16. lint

Introduction

The **lint** program examines C language source programs for a number of bugs and obscurities. It enforces the type rules of C language more strictly than the C compiler. It may also be used to enforce portability restrictions involved in moving programs between different machines and/or operating systems. It detects a number of legal but wasteful or error prone constructions. **lint** accepts multiple input files and library specifications and checks them for consistency.

Usage

The **lint** command has the form:

lint [*options*] *files ... library-descriptors ...*

where *options* are optional flags to control **lint** checking and messages; *files* are the files to be checked which end with **.c** or **.ln**; and *library-descriptors* are the names of libraries to be used in checking the program.

The options currently supported by the **lint** command are:

–a	Suppress messages about assignments of long values to variables that are not long.
–b	Suppress messages about break statements that cannot be reached.
–c	Only check for intra-file bugs; leave external information in files suffixed with **.ln**.
–h	Do not apply heuristics (which attempt to detect bugs, improve style, and reduce waste).
–n	Do not check for compatibility with either the standard or the portable **lint** library.
–o *name*	Create a lint library from input files named **llib–l***name*.**ln**.
–p	Attempt to check portability.
–u	Suppress messages about function and external variables used and not defined or defined and not used.
–v	Suppress messages about unused arguments in functions.

–x Do not report variables referred to by external declarations but never used.

When more than one option is used, they should be combined into a single argument, such as **–ab** or **–xha**.

The names of files that contain C language programs should end with the suffix **.c**, which is mandatory for **lint** and the C compiler.

lint accepts certain arguments, such as:

–lm

These arguments specify libraries that contain functions used in the C language program. The source code is tested for compatibility with these libraries. This is done by accessing library description files whose names are constructed from the library arguments. These files all begin with the comment:

```
/* LINTLIBRARY */
```

which is followed by a series of dummy function definitions. The critical parts of these definitions are the declaration of the function return type, whether the dummy function returns a value, and the number and types of arguments to the function. The VARARGS and ARGSUSED comments can be used to specify features of the library functions. The next section, "**lint** Message Types," describes how it is done.

lint library files are processed almost exactly like ordinary source files. The only difference is that functions that are defined in a library file but are not used in a source file do not result in messages. **lint** does not simulate a full library search algorithm and will print messages if the source files contain a redefinition of a library routine.

By default, **lint** checks the programs it is given against a standard library file that contains descriptions of the programs that are normally loaded when a C language program is run. When the **–p** option is used, another file is checked containing descriptions of the standard library routines which are expected to be portable across various machines. The **–n** option can be used to suppress all library checking.

lint Message Types

The following paragraphs describe the major categories of messages printed by **lint.**

Unused Variables and Functions

As sets of programs evolve and develop, previously used variables and arguments to functions may become unused. It is common for external variables or even entire functions to become unnecessary and yet not be removed from the source. Although these types of errors rarely cause working programs to fail, they are a source of inefficiency and make programs harder to understand and change. Also, information about such unused variables and functions can occasionally serve to discover bugs.

lint prints messages (unless suppressed by the **–u** or **–x** option) about variables and functions which are defined but not otherwise mentioned.

Certain styles of programming may permit a function to be written with an interface where some of the function's arguments are optional. Such a function can be designed to accomplish a variety of tasks depending on which arguments are used. Normally **lint** prints messages about unused arguments; however, the **–v** option is available to suppress the printing of these messages. When **–v** is in effect, no messages are produced about unused arguments except for those arguments which are unused and also declared as register arguments. This can be considered an active (and preventable) waste of the register resources of the machine.

Messages about unused arguments can be suppressed for one function by adding the comment:

```
/* ARGSUSED */
```

to the source code before the function. This has the effect of the **–v** option for only one function. Also, the comment:

```
/* VARARGS */
```

can be used to suppress messages about variable number of arguments in calls to a function. The comment should be added before the function definition. Sometimes, it is desirable to check the first several arguments and leave the later arguments unchecked. This can be done with a digit giving the number of arguments which should be checked. For example, the comment:

```
/* VARARGS2 */
```

will cause only the first two arguments to be checked.

When **lint** is applied to some but not all files out of a collection that are to be loaded together, it issues complaints about unused or undefined variables. This information is, of course, more distracting than helpful. Functions and variables that are defined may not be used; conversely, functions and variables defined elsewhere may be used. The **–u** option suppresses the spurious messages.

Set/Used Information

lint attempts to detect cases where a variable is used before it is set. **lint** detects local variables (automatic and register storage classes) whose first use appears physically earlier in the input file than the first assignment to the variable. It assumes that taking the address of a variable constitutes a "use" since the actual use may occur at any later time, in a data dependent fashion.

The restriction to the physical appearance of variables in the file makes the algorithm simple and quick to implement since the true flow of control need not be discovered. It does mean that **lint** can print error messages about program fragments that are legal, but these programs would probably be considered bad on stylistic grounds. Because static and external variables are initialized to zero, no meaningful information can be discovered about their uses. The **lint** program does deal with initialized automatic variables.

The set/used information also permits recognition of those local variables that are set and never used. These form a frequent source of inefficiencies and may also be symptomatic of bugs.

Flow of Control

lint attempts to detect unreachable portions of a program. It will print messages about unlabeled statements immediately following **goto**, **break**, **continue**, or **return** statements. It attempts to detect loops that cannot be left at the bottom and to recognize the special cases **while(1)** and **for(;;)** as infinite loops. **lint** also prints messages about loops that cannot be entered at the top. Valid programs may have such loops, but they are considered to be bad style. If you do not want messages about unreached portions of the program, use the **–b** option.

lint has no way of detecting functions that are called and never return. Thus, a call to **exit** may cause unreachable code which **lint** does not detect. The most serious effects of this are in the determination of returned function values (see "Function Values"). If a particular place in the program is thought to be unreachable in a way that is not apparent to **lint**, the comment:

```
/* NOTREACHED */
```

can be added to the source code at the appropriate place. This comment will

inform **lint** that a portion of the program cannot be reached, and **lint** will not print a message about the unreachable portion.

Programs generated by **yacc** and especially **lex** may have hundreds of unreachable **break** statements, but messages about them are of little importance. There is typically nothing the user can do about them, and the resulting messages would clutter up the **lint** output. The recommendation is to invoke **lint** with the **–b** option when dealing with such input.

Function Values

Sometimes functions return values that are never used. Sometimes programs incorrectly use function values that have never been returned. **lint** addresses this problem in a number of ways.

Locally, within a function definition, the appearance of the statements:

```
return(  expr );
```

and:

```
return ;
```

is cause for alarm; **lint** will give the message:

```
function name has return(e) and return
```

The most serious difficulty with this is detecting when a function return is implied by flow of control reaching the end of the function. This can be seen with a simple example:

```
f ( a ) {
            if ( a ) return ( 3 );
            g ();
        }
```

Notice that, if **a** tests false, **f** will call **g** and then return with no defined return value; this will trigger a message from **lint**. If **g**, like **exit**, never returns, the message will still be produced when in fact nothing is wrong. This comment in the source code will cause the message to be suppressed:

```
/*NOTREACHED*/
```

In practice, some potentially serious bugs have been discovered by this feature.

On a global scale, **lint** detects cases where a function returns a value that is

sometimes or never used. When the value is never used, it may constitute an inefficiency in the function definition that can be overcome by specifying the function as being of type (void), as in:

```
(void) fprintf(stderr,"File busy. Try again later!\n");
```

When the value is sometimes unused, it may represent bad style (e.g., not testing for error conditions).

The opposite problem, using a function value when the function does not return one, is also detected. This is a serious problem.

Type Checking

lint enforces the type checking rules of C language more strictly than the compilers do. The additional checking is in four major areas:

- across certain binary operators and implied assignments
- at the structure selection operators
- between the definition and uses of functions
- in the use of enumerations

There are a number of operators which have an implied balancing between types of the operands. The assignment, conditional (**?:**), and relational operators have this property. The argument of a **return** statement and expressions used in initialization suffer similar conversions. In these operations, **char**, **short**, **int**, **long**, **unsigned**, **float**, and **double** types may be freely intermixed. The types of pointers must agree exactly except that arrays of *x*s can, of course, be intermixed with pointers to *x*s.

The type checking rules also require that, in structure references, the left operand of the **->** be a pointer to structure, the left operand of the **.** be a structure, and the right operand of these operators be a member of the structure implied by the left operand. Similar checking is done for references to unions.

Strict rules apply to function argument and return value matching. The types **float** and **double** may be freely matched, as may the types **char**, **short**, **int**, and **unsigned**. Also, pointers can be matched with the associated arrays. Aside from this, all actual arguments must agree in type with their declared counterparts.

With enumerations, checks are made that enumeration variables or members are not mixed with other types or other enumerations and that the only operations applied are =, initialization, ==, !=, and function arguments and return values.

If it is desired to turn off strict type checking for an expression, the comment:

```
/* NOSTRICT */
```

should be added to the source code immediately before the expression. This comment will prevent strict type checking for only the next line in the program.

Type Casts

The type cast feature in C language was introduced largely as an aid to producing more portable programs. Consider the assignment:

```
p = 1 ;
```

where **p** is a character pointer. **lint** will print a message as a result of detecting this. Consider the assignment:

```
p = (char *)1 ;
```

in which a cast has been used to convert the integer to a character pointer. The programmer obviously had a strong motivation for doing this and has clearly signaled his intentions. Nevertheless, **lint** will continue to print messages about this.

Nonportable Character Use

On some systems, characters are signed quantities with a range from –128 to 127. On other C language implementations, characters take on only positive values. Thus, **lint** will print messages about certain comparisons and assignments as being illegal or nonportable. For example, the fragment:

```
char c;
        . . .
if( (c = getchar()) < 0 ) . . .
```

will work on one machine but will fail on machines where characters always take on positive values. The real solution is to declare *c* as an integer since **getchar** is actually returning integer values. In any case, **lint** will print the message:

```
nonportable character comparison
```

A similar issue arises with bit fields. When assignments of constant values are made to bit fields, the field may be too small to hold the value. This is especially true because on some machines bit fields are considered as signed quantities. While it may seem logical to consider that a two-bit field declared of type **int** cannot hold the value 3, the problem disappears if the bit field is declared to have type **unsigned**.

Assignments of longs to ints

Bugs may arise from the assignment of **long** to an **int**, which will truncate the contents. This may happen in programs which have been incompletely converted to use **typedefs**. When a **typedef** variable is changed from **int** to **long**, the program can stop working because some intermediate results may be assigned to **int**s, which are truncated. The **–a** option can be used to suppress messages about the assignment of **long**s to **int**s.

Strange Constructions

Several perfectly legal, but somewhat strange, constructions are detected by **lint**. It is hoped the messages encourage better code quality, clearer style, and may even point out bugs. The **–h** option is used to suppress these checks. For example, in the statement:

```
*p++ ;
```

the * does nothing. This provokes the message:

```
null effect
```

from **lint**. The following program fragment:

```
unsigned x ;
if( x < 0 ) . . .
```

results in a test that will never succeed. Similarly, the test:

```
if( x > 0 ) . . .
```

is equivalent to:

```
if( x != 0 )
```

which may not be the intended action. **lint** will print the message:

```
degenerate unsigned comparison
```

in these cases. If a program contains something similar to:

```
if( 1 != 0 ) . . .
```

lint will print the message:

```
constant in conditional context
```

since the comparison of 1 with 0 gives a constant result.

Another construction detected by **lint** involves operator precedence. Bugs which arise from misunderstandings about the precedence of operators can be accentuated by spacing and formatting, making such bugs extremely hard to find. For example, the statements:

```
if( x&077 == 0 ) . . .
```

and:

```
x<<2 + 40
```

probably do not do what was intended. The best solution is to parenthesize such expressions, and **lint** encourages this by an appropriate message.

Old Syntax

Several forms of older syntax are now illegal. These fall into two classes: assignment operators and initialization.

The older forms of assignment operators (e.g., =+, =–, ...) could cause ambiguous expressions, such as:

```
a =-1 ;
```

which could be taken as either of the following:

```
a =- 1 ;
```

```
a = -1 ;
```

The situation is especially perplexing if this kind of ambiguity arises as the result of a macro substitution. The newer and preferred operators (e.g., +=, –=, ...) have no such ambiguities. To encourage the abandonment of the older forms, **lint** prints messages about these old-fashioned operators.

A similar issue arises with initialization. The older language allowed:

```
int x 1;
```

to initialize *x* to 1. This also caused syntactic difficulties. For example, the initialization:

```
int x ( -1 ) ;
```

looks somewhat like the beginning of a function definition:

```
int x ( y ) { . . .
```

and the compiler must read past *x* to determine the correct meaning. Again, the problem is even more perplexing when the initializer involves a macro. The current syntax places an equals sign between the variable and the initializer:

```
int x = -1 ;
```

This is free of any possible syntactic ambiguity.

Pointer Alignment

Certain pointer assignments may be reasonable on some machines and illegal on others due entirely to alignment restrictions. **lint** tries to detect cases where pointers are assigned to other pointers and such alignment problems might arise. The message:

```
possible pointer alignment problem
```

results from this situation.

Multiple Uses and Side Effects

In complicated expressions, the best order in which to evaluate subexpressions may be highly machine dependent. For example, on machines in which the stack runs backwards, function arguments will probably be best evaluated from right to left. On machines with a stack running forward, left to right seems most attractive. Function calls embedded as arguments of other functions may or may not be treated similarly to ordinary arguments. Similar issues arise with other operators that have side effects, such as the assignment operators and the increment and decrement operators.

So that the efficiency of C language on a particular machine is not unduly compromised, the C language leaves the order of evaluation of complicated expressions up to the local compiler. In fact, the various C compilers have considerable differences in the order in which they will evaluate complicated expressions. In particular, if any variable is changed by a side effect and also used elsewhere in the same expression, the result is explicitly undefined.

lint checks for the important special case where a simple scalar variable is affected. For example, the statement:

```
a[i] = b[i++];
```

will cause **lint** to print the message:

```
warning: i evaluation order undefined
```

to call attention to this condition.