reilly, 1988.

This article was published in the May 2002 issue of *Embedded Systems Programming*. If you wish to cite the article in your own work, you may find the following MLA-style information helpful:

Jones, Nigel. "Introduction to Lint," *Embedded Systems Programming*, May 2002