Quality Code

Walking Upright

Whilst peer review is the best method of ensuring quality code, automated tools can also be employed. In this article we look at such a tool, and show how it can be used to improve your code.

BY STEVEN GOODWIN & DEAN WILSON

Program errors annoy users and hinder development. Software that takes a month to write will often take just as long again to debug and fix before the end product is deployed. Compiler warnings, although useful, do not go far enough to prevent a number of bugs that should be trapped at a much earlier stage.

Splint (formally called lclint) is a semantic checker which reads and understands your code. It can look at what you have said – and determine if that's what you meant.

In contrast, the compiler will point out show stopping syntax errors (such as undefined variables), which prevent an executable from being built. Many semantic errors can be caught by turning on all compiler warnings, but this is still not enough in most instances. (See Box 1: Swear and curse).

I'm a lumberjack...

Splint is available as a binary, or source package which can be downloaded from *www.splint.org*, with the current version (3.0.1.6) weighing in at around 1.5 MB.

The source is built using the standard GNU tool chain and should compile under any modern Unix. To build Splint from its source tarball where < version > is a string such as 3.0.1.6

```
$ tar -zxvf splint-<version>.tgz
$ cd splint-<version>
```

start the actual build

```
$ make
```

this stage requires root privileges

```
$ make install
```

If you have any difficulties or problems building from the source, there is a binary package available that runs straight out of the box with no external dependencies, and is available for Linux, FreeBSD, Windows and Solaris from the Splint homepage.

I've got two legs

Splint has to be able to read and parse the code in the same manner as a compiler would. So, if you have particular include directories that need to be used, you can either add a command line switch (as you would with gcc), or use the environment variable LARCH_PATH, mimicking how a Makefile would handle it.

For example, to add an include directory for a single run:

```
$ splint -I /usr/src/myproject➥
/include sptest.c
```

Or, to make that directory available on each run of splint in this terminal session (if using 'bash'), we use:

```
$ export LARCH_PATH=/usr/src➥
/myproject/include
```

To make this persistent across sessions add the line to your .bashrc or .bash_profile file, depending on your setup. Header files in the same directory as your source files do not need to be explicitly referenced as they are included by default.

You can test the install by typing:

```
$ splint --help version
```

A pleasant splint banner indicates that you now have a fully working install of the package. Now let us look at what it can do for us by running it with a simple sample program.

You're the doctor of my dreams

We have, below, the complete source code that we are going to use as our test bed for splint. Please note this code is not production quality, and deliberately contains bugs. It does, however, compile 100% cleanly with the strict gcc settings of -Wall, -ansi and -pedantic.

Box 1: Swear and curse

Splint can be a very exacting program. As an example of this, please note the following code sample:

```
#include <stdio.h>
int main(int argc, char **argv) {
int a=0;
if (a = 4);
return 0;
}
```

This code has no warnings when compiled with 'gcc test.c', but succeeds in detecting one warning when all compiler warnings are enabled with -Wall. Splint, in contrast, will pick up a grand total of 5 errors (or possible errors) in the same piece of code. Image the potential minefields present in a larger project!

THE AUTHORS

Steven Goodwin is a Lead Programmer who has just finished his fifth computer game. He has had more bugs than you've had hot dinners...but he claims they all belong to Dean Wilson.

Dean Wilson works in Perl, C and shell scripts at WebPerform Group Ltd in the City. His bug count currently exceeds the GNP of Japan...but he claims they all belong to Steven Goodwin.

```
$ gcc -Wall -ansi -pedantic ➥
test.c
```

Although these settings may appear overly conservative, in a real-world scenario where code is critical, or ported across multiple systems, these settings would be the norm (see Listing 1).

The program uses Numerology to calculate a mystical number which is derived from a persons name. This number can be used to tell your fortune, describe your personality, or demonstrate your character traits (like health, wealth, and gullibility). Allegedly!

So there's our code. 46 lines of code. 0 warnings. Can there be anything wrong? For starters it doesn't terminate – it appears to spin. So we need some extra help tracking down the error. Let us run splint and look for further clues…(see Listing 2)

All things dull and ugly

Wow! 14 warnings for a 'perfect' piece of code. Let's break these errors down to see where they come from, and why. The experienced reader may care to notice the different categories of problem that splint produces.

One of the first things to spot are the two 'Parameter … not used' errors on line 28, one with argc as the unused parameter, and one with argv. Both variables are present in our main() function, but neither are used. There is a good reason for this (in our case); our program doesn't use them.

Does this mean we can ignore this error? Only in this specific case. In virtually every other situation an unused parameter means there is some important data being orphaned inside the function. This should, invariably, be corrected. In this instance we should amend our source code to tell maintenance programmers that we are not using these parameters *intentionally*. For example:

```
    argc = argc;
    argv = argv;
```

Splint will no longer report this warning for argc and argv, although it will be reported for other unused parameters in the program. If you wish to ignore this type of error wholesale (i.e. in every problematic occurrence of the code) you can ask splint to ignore this type of error with the command:

```
$ splint --paramuse test.c
```

After running this command on the original source you will notice there are now only 12 warnings present, with both of those in the 'unused parameter' category having been removed.

Most of splint's warnings are grouped into such categories that can be ignored with a command line switch like '--paramuse'. This allows you to ignore specific types of error if you either don't agree with them, or they are not applicable to the product you are working on. Working through the error list above you should be able to pick out a number of such switches.

There are over one hundred different flags available, and so would be impractical to list them all here. You can review the categories available by using the command:

```
$ splint --help flags
```

The individual categories (memory, pointers and parameters, for example) can be shown with the equally simple:

```
$ splint --help memory
```

It's fun to charter an accountant…

Once the simpler errors have been removed, it is a good idea to progress through the list, solving each error in turn. After a problem area has been detected, briefly read the remaining problems to see if your fix could adversely affect other areas of the code.

When you are happy with your change, re-run splint to check that the error has gone, and no other warnings were produced as a result of your new code change.

Looking at our output, we see there's a problem with line 10.

```
t.c:10:19: Function parameter
array declared as manifest array
(size constant is meaningless)
```

Listing 1: Sample code

```
 1 #include <stdio.h>
 2
 3 int mapping[] = {
 4    1, 1, 4, 2, 4, 4, 2, 1,
 5    5, 4, 4, 2, 1, 5, 2, 5,
 6    5, 5, 4, 3, 3, 3, 3, 2,
 7    5, 5,
 8 };
 9
10 int num_calc(char array[11])
11 {
12 int i;
13 int total=0;
14 char c;
15
16 for(i=0;i<sizeof(array)/
   sizeof(array[0]);i++)
17    {
18    c = array[i];
19    if (c >= 'A' && c <='Z')
20       total +=mapping
   [(int)c-'A'];
21   else if (c >= 'a' && c
   <= 'z')
22       total +=mapping
   [(int)c-'a'];
23    }
24
25 return total;
26 }
27
28 int main
   (int argc, char**argv) {
29 char message[11] =
   {"Mystic Meg"};
30 unsigned int num;
31
32 if ((num = num_calc(message)))
33    {
34    /* reduce until its negative
   */
35    do
36       num -= 10;
37    while(num>=0);
38
39    /* Since we've overshot, add
   the last ten back*/
40    num += 10;
41
42    printf("The magic number for
   %s is %d\n", message, num);
43     }
44
45 return 0;
46 }
```

This tells us we are implying, by including the square brackets, that the function takes an array as an argument. It doesn't. 'C' can not pass arrays; only pointers to the start of arrays. We should therefore correct the code thus:

```
int num_calc(char *array)
```

Although not a problem as far as the compiler is concerned (both versions produce the same code), a maintenance programmer might imagine that this is actually trying to pass an array, and could be liable to introduce errors based on this incorrect assumption, as we will see later.

Gifts for all the family

Upon re-running splint we notice that we're down to 10 errors. Great, you might think. We've only made one (benign) change, but it has produced an unexpected side effect – it hides a potential error. This is another good reason why you should make changes incrementally to the code (as you would when the compiler produces errors) and not in bulk.

The error we lost concerned the sizeof operator in line 16. By referring to the *sizeof* a pointer, we are only considering (on a 32 bit machine) 4 bytes. By naming the pointer as if it were an array, the coder implied it would be 11 bytes long (the array size). This flaw can be fixed by re-writing the code correctly with the strlen function.

```
for(i=0;i<strlen(array);i++)
```

Our next error involves a type mismatch. Instead of blindly replacing the type we must make sure there are no complications. In this example, the 'strlen' function returns a value of type 'size_t'. This is a system-defined type (from malloc.h) which has enough capacity to store any possible memory location that the system could address. It is - sometimes referenced as the size of the 'sizeof' operator.

It is more 'correct' to use size_t because the loop is intended to reference an arbitrary array which could, in theory, extend across the whole memory.

Changing the type here does not cause us any problems, especially since under our (32 bit x86) machines it is only a change in sign (from signed to unsigned), but it is only true in this case. Changing signs arbitrarily is a dangerous comfort with which to surround yourself. You will see the proof of this point shortly.

Our next problem is one of types, which occurs twice (once at line 20, and once on 22): we use a char to reference an integer array element. Since 'C'

First output from splint

```
$ splint t.c

Splint 3.0.1.6 --- 23 June 1912

t.c:10:19: Function parameter
  array declared as manifest array
  (size constant is meaningless)
  A formal parameter is declared
  as an array with size. The size
  of the array is ignored in this
  context, since the array formal
  parameter is treated as a
  pointer.
  (Use -fixedformalarray
  to inhibit warning)
t.c: (in function num_calc)
t.c:16:18: Parameter to sizeof is
  an array-type function
  parameter: sizeof((array))
  Operand of a sizeof operator is
  a function parameter declared as
  an array. The value of sizeof
  will be the size of a pointer to
  the element type, not the number
  of elements in the array. (Use
  -sizeofformalarray to inhibit
  warning)
t.c:16:10: Operands of < have
  incompatible types (int,
  arbitrary unsigned integral
  type): i< sizeof((array))
  /sizeof((array[0]))
  To ignore signs in type
  comparisons use +ignoresigns
t.c:20:21: Incompatible types for
  - (int, char): (int)c - 'A'
  A character constant is used as
  an int. Use +charintliteral
  to allow character constants to
  be used as ints.  (This is safe
  since the actual type of a char
  constant is int.)
t.c:22:21: Incompatible types for
  - (int, char): (int)c - 'a'
t.c: (in function main)
t.c:29:20: Initializer block for
  message has 1 element, but
  declared as char [11]: "Mystic
  Meg"  Initializer does not
  define all elements of a
  declared array. (Use
  -initallelements to inhibit
  warning)
t.c:32:6: Assignment of int to
  unsigned int:
  num = num_calc(message)
t.c:32:5: Test expression for if
  not boolean, type unsigned int:
  (num = num_calc(message))
  Test expression type is not
  boolean or int. (Use
  -predboolint to inhibit warning)
t.c:37:8: Comparison of unsigned
  value involving zero: num >= 0
  An unsigned value is used in a
  comparison with zero in a way
  that is either a bug or
  confusing. (Use -unsignedcompare
  to inhibit warning)
t.c:42:53: Format argument 2 to
printf (%d) expects int gets
unsigned int: num
t.c:42:38: Corresponding format
  code
t.c:28:14: Parameter argc not used
  A function parameter is not used
  in the body of the function. If
  the argument is needed for type
  compatibility or future plans,
  use /*@unused@*/ in the argument
  declaration. (Use -paramuse to
  inhibit warning)
t.c:28:27: Parameter argv not used
t.c:3:5: Variable exported but not
  used outside t: mapping
  A declaration is exported, but
  not used outside this module.
  Declaration can use static
  qualifier. (Use -exportlocal to
  inhibit warning)
t.c:10:5: Function exported but
  not used outside t: num_calc
t.c:26:1: Definition of num_calc

Finished checking --- 14 code
warnings
```

allows chars to do this (using the rules of promotion), there is not really a big problem with the code.

Instead of passing an extra switch to the splint program, we shall formally fix the code with type casts. This, in addition to giving us a nice safe piece of code, allows the program to run under splint without warnings, even if someone else runs it without the command line switches.

Finland. Finland. Finland.

The next three issues are very simple so we shall cover them together (although in practice we actually stepped through each one in turn). We have (in order), an initializer with extraneous braces (29), an assignment inside a conditional (34) and (on the same line) the test expression itself which resolves to a non-boolean answer.

The braces problem does not show up under the compiler because strings in 'C' are simply arrays of characters, so an array of strings is just a bigger array (with NUL terminators at the end of each string). If the array were used more extensively, however, problems would soon arise.

The conditional assignment does not show up as a warning under gcc because there are two brackets around the expression. This trick, to stop compiler warnings under gcc, doesn't work under splint. And, because it should be lint-free we shall amend the code accordingly, thus we can use:

```
num = num_calc(message);
if (num > 0)
   {
      ... etc ...
```

The '> 0' not only provides a boolean result, but emphasizes the correct result we seek. Although the function does not currently return values less than zero, if it did (for error conditions, say), these would be picked up correctly too. (Remember that 'C' refers to all non-zero numbers as 'true').

Another quick run of splint and we're down to 5 warnings.

Some things in life are bad...

The next problem actually causes three issues. Sequentially speaking, the first error is what appears to be a simple type mismatch. However, remembering what we said earlier about changing types arbitrarily, we take a closer look at how this variable is deployed.

First off, the num_calc function returns an integer, and tries to assign it to an unsigned integer. So which should be changed, the function or the assignment? Since the function may return an error code as a negative number in the future it's not unreasonable to assume it should be a signed int.

The next problematic line (37) shows us the real crux of the problem: an unsigned value can not be negative by definition. This means the '< = 0' can never be true, which is the cause of our programming hanging.

Doh! Why didn't the programmer spot this? More to the point, perhaps: why doesn't gcc? An analysis of the algorithm shows us that the number needs to become negative in order for the loop to terminate (lines 35-37), and so we conclude that a signed integer is the way to go. Checking the third of these errors we notice that the printf format specifier is also wrong, confirming our suspicions.

Finland has it all

The last two errors are also connected. They both reference exported identifiers: one variable, one function. In 'C', it is possible to reference variables from one file in another by extern'ing them.

```
extern int mapping[];
```

While this is not necessarily a bad thing, it allows another file to corrupt the mapping data (or call our num_calc) without our permission. Generally, if the function is private to that file – make it private with the keyword 'static'. This, again, explains to the compiler what we mean, and not what we say.

Although it's a simple change, and may appear to some as inconsequential, it is very important and should not be ignored and allowed to fester.

...buttered scones for tea

And there you have it. A completely debugged and lint-free program. It has not taken a particularly long time to do it, but has provided a much stabler base from which to work, and exorcised many bad style demons that could confuse maintenance programmers in the future. This newly found confidence in the code will encourage further features to be added, and old ones enhanced.

As you can see, splint enables programmers to detect bugs before they become problems. It should find its way into the development cycle, along with -Wall as part of the build process, to shorten the bug-fix cycle, and so enabling developers to spend more time on new features. ■

Box 2: Never be rude

Whilst splint can highlight many of the semantic mistakes that gcc can not, it is by no means a stand alone or infallible program. Because it doesn't have to generate program code for the source, it can make (occasionally incorrect) assumptions about other parts of the code.

For example, it can miss situations where functions are used without format declarations. This can be fatal in situations where the return type of the function is a floating point number, and the implicit declaration will be deduced as an integer: which is incorrect. Fortunately, the compiler will spot this particular instance – so you must not be lulled into a sense of false security by running with lax compiler options.

Sometimes, the humanistic element of coding can also cause problems Splint is unable to detect; consider the source fragment below. Not only is Splint unable to find the errors, but usually a human being will also fail to notice them.

```
int a = 101;
int b = 020;
int c;
c = a/b;
```

Here, the result of c is not 101, but 0! This is because the value '101' is actually '10l', with a lower case 'L' at the end. The visual difference between '1' and 'l' is small, and very difficult to ascertain. This should be handled by enforcing coding standards that require the use of an upper case 'L', and commenting when such numbers are used, to ease the readability.

The same is true for numbers which are prefixed with zero, this will cause the 'C' compiler to treat them as octal numbers. This gives us, essentially:

```
int a = 10;
int b = 16;
```

So naturally, the integral result of 10/16 will be zero, causing a fairly severe bug.