Thinkage Ltd.
LINT Reference Manual

Last Revision: November 11, 1995

Table of Contents

Chapter 1

Introduction

LINT is a program that examines C source code and makes note of "irregularities" in the code. When LINT was first implemented, its primary function was locating bugs and inefficiencies. However, as the C programming language spread to a variety of machines and systems, LINT was enhanced to locate deviations from the strict definition of C.

The ANSI standard for C now provides the "official" definition of C. However, the ANSI standard does not entirely describe the behavior of C implementations, because the standard allows certain operations to be performed in a system-dependent way. If a program is written in conformance with the ANSI standard, avoiding such system-dependent features, we say that the program is written in *Strict C*.

Ideally, all programs should be written in Strict C, since these programs are maximally portable. One of the major functions of LINT is to detect code constructs which do not conform with Strict C. Such constructs are often valid and can be used in working programs, but they are not truly portable and may be indications of loose programming style. LINT shows where you have written non-portable code; for portability, you should try to remove such code.

Before we begin, we should note that it is inevitable that LINT will miss some problems and will also complain about code that turns out to be valid. The designers have tried to chart a course between too much laxness (which may miss significant irregularities) and too much nit-picking (which produces quantities of irrelevant output that you will likely ignore). Thus we have designed this version of LINT to report situations that are *usually* signs of errors. While odd constructions and unusual programming style can confuse LINT from time to time, the majority of material that LINT locates should deserve the programmer's attention.

Chapter 2
LINT Directives

LINT understands all the usual C preprocessing directives (e.g. **`#include, #define,`** etc.). In addition, LINT recognizes a number of other directives that have the same sort of format:

```
#pragma aligned
#pragma argsused
#pragma notreached
#pragma optresult
#pragma used
#pragma varargs
```

These directives tell LINT about special features of your source code and allow LINT to provide more meaningful diagnostic messages. The usage of each LINT directive is explained later in the manual.

ANSI C compilers should not reject these special LINT directives; the ANSI standard says that C compilers are supposed to ignore any **`#pragma`** directives they do not understand. However, non-ANSI C compilers may issue diagnostic messages for these **`#pragma`** directives, and even some ANSI C compilers may issue warnings about unrecognized **`#pragma`**s.

Chapter 3
LINT Output

LINT generally produces quite a lot of output. By default the output is written to the terminal, but it can be redirected to a file using the standard output redirection constructions on the LINT command line.

The first lines of output from LINT are diagnostic messages comparable to those that might be produced by a C compiler. These diagnostics describe easily detected things like syntax errors.

Following this comes output that is unique to LINT. Most lines in the output have the form

```
filename,lineno: class: text
```

where `filename` is the name of one of the source files that LINT is scanning and `lineno` is a line number within that file. The `class` field tells the type of problem found; see "Message Classes" below for more information. The `text` part of the message provides specific information about the problem.

Some lines may have the form

```
libname: class: text
```

where `libname` is the name of a LINT Library. For more information on LINT libraries, see the last section of this manual.

Message Classes

Each message produced by LINT is labelled with a string indicating what kind of problem the message describes. The following message classes are recognized:

`Error:`

Same as error messages produced by the compiler: code so incorrect that there is no way to generate any sort of object code.

`Warning:`

Marks a construct that is invalid but a compiler could still generate some kind of object code. For example, the ANSI standard does not allow assignments between pointers of different types, but most compilers can still generate sensible object code for such an operation.

`Unusual:`

Marks a construct which is valid, but which is not often seen in C programs. For example,

```
i=i;
```

is valid but suspicious, possibly indicating some problem in your code.

`Note:`

Marks a construct which is invalid but always works properly on the current machine. For example,

```
printf("%ld",1);
```

is invalid since `%ld` requires a **`long`** argument but `1` is just **`int`**. However, this always works properly on machines where **`long`** and **`int`** are the same size.

`MachDepd:`

Marks a valid construct whose behavior is machine-dependent. For example, the result of `(-7/2)` is machine-dependent, since the ANSI standard allows this kind of division to truncate either towards or away from zero.

`Extension:`

Marks a construct that is an extension to the ANSI standard.

`Efficiency:`

Marks code that has been written in a particularly inefficient way, or using constructs that are likely to execute slowly.

`NotMinimal:`

Marks a construct which is valid ANSI C but may not be accepted by older C compilers (e.g. **`void *`** pointers).

`Info:`

Provides cross-reference information while using LINT.

`IntrnErr:`

An internal error in LINT itself (for example, a table overflow).

Output Grouping

LINT output is grouped according to the functions of the source code.

The first line of a group of messages gives the name of a function or an external variable. After that come all the messages pertaining to that function or variable. The messages end with a blank line, followed by the group of messages for the next function or variable.

Groups are sorted in alphabetical order according to the name of the function or variable.

The first line of every function group gives the name of the function and the type of value that the function returns. If LINT has found a definition for the function, the first line of the function group is

```
Function "NAME" returns "TYPE".
```

where `NAME` is the name of the function and `TYPE` is the type of value that the function returns.

If LINT can't find a definition for the function, LINT looks at the first *reference* to the function that appears in the source code and prints

```
Function "NAME" is assumed to return "TYPE".
```

where `TYPE` is the type suggested by the way in which the function is used. In keeping with the rules of C, LINT assumes that a function returns **`int`** if the function is not explicitly declared to have a different type. If the first reference to the function does not use a return value from the function, the message is

```
Function "NAME" assumed to return no value.
```

If LINT scans a function definition and finds that there are no **`return`** statements that return a value, LINT outputs the message

```
Function "NAME" returns no value.
```

You should distinguish between this message and

```
Function "NAME" returns void.
```

The **`void`** message appears when a function is explicitly declared with the **`void`** keyword. The `"no value"` message appears when a function definition has no **`return`** statements.

The first appearance of an external variable is treated in much the same way as the first appearance of a function. You will see a message like

```
External "NAME" is "TYPE".
```

or

```
External "NAME" is assumed to be "TYPE".
```

With one exception, the messages we have described so far are always issued, whether or not there are errors. The exception is when a symbol appears in a LINT library (described later on) and is not referenced in normal source code. Messages about library functions and variables only appear if the symbol is used in source code.

Other messages in LINT output describe situations that may be errors. These messages refer to difficulties within the function with which they are grouped.

Type Information

At the end of its output, LINT displays a block of information describing the types of the program. A typical line of output is

```
filename,lineno: type
```

where `filename` is the name of the file where the type was defined and `lineno` is the line number where the definition began.

LINT provides information about any **`typedef`**s, enumerated types, structures, and/or unions that were involved in the messages produced. This lets you locate the definitions of these types.

If a type appears twice in this list, it usually means that there is more than one definition for the same type and the types are not assignment-compatible.

LINT also checks for structures and unions which are used in functions and which have been incompletely defined. Incomplete definitions are acceptable as long as there is a complete definition somewhere. All complete definitions have to be identical. If complete definitions conflict with each other, LINT issues a diagnostic beginning with

```
filename : type
```

where `filename` is the name of a file where the type was used and found to conflict with another definition.

Chapter 4
LINT Conventions

A function is *defined* by the function header that actually starts the function. This tells the type of value that the function returns and describes the arguments of the function. An external variable is *defined* by a declaration for the variable that is outside the scope of all functions and that does not include the keyword `extern`.

A function is *declared* by a declaration or function call inside another function. An external variable is *declared* by a declaration that includes the keyword `extern`. Such a variable declaration does not allocate space for the variable; it merely describes the type of the variable and presumes that the variable is defined elsewhere.

A variable is *set* if it is assigned a value or if a pointer to the variable is taken. Similarly, a variable is *used* if its value is used or if a pointer is taken. (You might wonder why taking a pointer marks a variable as both "set" and "used". The answer is that once a pointer to a variable is taken, LINT can't keep track of assignments or uses of the variable through the pointer. Once the pointer is taken, LINT can only assume that the data object may have been both set and used.)

Chapter 5
Unused Definitions and Declarations

One of the easiest problems for LINT to find is a variable or function that is defined but not used or called in the rest of the program. Such items can usually be deleted, since they are not performing any function in the program. The diagnostic messages for such situations are listed and explained below.

`Function "NAME" is never called.`

The given function was defined in the source code but never called.

`External "NAME" is never used.`

The given external variable was defined but never used.

`External "NAME" is defined but never set or used.`

This is essentially the same as the previous message.

`External "NAME" is set but never used.`

The given external variable was assigned a value, but then was never used for anything.

Messages are *not* generated if a symbol in a LINT library is unused.

Marking Symbols As Used

Sometimes, you may intentionally declare a variable without using it. For example, suppose a variable is only used inside a **`#if`** block and that block of code is not compiled because the **`#if`** condition is not met. LINT would normally issue an error message for this condition, but you might not consider this situation an error.

To avoid this sort of message, you can add the LINT directive

```
#pragma used NAME NAME NAME ...
```

to your source code. This tells LINT that the names listed in the directive are used by your program, even if it doesn't look like they are.

Chapter 6

Undefined or Undeclared Data Objects

LINT uses a simple-minded way to find places where undefined variables are used: a variable is assumed to be undefined if a statement using the variable appears in the code before the variable is set. Of course, it is possible to construct programs where this approach fails. With the use of spaghetti-like **`goto`**s, the top-down technique can be outwitted.

Because static and external variables are automatically initialized to zero by the compiler if they are not initialized explicitly, LINT does not pick up problems with these variables.

The following diagnostics pertain to data objects that are used before they are set, defined, or declared.

`Function "NAME" is called but never defined.`

The source code contains a call to the given function, but the function is not defined anywhere in the code.

`"NAME" is set but never defined.`

The given variable has been assigned a value, but a proper definition does not appear in the source code.

`"NAME" is used but never defined.`

The value of the given variable was used, but the variable was never defined.

`"NAME" is used but never defined or set.`

The value of the given variable has been used, but the variable was neither defined nor set.

`"NAME" is defined and used but never set.`

The given variable is defined and used, but never set. This message is only issued for local variables, since all external variables are automatically initialized to zero (if no explicit initialization is specified).

Chapter 7
Unreachable Code

LINT attempts to detect parts of the source code that cannot be reached (for example, unlabelled statements following a **`goto`**). It also attempts to find loops that can never be exited from the bottom, such as **`while(1)...`** At the same time, LINT finds loops that cannot be entered from the top.

The message that is issued for detected unreachable code is

```
Non reachable code at "TEXT".
```

where `TEXT` is the source code that cannot be reached.

Marking Unreachable Code

LINT cannot detect functions which are called and never return (for example, ones that terminate execution via `exit`). This means that it is possible for LINT to miss some unreachable code, as in

```
g()
{
    exit(-1);
}
f()
{
    ...
    g();
    /* everything that follows
     * is unreachable */
    ...
}
```

To mark this kind of situation, you can use the LINT directive

```
#pragma notreached
```

When this directive appears in your source code, LINT regards any source code following the directive as unreachable, up to the end of the block or the next statement label. Appropriate diagnostic messages will be issued.

Chapter 8
Function Return Values

LINT reports functions that contain both

```
return( expression );
```

and

```
return;
```

This kind of situation usually results in errors (since the calling function may assume a value is being returned but one of the **`return`** statements does not return a value).

Correspondingly, LINT makes note of whether or not a call to a function makes use of a return value from the function. If the caller expects a value but the function does not return one, it is clearly a bug. If the function returns a value but the the caller does not use it, it could be an error, an inefficiency, or sloppy programming style.

If a function definition contains no **`return`** statements that return a value, LINT regards the function type as **`int`** unless it is explicitly declared differently. If LINT finds a reference to the function before it finds the function definition, it assumes that the function returns a value unless the function is explicitly declared as **`void`**. Programmers who are used to looser controls than this will find themselves with a large number of diagnostic messages.

The messages related to function return values are listed below.

`Value is used but none returned.`

A function did not return a value, but the statement that called the function behaved as if a value was returned.

`Function NAME has no return value`

A function was defined as returning a value (the function is not **`void)`** but it did not contain a

```
return(expression)
```

statement.

`Return value ignored.`

The given function contains a

```
return(expression);
```

statement, but the caller never uses the return result.

`Return value sometimes ignored.`

This is similar to the last situation, except that the return result is sometimes used, sometimes not.

Ignoring Return Values

There is always the possibility that you want to write a function whose return value can be ignored. The C library has several functions whose return value is superfluous. For example,

```
strcpy(A,B)
```

copies string `B` into string `A` and returns a pointer to `A`. This is unnecessary, since the caller already has a pointer to `A`.

LINT lets you mark functions with optional return values by using a directive similar to the **#**directives of the C preprocessor.

```
#pragma optresult
```

can be placed immediately before the definition of a function that returns an optional result. LINT then recognizes that this function returns a result that can be ignored; LINT does not give error messages if the result is ignored.

Chapter 9

Type Checking

LINT enforces stronger type-checking than most compilers do. In expressions where different types of data are intermixed, LINT observes the strict conventions for conversions of data and requires that everything else be properly cast.

LINT checks for type incompatibilities in three places:

(a) in the arguments passed to functions
(b) in the return values received by functions
(c) in structure manipulations

For fullest type-checking, the code should use prototypes wherever possible. LINT also checks for situations in which some function calls take place with a prototype in scope while others do not.

When arguments are passed to functions, LINT issues a diagnostic if the argument type passed by the caller differs from the argument type expected by the called function. Similarly, LINT complains if the argument type returned by the called function differs from the argument type expected by the caller. The messages associated with this situation are given below.

`Argument N is "TYPE1", but declared as "TYPE2".`

> When the given function was called, the caller passed a value of type TYPE1 for argument number N, but the called function expected a value of TYPE2. For example, you might see
>
> `Argument 2 is "int", declared "unsigned int".`
>
> This indicates that the caller passed an integer value as the second argument of the function, but the called function expected an unsigned integer.

`Argument N is "TYPE", but declared as a different "TYPE"`

> This is an odd situation where the definition and a declaration give the argument different types, but the strings used to print out `TYPE` turn out to be the same. For example, suppose that a function has a local **`struct`** `X`, but there is also an external **`struct`** `X` which is not compatible with the local one. If the program invalidly tries to combine external and local variables of type X, you get the above message; even though the types have the same name, they are different. The same sort of situation may happen with other LINT error messages.

`"NAME" is redeclared as function type "TYPE".`

When `NAME` was first declared, it was said to return a value of one type. Now `NAME` has been declared or used as a function returning a different type of value.

`"NAME" declared as returning "TYPE"`

The function `NAME` has been declared as returning a `TYPE` that is different than its definition.

`"NAME" declared as "TYPE"`

The variable `NAME` has been declared with a `TYPE` that is different than its definition.

`"NAME" is redefined as function.`

`NAME` was originally declared as a variable, but is now being defined as a function.

LINT also issues a message whenever a data object is implicitly shortened. For example, suppose we have

```
long l;
int i;
  /* stuff */
i = l;
```

The C compiler automatically shortens the long "l" and assigns the result to "i". However, LINT flags this as a potential problem (since different integer sizes on different machines may affect how this works). No error message is issued if the conversion is done explicitly, as in

```
i = (int) l;
```

As a final note about type-checking, LINT occasionally gets some complicated types wrong. For example, warnings about structures that contain arrays of structures do not give the right type (namely **`struct`**). The same problem occurs when LINT is not given the internal structure of a **`struct`**. For example, it is valid for a source module to declare

```
struct ABC *ptr;
```

without describing the internal structure of the `ABC` **`struct`**. In such cases, LINT does not have enough information to determine if code is valid, so the warning messages may not be entirely accurate.

Chapter 10
Function Declarations

LINT compares all the available prototype declarations for a function to make sure that the prototypes match *exactly*. For example, LINT notes situations where one prototype gives **`void *`** as the type of an argument while another types the same argument as **`char *`**. In practice, there is no difference between these two pointer types; however, LINT makes note of the situation because it may be indicative of an error.

In addition to checking that the argument types expected by a called function match the argument types passed by the caller, LINT also checks that the *number* of arguments expected by the called function matches the number of arguments passed by the caller. If there is a mismatch, it prints

```
        Called with N arguments, requires M
```

where N is the number of arguments that the caller passes and M is the number of arguments that the function expects.

Whenever possible, LINT checks to see that argument declarations in function prototypes agree exactly with the declarations in the function definition. If the declarations do not agree, LINT outputs

```
        Prototype argument N is "TYPE" but declared as "TYPE"
```

LINT expects argument declarations to match exactly. For example, the **`void *`** type is considered assignment-compatible with all pointer types and therefore there is not a conflict between **`void *`** and another pointer type. Nevertheless, LINT points out the difference, simply to warn you that there seems to be a discrepancy in your code.

Finally, LINT makes sure that the number of arguments in a prototype declaration matches the number in the function definition. If not, LINT prints

```
        Prototyped with N arguments, but requires M
```

where `M` and `N` are both integers.

The #pragma varargs Directive

In earlier versions of C it was valid to pass more arguments than the function definition specifies or to pass fewer arguments. This is not valid in ANSI C unless you use the

"`...`" construct in the prototype; however, older code may still make use of such calling sequences and most compilers still handle the situation properly.

In order to handle all the possibilities of this situation, LINT introduces the **`#pragma varargs`** directive. The directive has two forms.

```
#pragma varargs N
```

indicates that the next function to be defined can take a variable number of arguments but must have a minimum of N arguments. For example, we might write

```
#pragma varargs 3
int Mini(N,a,b,c,d);
int N,a,b,c,d;
{ ....
```

to declare a function `Mini` that must take at least three arguments and can have more. In this case, LINT does not issue an error message if you have at least three integer arguments present.

With this form of the **`#pragma varargs`** directive, LINT typechecks all the arguments that are present. In the above example, LINT ensures that all the arguments being passed are integers.

The second form of the directive is

```
#pragma varargs N M
```

where both `N` and `M` are integers. This indicates that the next function to be defined must have a minimum of `N` arguments, and LINT should typecheck up to `M` arguments if they appear. For example, you can imagine a `printf`-like function that takes a format string plus one or more values to print. This could be preceded with

```
#pragma varargs 2 1
```

showing that the function must always have two arguments (a format string and at least one output value) but that only the first argument should be typechecked (since the output value(s) need not have a fixed type).

The directive can also take the form

```
#pragma varargs printf
#pragma varargs scanf
```

These forms may be used to indicate functions that take `format` strings comparable to the strings of `printf` and `scanf` (respectively). LINT compares the placeholders in the `format` string to the types of the arguments that follow the `format` string, and reports any conflicts between types. For example, it tells you if the argument corresponding to a `%s` placeholder is an **`int`** value (incompatible with `%s`).

The #pragma argsused Directive

Some functions are able to use all the arguments they are passed without actually referring to every argument by name. For example, consider a function `Max10` which can accept up to 10 **`int`** arguments and return the value of the largest argument. The first argument passed to `Max10` tells how many values have actually been passed. This could be defined with

```
int Max10(N,a,b,c,d,e,f,g,h,i)
int N,a,b,c,d,e,f,g,h,i;
{
    int j,m,*p;
    m = a;
    p = &a;
    for (j=1; j < N; ++j)
        if (p[j] > m) m = p[j];
    return(m);
}
```

As you can see, the function does not refer to the parameters `b`, `c`, etc. by name. However, it does look at their values, since it walks through the stack using offsets from `a`. Therefore the values are used.

This sort of coding practice is a violation of the ANSI standard. The standard doesn't let you use subscripts to access memory locations outside the bounds of a single object, so `p[j]` is invalid if `j` is not zero. On the other hand, many compilers still accept code like this and older programs may use it.

For such programs, you can avoid some diagnostic messages from LINT by putting the LINT directive

```
#pragma argsused
```

on a line preceding the function, as in

```
#pragma argsused
int Max10(N,a,b,c,d,e,f,g,h,i)
   ...
```

This tells LINT that the function uses all its arguments, even if it doesn't appear to.

Chapter 11

Alternate Forms

For compatibility with the Bell Labs' version of LINT, this version of LINT may accept some directives in the form of comments. These are only recognized if the `+ControlComments` option is specified on the LINT command line.

```
/*VARARGSN*/
```

(where N is an integer) is automatically converted to

```
#pragma varargs 0 N
```

For example,

```
/*VARARGS3*/
```

is equivalent to

```
#pragma varargs 0 3
```

In addition,

```
/*VARARGS*/
```

(with no value N) is equivalent to

```
#pragma varargs 0
```

In addition to `VARARGS`, the comment

```
/*ARGSUSED*/
```

is automatically converted to

```
#pragma argsused
```

and the comment

```
/*NOTREACHED*/
```

is equivalent to

```
#pragma notreached
```

In all cases, the comment form may not contain white space (blanks, tabs, or new-lines). All alphabetic characters must be in upper case and there can be nothing else in the comment except the keyword and any number that should follow.

Chapter 12

Miscellaneous Notes

LINT always uses the full name of functions and variables, and always distinguishes between upper and lower case letters. This is the way that C compilers are supposed to work as well. However, some compilers or loaders truncate long names to a certain number of characters, and some compilers or loaders do not distinguish between the case of letters in names. For this reason, LINT issues the following warnings.

`NAME1 and NAME2 not unique in first N caseless characters.`

N is the number of characters to which names may be truncated to meet loader requirements.

`NAME1 and NAME2 not unique without case distinction.`

This applies to names like `VAR`, `var`, and `Var` used in the same program.

LINT detects redeclaration of functions and variables defined outside the scope of any function. The set of messages dedicated to this kind of problem are given below.

`NAME redeclared.`

A given variable has been invalidly declared in a second place.

`NAME retyped as "TYPE".`

A given variable has been used as if it had a type different from its declared type.

`NAME redeclared with type "TYPE".`

The given variable appears in a new declaration with a different type than it previously had.

`NAME is redefined as function.`

A name that was previously given to an external variable has now been given to a function.

LINT assumes that **`short`** integers have a length of 16 bits and that **`long`** integers have a length of 32 bits, regardless of what length these data types have on your machine. The reason is that short and long integers are guaranteed to have at least these lengths on all machines. Your hardware may allow longer integers (for example, 36 bits) but code that uses the increased size is not portable to machines with smaller words. For the same reason, LINT assumes that all characters are 8-bit signed quantities, even though they may be longer or unsigned on your machine.

Chapter 13
Extra Compiler Warnings

In addition to the diagnostics produced especially for LINT, the LINT output also contains all the error and warning messages produced by the parsing phase of the C compiler. Most of these are self-explanatory, but there are a few that deserve further comment.

`Constant "CONSTANT" too large for "TYPE".`

> As noted earlier, LINT assumes short integers are 16 bits long and long integers are 32 bits long. Therefore you may get this message, even though a short or long integer type on your machine is actually long enough to hold a given constant.

`Character constant with no characters.`

> Some compilers do not accept a null character constant. Your compiler may accept such constants, but LINT still complains about the construct.

`Constant valued logical expression.`

> This refers to situations like

```
while (1) { ... }
```

Alignment Problems

In addition to the error messages listed above, you may receive the warning

```
Possible alignment problems with cast
```

When casting data of one type to data of another, alignment problems may occur. For example, there may be difficulties in casting a character pointer to an integer pointer if the character pointer is not aimed at a word boundary. At times, however, you may have taken pains to make sure that this kind of alignment problem will not occur. In particular, you may have created a function that returns a value or pointer that is suitably aligned for any use. This is true of functions like `malloc`; the value that `malloc` returns is officially a **`void *`** pointer, but it is suitably aligned to store any sort of data.

If you have a function that returns a suitably aligned pointer, you can tell LINT by placing

```
#pragma aligned
```

in front of the start of the function declaration. This tells LINT that the function itself deals with any alignment problems.

Chapter 14

The LINT Command Line

Syntax:

```
tlint [file] [option]*

(+|-)ControlComments (-)   (+|-)Declarations (-)
(+|-)Keep (-)              (+|-)StandardLibrary (+)
(+|-)Verbose (-)           (+|-)Wide (-)
Configuration=file         CrossReference=keyword
Define=name=value          Include=directory
Installation=file          Library=lib
libraryName=string         Output=file
Output+=file              StandardInclude=directory
StandardLibrary=name       Summary=file
Target=keyword             indeX=file
```

Examples:

```
tlint myfile
tlint x=files o=incls ln=proj def=PROJECT=1
```

Options:

`file`

is a source file containing C code. The other options on the command line determine whether LINT scans this file for problem spots, or uses the file to create a summary or an entry in a LINT library.

`indeX=file`

gives the name of a text file. Each line in this text file should contain the name of one C source file. LINT checks each of these source files individually. In addition to a source file name, a line in an index file may specify `ControlComments`, `Configuration=`, `Include=` and `Define=` options. These options only apply to the source file named on the same line. For example, if the line

```
myfile define=VERSION=1
```

appeared in an index file, LINT would use the given option when examining `myfile` but not for other files named in the index file. When processing an index line, LINT normally uses all the options appearing on the main command line, followed by the options specified on the index line. As a special case, however, `Include=` options on a line in an index file are used before any `Include=` options on the LINT command line. Only one `indeX=` option may appear on the command line.

`Configuration=file`

specifies a configuration file for LINT. For more information on configuration files, see below.

`+ControlComments`

indicates that control comments of the type

```
/*VARARGS*/
```

should be processed. The default is `-ControlComments`, in which case such comments are ignored. A `ControlComments` option on a line in an index file overrides any `ControlComments` option on the main LINT command line.

`CrossReference=keyword`

indicates that LINT output should include a cross-reference table that shows where symbols are defined, referenced, set, and used. (By default, cross-references are not provided.) Possible values for the `keyword` are:

`All` - cross-references all symbols;

`External` - only cross-references external symbols.

`+Declarations`

may be used when creating a LINT library or summary file. With this option, LINT treats every file scope declaration as if it were a definition for the variable, instead of a simple reference.

`Define=name=value`

has the same effect as

```
#define name value
```

in the C source code. The option indicates that `"name"` should be replaced with `"value"` (as text) wherever it appears in the source code being examined. If `"value"` contains blanks or tab characters, it should be enclosed in double or single quotes.

`Include=directory`

is the same as the `Include=` option for the C compiler. When LINT tries to find quoted include files, as in

```
#include "file"
```

it begins by searching the given directory for the file. If the file is not found there, LINT searches any directory named in `StandardInclude=` options, and finally searches the directory that contains the source file being examined. Any number of `Include=` options may be specified. Directories are searched in the order given on the command line.

`INSTallation=file`

tells LINT where to find the installation file. The installation file tells where various software components have been installed. For more information, see the section on *Installation Files* below.

If you do not specify an `INSTallation=` option on the command line, LINT checks for an environment variable named `LINT_INST` and uses its value as the name of the installation file. If this environment variable does not exist, LINT uses the default installation file.

`+Keep`

does not delete intermediate files left by the preprocessor and the LINT steps.

`Library=lib`

names a LINT library that should be used when checking source code. Any number of `Library=` options may appear on the command line. By default, LINT automatically includes LINT libraries of standard C functions (e.g. `printf`) so that such functions are always recognized.

`libraryName=string`

is used when creating a LINT library. See below for more details.

`Output=file`

is used when creating a LINT library or summary file (see below). The output overwrites the file's current contents.

`Output+=file`

is the same as `Output=file`, except that output is written to the end of the current contents of the specified file, rather than overwriting what the file already holds.

`StandardInclude=directory`

is the same as the `StandardInclude=` option for the C compiler. When LINT tries to find include files whose names are enclosed in angle brackets, as in

```
#include <file>
```

it begins by searching the given directory for the file. If the file is not found there, LINT searches directories named in `Include=` options, and finally searches the directory that contains the source file being examined. Any number of `StandardInclude=` options may be specified. Directories are searched in the order given on the command line.

`StandardLibrary=name`

asks LINT to include the standard library indicated by "name". The following names are recognized:

`C` - the standard C library. This is included automatically, unless `-StandardLibrary` is specified.

`-StandardLibrary`
does not include the standard `C` library routines.

`Summary=file`
states that the given file is a summary file that should be used as input to the LINT operation. See below for more on summary files.

`Target=keyword`
controls the kind of problem-checking you want to do. The following keywords are recognized.

`Host` - describes any problems that may arise if the program is run on the host machine (i.e. the machine where you are running LINT). This is the default.

`Extensions` - points out any extensions to C that might have been used, as well as any machine-dependencies in the code. It also describes the problems detected with `Target=Host`.

`Minimal` - points out any features that may cause problems if you port this program to a non-ANSI compiler. This will make note of all ANSI features used. It also describes the problems detected with `Target=Extensions`.

`+Verbose`
prints out the name of the file being examined. The default is `-Verbose`.

`+Wide`
prints output in a format that is 132 columns wide. The default is `-Wide`, which prints in an 80-column format.

Abbreviating Options:

The option keywords given above can be abbreviated by omitting any or all of the letters shown in lower case. For example, `CrossReference=All` may be abbreviated to

```
crossref=all
cref=all
cr=all
cr=a
```

and so on. When entering option keywords, you may type letters in upper, lower, or mixed case; the use of upper and lower case in this documentation is simply to show what letters are and aren't required.

Other Ways to Use LINT

In addition to checking C code for irregularities, LINT can:

(a) Extract a summary file of information that can be used in other LINT operations.

(b) Create a LINT library.

Summary files and LINT libraries are discussed below.

Summary Files

A summary file contains a "summary" of your source code: the names and types of all the external variables and functions defined or referenced in the code, plus any other information LINT may need when checking the code (such as the declared types of function parameters). Summary files also record any error messages that might be issued by the C compiler when parsing the source code. All of this information is stored in a special format that is not directly readable by humans.

To create a summary file from a normal source code file, use the `Output=file` option, as in

```
tlint src1.c output=summ
```

You can add more material to the same summary file using `Output+=file`, as in

```
tlint src2.c output+=summ
```

This appends new material to the existing contents of the summary file.

When LINT creates a Summary file, LINT only summarizes your source code and checks for syntax errors. It does not check for such problems as non-portable constructs or type mismatches. However, you can run a summary file through LINT again to do standard type-checking. For example,

```
tlint srcfile output=lf
tlint summ=lf
```

creates a summary file, then uses the information in the summary file to see if types match in symbol references and definitions.

The reason for creating summary files is the same as the reason for breaking up the source code of your program into several source files: it is easier to deal with source code in small pieces than in one big hunk. Some users may keep a summary file for every source file. If the code in one source file is changed, you can create a summary file from that source file, then run all the summary files of the program through LINT to see if the change has caused any problems. This is much faster than using LINT on all the raw source code.

LINT Libraries

A LINT library is similar to a summary file, in that it contains a summary of C source code. However, LINT libraries are intended to parallel the way that object libraries work.

To create a LINT library, specify both the `Output=` option and the `libraryName=` option on the LINT command line. For example,

```
tlint file output=mylib libname="abc"
```

creates a LINT library named `abc` in the file `mylib`. This name is used in LINT diagnostic messages related to symbols found in the LINT library. Using LINT in this way only generates minimal messages; full messages are printed when you actually use the LINT library.

To use a LINT library, specify the `Library=` option on the LINT command line, as in

```
tlint myfile library=mylib
```

When LINT finds that `myfile` contains a reference to an undefined symbol, LINT checks the information in the given LINT library to see if the symbol is defined there. This works just like compiling a module while referring to an object library. No error occurs if a symbol in the LINT library has the same name as a symbol in the source file; the source file symbol is the one that is used.

Functions should only be placed in a LINT library when you are sure they contain no errors. LINT does not look for errors when it is creating a LINT library.

Review of Inputs and Outputs

To specify a file as an input, simply give the file's name. To specify a summary file as input, use `Summary=file`. To specify a LINT library as input, use `Library=file`. LINT uses all of the contents of a source file or a summary file; it only uses library entries if they are referenced by source code or a summary file.

An index file contains partial command lines to be used by LINT in its operations. These command lines can name source files as input, but not summary files or LINT libraries.

When both an `Output=` and `libraryName=` option are specified, LINT creates a LINT library. If only `Output=` is specified, LINT creates a summary file. If `Output=` is not specified, LINT examines a source file or summary file for problem spots and writes its diagnostics to the standard output.

Multiple Definitions

If LINT finds two definitions for the same function or external variable, it always outputs a diagnostic message. If the two definitions are both in normal source code, LINT arbitrarily chooses the first definition it finds as the "correct" definition, and issues diagnostics for any later deviations from this definition. If one of the two definitions is in a LINT library and the other is in normal source code, LINT chooses the non-library version as the "correct" definition. In this way, a definition in normal source code overrides a library definition.

Configuration Files

A configuration file consists of a series of directives that control the behavior of LINT. The possible directives are explained below.

`define name string`

> has the same format and purpose as a **`#define`** directive in normal C code. It creates a manifest or macro with the given value.

`inline includefile`

> lets you simulate an include file. When LINT finds an `inline` directive, it begins to gather input lines up until the first line consisting of only a `'#'` character. For example, in

```
inline sim.h
extern int junk1;
extern int junk2;
#
```

> LINT collects the two declarations and associates them with the name `"sim.h"`. If a program contains the directive,

```
#include <sim.h>
```

> LINT takes the gathered text and includes it at that point in the program, as if it had come from an included file.

`map includename1 includename2`

> says that all references of the form

```
#include includename1
```

should be converted to

```
#include includename2
```

`null includefile`

tells LINT to ignore all `#include` directives that attempt to include the specified file.

`search pathname`

is equivalent to `Include=pathname` on the LINT command line.

`system_search pathname`

is equivalent to `StandardInclude=pathname` on the LINT command line.

Installation Files:

An installation file specifies the pathnames for software and data files used by LINT. Installation files are text files made up of comment lines and option lines.

Comment lines:

Any line whose first non-blank character is `#` will be taken as a comment. Blank lines are also considered comments.

Option lines:

Option lines have the format

```
Keyword=pathname
```

In this documentation, keywords are written with some letters in upper case and some in lower case. You may abbreviate keywords by omitting any or all of the letters shown in lower case. The remaining letters may be entered in either upper or lower case; the documentation simply uses upper case to show which characters may not be omitted.

In this version of LINT, possible option lines are:

`Include=pathname`

gives the directory containing the LINT include files.

`Library=pathname`

gives the directory containing the LINT libraries.

`Program=pathname`

gives the directory containing the LINT executable files (CPP, and so on).