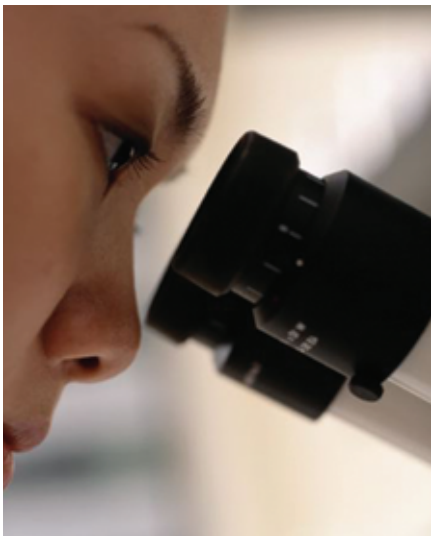
Checking Code Quality with Splint

Greater Detail



Splint parses C source code and discovers typical programming errors. There is no need to change the code to allow this, but you can achieve better results with a bit more effort. Developers can describe their programs in special annotations allowing Splint a better understanding of what is going on and helping the tool discover even more dirty code.

BY HERWART KIRAM

Without modifying the program code or using special flags, the static semantics checker, Splint [1], applies its sleuthing skills to discovering faulty code. An article in issue 27, February 2003 [2] described Splint's capabilities when simply thrown in at the deep end. Programmers can optimize their use of the program, provided they are not too busy for a bit more work that involves adding special annotations to their code. Splint evaluates the annotations to gain a better understanding of the source code, and this in turn puts the checker in a position to analyze a whole new range of problems. These include:

- Infinite loops
- Memory de-allocation errors (memory leaks)
- De-referencing null-pointers
- Name convention infringements
- Inappropriate interfaces
- Buffer overflows
- Inaccessible code

The C compiler itself is perfectly capable of indicating non-complex issues. But the more complex the pitfall the programmer falls for, the more effort tools have to put into discovering programming errors. Figure 1 juxtaposes the effort and return on effort for these tools: formal verification tools will discover more errors, but they also involve non-trivial effort. The effort involved in running Splint is minimal, but the tool still finds a lot of bugs.

Commands

There are a large number of options and parameters available for the Splint command. The syntax is as follows:

```
splint flags -f filename program
```

where *flags* is a list of options that enable or disable various checks. In contrast to most UNIX programs a minus sign – will disable the option, and a plus sign + will enable it. Additionally, there are shortcuts and checking levels that simultaneously affect a whole bunch of flags. Splint provides four distinct levels with increasingly strict checking criteria: *-weak*, *-standard* (Default), *-checks*, and *-strict* (see Box: Splint Checking Levels).

If the file *~/.splintrc* exists and is readable, Splint will use the default flags set in the file. The *-f filename* flag allows you to specify another file as default. The following call will check the *Example1.c* program in Listing 1:

```
splint -checks +boundswrite ➥
  -exportheader Example1.c
```

The parameters tell Splint to use the *checks* level without the *exportheader* check but enabling *boundswrite*.

Smart Comments

Splint derives its understanding of the meaning of individual code sections from the source code. Thus Splint cannot know whether a programmer actually intended to write what Splint regards as broken code, or if it is simply a programming error. The program developer can supply additional information in the form of special annotations that are passed to Splint.

Listing 1 contains some annotations of this type. The main function in line 3

Box: Splint Checking Levels

Weak: Weak check for typical, uncommented programs. At this level Splint does not discover modifies (side effects that occur due to hidden value modifications) and does not investigate macros. Values returned by functions can be ignored by their caller. *bool*, *int*, *char* and user-defined *enum* types are equivalent.

Standard: In addition to the checks performed at the weak level, Splint additionally checks whether the program uses released memory, derferences null-pointers, contains inaccessible code or infinite loops, and parses values returned by functions. This stage also checks whether macros are ok and if functions use all of their parameters. *bool*, *int*, and *char* type are not equivalent and cannot be mixed (without explicit type conversion).

Checks: This stricter check additionally ensures that functions exactly adhere to their interface definition. At this stage *enum* and *int* are also regarded as different types.

Strict: The strictest checking level is of limited use for real programs. The manpage promises a reward to the first person to write a real program that navigates this stage without provoking any warnings.

expects a list of arguments that it never uses. The code annotation /*@*unused* @*/ prevents Splint from outputting an error message. The program developer knows that the variables will not be used, but is using them on purpose. Code annotations always start with /*@ and are terminated by @*/. The following command displays a list of annotations:

```
splint -help annotations
```

Invalid pointers are a major hassle in software development and account for fifty percent of all errors. Splint can also help out with bugs of this type. A small but nasty bug has found its way into Listing 1 – line 9 de-allocates memory containing a string, and line 10 attempts to output precisely this string.

The example we supplied is non-critical and easily noticed, but in a real program several hundred lines of code could be separating the de-allocation and the illegal access. The memory address might have been overwritten by arbitrary data or be outside of the valid address range. Splint reacts with the following warning:

```
Example1.c:10:26: Variable ➥
dpointer used after being ➥
released
```

Destroyed Pointers

Things become more difficult when memory allocation and de-allocation occurs in a function instead of the main program. The program in Listing 2 request memory in the *allocmem()* routine. It expects string input from the user in *get_data()* and inverts this input in *reverse_data()*. It then outputs the string and de-allocates memory in *deallocmem()*. In the current form the program produces the following Splint error message, amongst others:

```
linttest.c:70:8: Implicitly ➥
temp storage char_p passed as ➥
only param: free (char_p)
```

Splint means that the allocated memory has not been de-allocated. The error message disappears when the /*@*only* @*/ annotation is inserted directly before *char** in line 11. Now Splint understands that this is an exclusive reference; in other words the pointer is forced to de-allocate the memory it references. There are three ways for the pointer to pass on this obligation to other pointers:

- Passing it to a function in a parameter. This uses the *only* attribute.
- Passing it to an external reference that also has the *only* attribute.
- Returning a value that also has the *only* attribute.

After passing on this obligation to another pointer, the original pointer can be regarded as dead, and should not be re-used. This allows Splint to ensure that the program de-allocates any memory it has allocated, exactly once.

To ensure that this works properly the obligation to de-allocate memory must be stipulated. This is caused by routines that allocate memory, such as *malloc()* or *calloc()*.

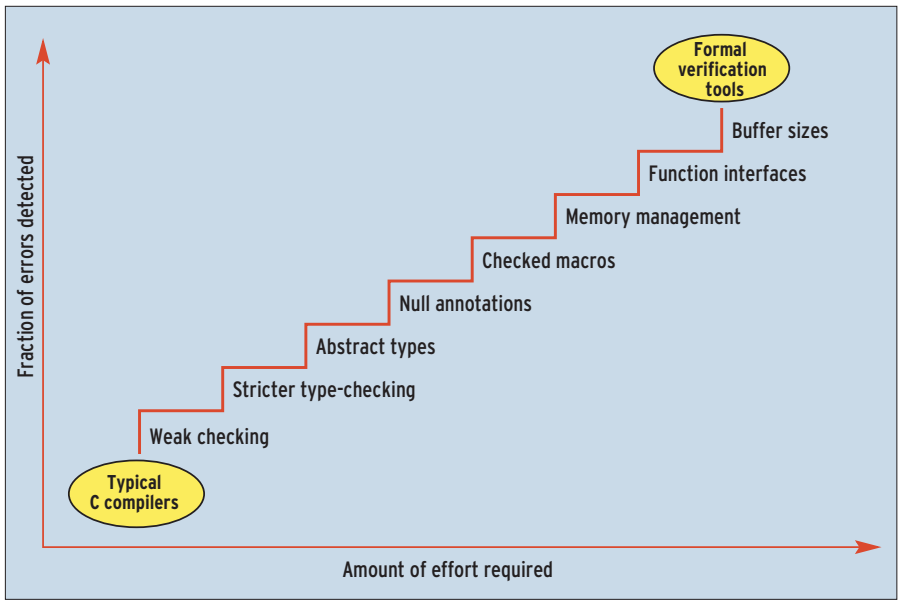


Figure 1: Most C compilers warn you about simple bugs. More complex error classes require more effort. Splint allows the programmer to select an appropriate checking level (diagram based on [3])

Annotated Libraries

Splint provides its own versions of standard library headers. These headers contain annotated code which Splint will automatically use when checking a program. The *malloc()* function looks like this:

```
/*@only@*/ /*@null@*/ ➥
void *malloc(size_t size);
```

The return value is a pointer that references a memory area that the caller must de-allocate (*only*). The pointer may be null. A quick look at the headers sup-

Listing 1: Example1.c

```
#include <stdlib.h>

int main(/*@unused@*/ int argc, /*@unused@*/ char **argv)
{
   char* dpointer=NULL;
   dpointer = (char*)calloc((size_t)20, (size_t)1);
   if (dpointer==NULL) return 1;
   strcpy (dpointer, "Hello World");
   free (dpointer);
   printf("Output: %s\n", dpointer); /* dpointer points to Nirvana */
   return 0;
}
```

plied with Splint, *standard.h* and *posix.h* can tell you a lot about annotated code.

The counterpart to *malloc()* – the *free()* function – looks like this:

```
void free(/*@only@*/ /*@out@*/
 /*@null@*/ void* ptr);
```

This function expects a pointer to a memory area that it will need to de-allocate (*only*, the function thus assumes responsibility). The calling function is not permitted to use this memory again as it has handed over responsibility for it. The pointer can be null, so the memory area may not have been initialized (*out*).

It is also possible to assign the *only* pointer to other pointers. To do so, the pointer must have the */*@temp@*/* attribute. Temp pointers are not permitted to allocate or de-allocate memory, and the program cannot access them after de-allocating a memory area. To ensure that this will work for non-annotated programs, Splint handles all pointers as temp by default.

Null Pointers

Null pointer access is a common pitfall in C programming. Unless otherwise specified, Splint assumes that a pointer can never be null. To allow this to happen, programmers need to specify the */*@null@*/* attribute. Again the *malloc()* function provides a good example, as it returns a null pointer if no memory is available. Splint will issue a warning if a program de-references the null pointer. The warning for Listing 2 is as follows:

```
linttest.c:24:23: Possibly null
 storage dpointer passed as non
-null param: get_data (dpointer)
```

You can stop Splint from complaining by re-instating line 23. This will allow Splint to recognize that the *If* condition allows the program to de-reference the null pointer.

Correct Interfacing

Functions use their interfaces to exchange information with the calling environment. Function prototypes specify the type and number of interfaces. It is good programming technique to document the arguments a function will modify and those it will leave untouched. The */*@modifies@*/* annotation allows you to tell Splint which arguments and global variables a routine can modify. Any discrepancies that occur may be indicative of a programming error.

Interface definitions also reproduce user-defined constants, similar to *const* in C++. Splint checks whether arguments not designated as modifiable are in fact modified at runtime. This check is enabled by setting *+mods* flag. But the fact that a function does not modify a modifiable argument can also indicate an error. To check for this case, you will need to set the *+mustmod* flag. These tests are available in *checks*.

We have assigned the modifies annotation to all of the routines in our sample program. As *get_data()* changes the global variable *S_num* although the

Listing 2: linttest.c

```
// Sample program 2
#include <stdlib.h>
#include <stdio.h>
#define BUFSIZE 200
// Name convention: all evaluated
   types start with
// "T_" and contain only small
   letters apart from this
/*@ +matchanyintegral +typeprefix
   T_&* @*/
static /*@null@*/ char*
   allocmem(void) /*@modifies
   nothing@*/ ;
static void deallocmem(char*) ;
static int get_data (char
   *outputdata) /*@modifies
   outputdata@*/;
static int reverse_data(char
   *inputdata) /*@modifies
   inputdata@*/;
static int S_num;
int main(/*@unused@*/ int argc,
   /*@unused@*/char **argv)
{
   char* dpointer=NULL;
   int char_cnt=0;
   int space_cnt=0;
   S_num=0;
   dpointer = allocmem();
// if (dpointer==NULL) return 0;
   char_cnt = get_data (dpointer);
   char_cnt = reverse_data
     (dpointer);
   printf ("Number of characters:
     %i Output: %s\n", char_cnt,
     dpointer);
   deallocmem (dpointer);
   if (char_cnt ==0)
   {
      return 0;
   }
   else
   {
      return 0;
   }
   return 0; // Code inaccessible
}
int reverse_data(char *inputdata)
{
   int i =0;
   char c;
   int a = strlen(inputdata);
   while (i<=a/2-1)
   {
      c=*(inputdata+i);
      *(inputdata+i) =
        *(inputdata+a-i-1);
      *(inputdata+a-i-1)=c;
//    i++;
   }
   return a;
}
int get_data (char *inputdata)
{
   int i=0;
   printf ("Enter a strings: ");
   (void) fgets (inputdata,
     BUFSIZE-1, stdin);
   S_num = strlen (inputdata);
   return i;
}
char* allocmem(void)
{
   return
(char*)malloc((size_t)BUFSIZE);
}
void deallocmem(char* char_p)
{
   free (char_p);
   return;
}
```

modifies annotation is missing, Splint issues the following warning:

```
linttest.c:58:2: Undocumented ➥
  modification of S_num: ➥
  S_num = strlen(outputdata)
```

Infinite Loops

Splint can test for infinite loops, although this will only work in extremely simple cases, such as the *reverse_data()* function in our sample program (see Listing 2). Splint issues the following warning:

```
linttest.c:43:9:Suspected ➥
infinite loop. No value used ➥
in loop test (i, a) is modified➥
 by test or loop body.
```

Unfortunately, this warning will disappear if you insert *i=3;* before the end of the while loop. The loop is still infinite, as the loop variable is a constant, but Splint no longer recognizes the problem. *i++;* would be preferable.

Table 1: Naming conventions

Category	
enumprefix	enumerated types *enum*
globprefix	global variables
typeprefix	user-defined types *typedef*
externalprefix	external names
localprefix	local names
constprefix	constants
protoparamprefix	parameters in function prototypes
Character Codes	
^	any capital letter (A-Z)
&	any small letter (a-z)
%	any character that is not a capital letter
~	any character that is not a small letter
$	any letter (A-Z, a-z)
#	any number 0-9
/	any letter or number (A-Z, a-z, 0-9)

Splint also checks code accessibility. The return statement at the end of *main()* can never be executed as both branches of the previous if condition contain return statements. This causes Splint to report a bug. To suppress the warning, you can insert a code annotation locally, such as the following line, to disable the check before the return statement:

```
/*@ -unreachable @*/
```

and then enable the check again using:

```
/*@ +unreachable @*/
```

This variant allows you to enable certain checks at critical points, at the same time avoiding redundant warnings and false positives. Line 8 in Listing 2 provides another example of this. The *matchanyintegral* flag prevents Splint from recognizing the assignment of *size_t* to *int* as an error.

Naming Conventions

Naming conventions are useful as they assure the readability of code for a team of developers working on it. It does not really matter what conventions you use, but it is important that each developer applies them consistently. Splint can also check for this. Some flags specify the appearance of various types of variables or user-defined types. These flags start with the name of the category (such as *local* for local variables) plus the *prefix* keyword. The pattern for the name then follows after a space character. The pattern uses metacharacters to specify the rules that apply to the name. Unfortunately, you cannot use regular expressions. For example:

```
/*@localprefix L_&* @*/
```

This example specfies that local variable must start with *L_* and can contain only small letters after this point. (* means that the previous character can appear any number of times). Table 1 shows the categories and metacharacters for these rules. Line 8 of Listing 2 shows another example. Any user-defined types must start with *T_*, followed by small letters.

Macros

The macro preprocessor is a powerful tool, but it does harbor a number of pitfalls. Macros simply replace text and thus contravene C syntax. A macro that calculates a square is a classical example of this:

```
#define Quadrat(x) x*x
```

This definition works fine, as long as only single values are passed to the macro, as in *Quadrat(i)*. Problems start to occur if you do things like *Quadrat(i+1)*. The preprocessor will convert this text to *i+1*i+1*, and this corresponds to *i+i+1*, rather than *(i+1)*(i+1)*. A call to Quadrat(i++) is also problematic, as the result of *i++*i++* will vary from one implementation to the next. If you launch Splint with the *+allmacros* flag, the tool can check the following, amongst other things:

- A macro parameter cannot be used with a decrement or increment operator.
- Macro parameters must be surrounded by brackets.
- Each macro parameter can be used exactly once only.

Buffer Overflows

Buffer overflows are particularly dangerous in C programs. Many exploits leverage bugs of this type; memory access outside the bounds envisaged by the programmer can thus cause serious problems. Splint uses internal variables called *maxSet* and *maxRead* to manage memory blocks, and creates these variables automatically for each vector. *maxSet* specifies the threshold for valid memory write access, *maxRead* the threshold for read access. This is quite sufficient for simple cases:

```
int myarray[10];
int i = 8;
myarray[i+4]=0;
myarray[i-12]=0;
```

If you have enabled the bounds checking option, Splint will discover that the first attempt to access the array will probably write past the end of the array:

```
Possible out-of-bounds store: ➤
  myarray[i + 4]
```

Unfortunately, the tool does not recognize negative indexing.

Static checks often do not reveal the actual index values at runtime. To avoid this, you can use the */*@ensure@*/* and */*@require@*/* code annotations to stipulate threshold values for parameters passed to function interfaces. However, this requires a great deal of effort, without actually guaranteeing success. It only makes sense that buffer overflow tests are not part of a specific checking level, but need to be enabled explicitly by setting the *+boundswrite* and *+boundsread* flags. The authors of Splint describe their own technique for recognizing possible buffer overflows with a static bounds checker without actually running the program (see [5]).

Enum and Numbers

In C it is permissible to assign integer values to variables of arbitrary enumerated (*enum*) types, instead of just using the declared values:

```
enum weekday =
  {Mo, Di, Mi, Do Fr, Sa, So};
weekday = 15;
```

Splint will report an error in this case, provided you set the *-checks* flag. The tool will also recognize use of variables before they have been defined, and let you know if the program ignores function return values.

Conclusion

The comprehensive User Manual for Splint [3] provides more detail on the program's capabilities. Splint is at its most valuable when integrated at the outset of a development process, and used consistently at any point after. It does not take longer to write annotated programs than to write programs without annotations. However, using code annotations does make a developer think about the meaning of a variable or parameter, and possibly add a code annotation. And will improve the quality of the program code. Splint is a big help when it comes avoiding errors, and it saves you a lot of debugging time.

Splint's biggest downer is the fact that it does not support C++. However, the tool is GPL and project leader, David Evans, has promised to support anyone prepared to write a C++ front-end. And that's what I call a meaningful task that would benefit the whole developer community. ■

INFO

[1] Splint homepage: *http://www.splint.org/*

[2] Steven Goodwin and Dean Wilson: "Walking Upright – Quality Code". Linux Magazine, Issue 27, p76

[3] Splint User Manual: *http://www.splint.org/manual/*

[4] FAQs: *http://www.splint.org/faq.html*

[5] Whitepaper on buffer overflows *http://www.cs.virginia.edu/~evans/usenix01-abstract.html*

THE AUTHOR

Herwart Kiram has been working as a software developer in the telecommunications industry for over ten years. He specializes in Linux and communication protocols.