LCLint User's Guide

Version 2.4
April 1998

David Evans
Software Devices and Systems Group
MIT Laboratory for Computer Science

Larch

Acknowledgments

John Guttag and Jim Horning had the original idea for LCLint, have provided valuable advice on its functionality and design, and been instrumental in its development. This work has also benefited greatly from discussions with Mike Burrows, Stephen Garland, Colin Godfrey, Steve Harrison, Daniel Jackson, Angelika Leeb, Ulana Legedza, Anya Pogosyants, Navneet Singh, Raymie Stata, Yang Meng Tan, and Mark Vandevoorde. I especially thank Angelika Leeb for many constructive comments on improving this document, Raymie Stata for help designing and setting up the LCLint web site and Mark Vandevoorde for technical assistance.

Much of LCLint's development has been driven by feedback from users in academia and industry. Many more people than I can mention here have made contributions by suggesting improvements, reporting bugs, porting early versions of LCLint to other platforms. Particularly heroic contributions have been made by Eric Bloodworth, Jutta Degener, Rick Farnbach, Chris Flatters, Huver Hu, John Gerard Malecki, Thomas G. McWilliams, Michael Meskes, Richard O'Keefe, Jens Schweikhardt, and Albert L. Ting. Martin "Herbert" Dietze and Mike Smith performed valiantly in producing the original Win32 and OS2 ports.

LCLint incorporates the original LCL checker developed by Yang Meng Tan. This was built on the DECspec Project (Joe Wild, Gary Feldman, Steve Garland, and Bill McKeeman). The LSL checker used by LCLint was developed by Steve Garland. The original C grammar for LCLint was provided by Nate Osgood.

This research was supported by grants from ARPA (N0014-92-J-1795), NSF (9115797-CCR) and DEC ERP. David Evans was supported by an Intel Foundation Fellowship. LCLint was developed on DEC Alpha and DECmips machines provided by Digital Equipment Corporation and Pentium II™ machines donated by Intel. The Win32 version of LCLint was produced using Visual Studio™ software donated by Microsoft. This document was produced using Pentium II™ Computers donated by Intel Corporation and Microsoft Office™ software donated by Microsoft.

Contents

1. Overview 2

2. Operation 3

2.1 Messages 3

2.2 Flags 4

2.3 Stylized Comments 4

2.3.1 Annotations 5

2.3.2 Control Comments 5

3. Abstract Types 6

3.1 Access 7

3.2 Mutability 8

3.3 Boolean Types 8

3.4 Primitive C Types 9

3.4.1 Characters 9

3.4.2 Enumerators 10

3.4.3 Numeric Types 10

3.4.4 Arbitrary Integral Types 10

4. Function Interfaces 12

4.1 Modifications 12

4.1.1 Special Modifications 12

4.1.2 Missing Modifies Clauses 13

4.1.3 Limitations 13

4.2 Global Variables 14

4.2.1 Controlling Globals Checking 14

4.3 Declaration Consistency 15

5. Memory Management 16

5.1 Storage Model 16

5.2 Deallocation Errors 17

5.2.1 Unshared References 17

5.2.2 Temporary Parameters 18

5.2.3 Owned and Dependent References 18

5.2.4 Kept Parameters 19

5.2.5 Shared References 19

5.2.6 Stack References 19

5.2.7 Inner Storage 19

5.3 Implicit Memory Annotations 20

5.4 Reference Counting 21

6. Sharing 23

6.1 Aliasing 23

6.1.1 Unique Parameters 23

6.1.2 Returned Parameters 23
6.2 Exposure 24
6.2.1 Read-Only Storage 24
6.2.2 Exposed Storage 25

7. Value Constraints 26

7.1 Use Before Definition 26
7.1.1 Undefined Parameters 27
7.1.2 Relaxing Checking 27
7.1.3 Partially Defined Structures 28
7.1.4 Global Variables 27
7.2 Null Pointers 28
7.2.1 Predicate Functions 29
7.2.2 Overriding Null Types 30
7.2.3 Relaxing Null Checking 30
7.3 Execution 31
7.4 Special Clauses 32

8. Macros 35

8.1 Constant Macros 35
8.2 Function-like Macros 35
8.2.1 Side-Effect Free Parameters 36
8.2.2 Polymorphism 37
8.3 Controlling Macro Checking 37
8.4 Iterators 38

9. Naming Conventions 40

9.1 Type-Based Naming Conventions 40
9.1.1 Czech Names 40
9.1.2 Slovak Names 41
9.1.3 Czechoslovak Names 41
9.2 Namespace Prefixes 41
9.3 Naming Restrictions 43
9.3.1 Reserved Names 43
9.3.2 Distinct Identifiers 43

10. Other Checks 45

10.1 Undefined Evaluation Order 45
10.2 Problematic Control Structures 46
10.2.1 Likely Infinite Loops 46
10.2.2 Switches 46
10.2.3 Deep Breaks 47
10.2.4 Loop and If Bodies 48
10.2.5 Complete if-else Logic 48
10.3 Suspicious Statements 48
10.3.1 Statements with No Effects 48
10.3.2 Ignored Return Values 49
10.4 Unused Declarations 49
10.5 Complete Programs 50

10.5.1 Unnecessary External Names....50
10.5.2 Declarations Missing from Headers....50
10.6 Compiler Limits....51

Appendix A Availability....52

Appendix B Communication....52

Appendix C Flags....53

Global Flags....53
Help....53
Initialization....54
Pre-processor....54
Libraries....54
Output....55
Expected Errors....55
Message Format....56
Mode Selector Flags....56
Checking Flags....57
Key....57
Types....57
Function Interfaces....60
Memory Management....63
Sharing....65
Use Before Definition (Section 7.1)....66
Null Pointers (Section 7.2)....66
Macros (Section 8)....67
Iterators....68
Naming Conventions....68
Other Checks....72
Flag Name Abbreviations....78

Appendix D Annotations....79

Functions....79
Iterators (Section 8.4)....80
Constants (Section 8.1)....80
Alternate Types (Section 8.2.2)....80
Declarator Annotations....80

Appendix E Control Comments....85

Error Suppression....85
Type Access....85
Macro Expansion....85
Traditional Lint Comments....86

Appendix F Libraries....87

Standard Libraries....87
User Libraries....89
Header File Inclusion....89

Appendix G Specifications....90

Specification Flags....92

Appendix H Emacs 94
Running LCLint .....**¡Error!Marcador no definido.**
Editing Abbreviations 94

References.....94

Figures and Tables

Figure 1. Effort/benefit curve for LCLint..... 2
Figure 2. Violations using abstract types. 7
Figure 3. Boolean checking..... 9
Figure 4. Modifies checking..... 13
Figure 5. Globals checking..... 14
Figure 6. Deallocation errors..... 18
Figure 7. Stack references. 20
Figure 8. Implicit annotations. 21
Figure 9. Reference counting. 22
Figure 10. Unique parameters. 23
Figure 11. Returned parameters. 24
Figure 12. Exposure checking..... 26
Figure 13. Use before definition..... 27
Figure 14. Annotated globals lists..... 28
Figure 15. Null checking..... 29
Figure 16. Using notnull..... 31
Figure 17. Special Clauses. 34
Figure 18. Naming checks..... 44
Figure 19. Evaluation order..... 45
Figure 20. Infinite loop checking 47
Figure 21. Switch checking..... 47
Figure 22. Statements with no effect..... 49
Figure 23. Ignored return values. 50

Table 1. Prefix character codes. 42
Table 2. Flag name abbreviations..... 78

LCLint User's Guide

LCLint is a tool for statically checking C programs. With minimal effort, LCLint can be used as a better lint.[1] If additional effort is invested adding annotations to programs, LCLint can perform stronger checks than can be done by any standard lint.

Some problems detected by LCLint include:

- Violations of information hiding. A user-defined type can be declared as *abstract*, and a message is reported where code inappropriately depends on the representation of the type. (Section 3)
- Inconsistent modification of caller-visible state. Functions can be annotated with information on what caller-visible state may be modified by the function, and an error is reported if the modifications produced by the function contradict its declaration. (Section 4.1)
- Inconsistent use of global variables. Information on what global and file scope variables a function may use can be added to function declarations, and a message is reported if the implementation of the function uses other global variables or does not use every global variable listed in its declaration. (Section 4.2)
- Memory management errors. Instances where storage that has been deallocated is used, or where storage is not deallocated (memory leaks). (Section 5)
- Dangerous data sharing or unexpected aliasing. Parameters to a function share storage in a way that may lead to undefined or undesired behavior, or a reference to storage within the representation of an abstract type is created. (Section 6)
- Using possibly undefined storage or returning storage that is not completely defined (except as documented). (Section 7.1)
- Dereferencing a possibly null pointer. (Section 7.2)
- Dangerous macro implementations or invocations. (Section 8)
- Violations of customized naming conventions. (Section 9)
- Program behavior that is undefined because it depends on order of evaluation, likely infinite loops, fall-through cases, incomplete logic, statements with no effect, ignored return values, unused declarations, and exceeding certain standard limits. (Section 10)

LCLint checking can be customized to select what classes of errors are reported using command line flags and stylized comments in the code.

This document is a guide to using LCLint. Section 1 is a brief overview of the design goals of LCLint. Section 2 explains how to run LCLint, interpret messages and control checking. Sections 3–10 describe particular checks done by LCLint.

---

[1] Lint is a common programming tool for detecting anomalies in C programs. S. C. Johnson developed the original lint in the late seventies, mainly because early versions of C did not support function prototypes.

1. Overview

The main goals for LCLint are to:

- Detect a large number of bugs in typical C programs, without producing an unacceptable number of spurious messages. We are willing to accept the possibility that a few spurious messages are produced as long as it enables significantly more powerful checking and the spurious messages can be suppressed easily.
- Support a programming methodology involving abstract types and clean, documented interfaces in standard C programs.
- Provide a gradual transition for programmers. LCLint can be used like a better standard lint with minimal effort. Adding a few annotations to programs enables significantly better checking. As more effort is put into annotating programs, better checking results. A representational effort/benefit curve for using LCLint is shown in Figure 1. As different checks are turned on and more information is given in code annotations the number of bugs that can be detected increases dramatically.
- Provide enough flexibility so that LCLint can be used effectively with a wide range of coding styles. Especially important is making it easy to use LCLint effectively to maintain and modify legacy code.
- Check programs quickly and with no user interaction. LCLint runs faster than most compilers. Libraries can be used to enable fast checking of a few modules in a large program.

LCLint does many of the traditional lint checks including unused declarations, type inconsistencies, use-before-definition, unreachable code, ignored return values, execution paths with no return, likely infinite loops, and fall-through cases. This document focuses on more powerful checks that

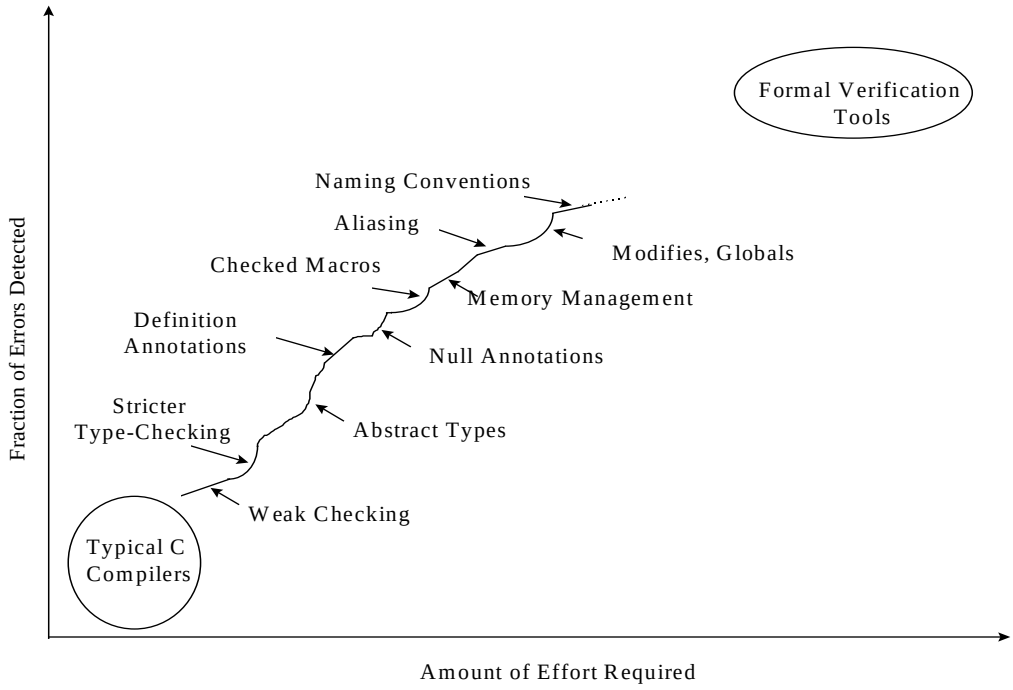

Figure 1. Effort/benefit curve for LCLint.

are made possible by additional information given in source code annotations.[2] Annotations are stylized comments that document certain assumptions about functions, variables, parameters and types. They may be used to indicate where the representation of a user-defined type is hidden, to limit where a global variable may be used or modified, to constrain what a function implementation may do to its parameters, and to express checked assumptions about variables, types, structure fields, function parameters, and function results. In addition to the checks specifically enabled by annotations, many of the traditional lint checks are improved by exploiting this additional information.

2. Operation

LCLint is invoked by listing files to be checked. Initialization files, command line flags, and stylized comments may be used to customize checking globally and locally.

The best way to learn to use LCLint, of course, is to actually use it (if you don't already have LCLint installed on your system, see page 52). Before you read much further in this document, I recommend finding a small C program. Then, try running:

```
lclint *.c
```

For the most C programs, this will produce a large number of messages. To turn off reporting for some of the messages, try:

```
lclint -weak *.c
```

The `-weak` flag is a mode flag that sets many checking parameters to select weaker checking than is done in the default mode. Other LCLint flags will be introduced in the following sections; a complete list is given in Appendix C.

2.1 Messages

The user can customize the format and content of messages printed by LCLint. A typical message is:

```
sample.c: (in function faucet)
sample.c:11:12: Fresh storage x not released before return
  A memory leak has been detected. Newly-allocated or only-qualified storage is not
  released before the last reference to it is lost. (-mustfree will suppress message)
   sample.c:5:47: Fresh storage x allocated
```

The first line gives the name of the function in which the error is found. This is printed before the first message reported for a function. (The function context is not printed if `-showfunc` is used.)

The second line is the text of the message. This message reports a memory leak — storage allocated in a function is not deallocated before the function returns. The file name, line and column number where the error is located precedes the text. The column numbers are used by emacs compile mode to jump to the appropriate line and column location. (Column numbers are not printed if `-showcol` is used.)

The next line is a hint giving more information about the suspected error. Most hints also include information on how the message may be suppressed. For this message, setting the `-mustfree` flag would prevent the message from being reported. Hints may be turned off by using `-hints`. Normally, a hint is given only the first time a class of error is reported. To have LCLint print a hint for every message regardless, use `+forcehints`.

---

[2] Another way to provide extra information about code is to use formal specifications (Appendix G).

The final line of the message gives additional location information. For this message, it tells where the leaking storage is allocated.

The generic message format is (parts enclosed in square brackets are optional):

```
[<file>:<line> (in <context>)]
<file>:<line>[,<column>]: message
   [hint]
    <file>:<line>,<column>: extra location information, if appropriate
```

The text of messages and hints may be longer than one line. They are split into lines of length less than the value set using `-linelen` *<number>*. The default line length is 80 characters. LCLint attempts to split lines in a sensible place as near to the line length limit as possible.

The `+parenfileformat` flag can be used to generate file locations in the format recognized by Microsoft Developer Studio. If `+parenfileformat` is set, the line number follows the file name in parentheses (e.g., `sample.c(11)`.)

2.2 Flags

So that many programming styles can be supported, LCLint provides over 300 flags for controlling checking and message reporting. Some of the flags are introduced in the body of this document. Appendix C describes every flag. Modes and shortcut flags are provided for setting many flags at once. Individual flags can override the mode settings.

Flags are preceded by `+` or `-`. When a flag is preceded by `+` we say it is *on*; when it is preceded by `-` it is *off*. The precise meaning of on and off depends on the type of flag.

The `+`/`-` flag settings are used for consistency and clarity, but contradict standard UNIX usage and it is easy to accidentally use the wrong one. To reduce the likelihood of using the wrong flag, LCLint issues warnings when a flag is set in an unusual way. Warnings are issued when a flag is redundantly set to the value it already had (these errors are not reported if the flag is set using a stylized comment), if a mode flag or special flag is set after a more specific flag that will be set by the general flag was already set, if value flags are given unreasonable values, of if flags are set in an inconsistent way. The `-warnflags` flag suppresses these warnings.

Default flag settings will be read from `~/.lclintrc` if it is readable. If there is a `.lclintrc` file in the working directory, settings in this file will be read next and its settings will override those in `~/.lclintrc`. Command-line flags override settings in either file. The syntax of the `.lclintrc` file is the same as that of command-line flags, except that flags may be on separate lines and the # character may be used to indicate that the remainder of the line is a comment. The `-nof` flag prevents the `~/.lclintrc` file from being loaded. The `-f` *<filename>* flag loads options from *filename*.

To make flag names more readable, hyphens (`-`), underscores (_) and spaces in flags at the command line are ignored. Hence, `warnflags`, `warn-flags` and `warn_flags` all select the `warnflags` option.

2.3 Stylized Comments

Stylized comments are used to provide extra information about a type, variable or function interface to improve checking, or to control flag settings locally.

All stylized comments begin with `/*@` and are closed by the end of the comment. The role of the `@` may be played by any printable character. Use `-commentchar` *<char>* to select a different stylized comment marker.

2.3.1 Annotations

Annotations are stylized comments that follow a definite syntax. Although they are comments, they may only be used in fixed grammatical contexts (e.g., like a type qualifier).

Syntactic comments for function interfaces are described in Section 4; comments for declaring constants in Section 8.1 and comments for declaring iterators in Section 8.4. Sections 3–7 include descriptions of annotations for expressing assumptions about variables, parameters, return values, structure fields and type definitions. A summary of annotations is found in Appendix D.

2.3.2 Control Comments

Unlike annotations, control comments may appear between any two tokens in a C program.[3] Syntactically, they are no different from standard comments. Control comments are used to provide source-level control of LCLint checking. They may be used to suppress spurious messages, set flags, and control checking locally in other ways. A complete description of control comments is found in Appendix E.

Most flags (all except those characterized as "global" in Appendix C) can be set locally using control comments. A control comment can set flags locally to override the command line settings. The original flag settings are restored before processing the next file. The syntax for setting flags in control comments is the same as that of the command line, except that flags may also be preceded by = to restore their setting to the original command-line value. For instance,

```
/*@+boolint -modifies =showfunc@*/
```

sets `boolint` on (this makes `bool` and `int` indistinguishable types), sets `modifies` off (this prevents reporting of modification errors), and sets `showfunc` to its original setting (this controls whether or not the name of a function is displayed before a message).

[3] Unlike regular C comments, control comments should not be used within a single token. They may introduce new separators in the code during parsing.

3. Abstract Types

Traditionally, programming books wax mathematical when they arrive at the topic of abstract data types… Such books make it seem as if you'd never actually use an abstract data type except as a sleep aid.

Steve McConnell

Information hiding is a technique for handling complexity. By hiding implementation details, programs can be understood and developed in distinct modules and the effects of a change can be localized. One technique for information hiding is data abstraction. An abstract type is used to represent some natural program abstraction. It provides functions for manipulating instances of the type. The module that implements these functions is called the *implementation* module. We call the functions that are part of the implementation of an abstract type the *operations* of the type. Other modules that use the abstract type are called *clients*.

Clients may use the type name and operations, but should not manipulate or rely on the actual representation of the type. Only the implementation module may manipulate the representation of an abstract type. This hides information, since implementers and maintainers of client modules should not need to know anything about how the abstract type is implemented. It provides modularity, since the representation of an abstract type can be changed without having to change any client code.

LCLint supports abstract types by detecting places where client code depends on the concrete representation of an abstract type.

To declare an abstract type, the `abstract` annotation is added to a `typedef`. For example (in `mstring.h`),

```
typedef /*@abstract@*/ char *mstring;
```

declares `mstring` as an abstract type. It is implemented using a `char *`, but clients of the type should not depend on or need to be aware of this. If it later becomes apparent that a better representation such as a string table should be used, we should be able to change the implementation of `mstring` without having to change or inspect any client code.

In a client module, abstract types are checked by name, not structure. LCLint reports an error if an instance of `mstring` is passed as a `char *` (for instance, as an argument to `strlen`), since the correctness of this call depends on the representation of the abstract type. LCLint also reports errors if any C operator except assignment (`=`) or `sizeof` is used on an abstract type. The assignment operator is allowed since its semantics do not depend on the representation of the type.[4] The use of `sizeof` is also permitted, since this is the only way for clients to allocate pointers to the abstract type. Type casting objects to or from abstract types in a client module is an abstraction violation and will generate a warning message.

Normally, LCLint will assume a type definition is not abstract unless the `/*@abstract@*/` qualifier is used. If instead you want all user-defined types to be abstract types unless they are marked as `concrete`, the `+imp-abstract` flag can be used. This adds an implicit `abstract` annotation to any `typedef` that is not marked with `/*@concrete@*/`.

Some examples of abstraction violations detected by LCLint are shown in Figure 2.

[4] For abstract types whose instances can change value, a client does need to know if assignment has copy or sharing semantics (see Section 3.2).

palindrome.c

```
# include "bool.h"
# include "mstring.h"

bool isPalindrome (mstring s)
{
6 char *current = (char *) s;
7 int i, len = (int) strlen (s);

  for (i = 0; i <= (len+1) / 2; i++)
    {
11    if (current[i] != s[len-i-1])
        return FALSE;
    }
  return TRUE;
}

bool callPal (void)
{
19 return (isPalindrome ("bob"));
}
```

Running LCLint

```
> lclint palindrome.c
LCLint 2.4 --- 10 Apr 98

palindrome.c: (in function isPalindrome)
palindrome.c:6: Cast from underlying abstract type
                mstring: (char *)s
palindrome.c:7: Function strlen expects arg 1 to be
                char * gets mstring: s
palindrome.c:11: Array fetch from non-array (mstring):
                 s[len - i - 1]
palindrome.c: (in function callPal)
palindrome.c:19: Function isPalindrome expects arg 1 to
                 be mstring gets char *: "bob"

Finished LCLint checking --- 4 code errors found

In client code, the abstract type is a distinct type,
incompatible with its concrete representation.
```

Figure[5] 2. Violations using abstract types.

3.1 Access

Where code may manipulate the representation of an abstract type, we say the code has *access* to that type. If code has access to an abstract type, the representation of the type and the abstract type are indistinguishable. Usually, a single program module that is the only code that has access to the type representation implements an abstract type. Sometimes, more complicated access control is desired if the implementation of an abstract type is split across program files, or particular client code needs to access the representation.

There are a several ways of selecting what code has access the representation of an abstract type:

- Modules. An abstract type defined in *M*.h is accessible in *M*.c. Controlled by the `accessmodule` flag. This means when `accessmodule` is on, as it is by default, the module access rule is in effect. If `accessmodule` is off (when `-access-module` is used), the module access rule is not in effect and an abstract type defined in *M*.h is not necessarily accessible in *M*.c
- File names. An abstract type named *type* is accessible in files named *type*.*<extension>*. For example, the representation of `mstring` is accessible in `mstring.h` and `mstring.c`. Controlled by the `access-file` flag.
- Function names. An abstract type named *type* may be accessible in a function named *type_name* or *typeName*. For example, `mstring_length` and `mstringLength` would have access to the `mstring` abstract type. Controlled by `accessfunction` and the naming convention (see Section 9).
- Access control comments. The syntax `/*@access` *type*`,`+`@*/`[6] allows the following code to access the representation of *type*. Similarly, `/*@noaccess` *type*`,`+`@*/` restricts access

[5] Output from LCLint is displayed in sans-serif font. The command line is preceded by >, the rest is output from LCLint. Explanations added to the code or LCLint output are shown in *italics*. Code shown in the figures in this document is available via anonymous `ftp` from `ftp://larch.lcs.mit.edu/pub/Larch/lclint/guide.tar.gz`

to the representation of *type*. The type in a `noaccess` comment must have been declared as an abstract type.

3.2 Mutability

We can view types as being *mutable* or *immutable*. A type is mutable if passing it as a parameter to a function call can change the value of an instance of the type.[7] For example, the primitive type `int` is immutable. If `i` is a local variable of type `int` and no variables point to the location where `i` is stored, the value of `i` must be the same before and after the call `f(i)`. Structure and union types are also immutable, since they are copied when they are passed as arguments. On the other hand, pointer types are mutable. If `x` is a local variable of type `int *`, the value of `*x` (and hence, the value of the object `x`) can be changed by the function call `g(x)`.

The mutability of a concrete type is determined by its type definition. For abstract types, mutability does not depend on the type representation but on what operations the type provides. If an abstract type has operations that may change the value of instances of the type, the type is mutable. If not, it is immutable. The value of an instance of an immutable type never changes. Since object sharing is noticeable only for mutable types, they are checked differently from immutable types.
The `/*@mutable@*/` and `/*@immutable@*/` annotations are used to declare an abstract type as mutable or immutable. (If neither is used, the abstract type is assumed to be mutable.) For example,

```
typedef /*@abstract@*/ /*@mutable@*/ char *mstring;
typedef /*@abstract@*/ /*@immutable@*/ int weekDay;
```

declares `mstring` as a mutable abstract type and `weekDay` as an immutable abstract type.

Clients of a mutable abstract type need to know the semantics of assignment. After the assignment expression `s = t`, do `s` and `t` refer to the same object (that is, will changes to the value of `s` also change the value of `t`)?

LCLint prescribes that all abstract types have sharing semantics, so `s` and `t` would indeed be the same object. LCLint will report an error if a mutable type is implemented with a representation (e.g., a `struct`) that does not provide sharing semantics (controlled by `mutrep` flag).

The mutability of an abstract type is not necessarily the same as the mutability of its representation. We could use the immutable concrete type `int` to represent mutable strings using an index into a string table, or declare `mstring` as immutable as long as no operations are provided that modify the value of an `mstring`.

3.3 Boolean Types

Standard C has no boolean representation – the result of a comparison operator is an integer, and no type checking is done for test expressions. Many common errors can be detected by introducing a distinct boolean type and stronger type checking.

---

[6] The meta-notation, `item,`$^{+}$ is used to denote a comma separated list of items. For example,

```
/*@access mstring, intSet@*/
```

provides access to the representations of both `mstring` and `intSet`.)

[7] Through the parameter. Modifications using some other variable that has a pointer to the location of this parameter are not considered.

Use the `-booltype <name>` flag to select the type name is used to represent boolean values.[8] Relations, comparisons and certain standard library functions are declared to return booleans.

LCLint checks that the test expression in an `if`, `while`, or `for` statement or an operand to `&&`, `||` or `!` is a boolean. If the type of a test expression is not a boolean, LCLint will report an error depending on the type of the test expression and flag settings. If the test expression has pointer type, LCLint reports an error if `predboolptr` is on (this can be used to prevent messages for the idiom of testing if a pointer is not null without a comparison). If it is type `int`, an error is reported if `pred-bool-int` is on. For all other types, LCLint reports an error if `pred-bool-others` is on.

Since using = instead of == is such a common bug, reporting of test expressions that are assignments is controlled by the separate `pred-assign` flag. The message can be suppressed by adding extra parentheses around the test expression.

Appendix C describes other flags for controlling boolean checking.

bool.c

```
# include "bool.h"

int f (int i, char *s,
       bool b1, bool b2)
{
6  if (i = 3)
7    return b1;
8  if (!I || s)
9    return i;
10 if (s)
11   return 7;
12 if (b1 == b2)
13   return 3;
14 return 2;
}
```

Running LCLint

```
> lclint bool.c +predboolptr –booltype bool
LCLint 2.4 --- 10 Apr 98

bool.c:7: Return value type bool does not match declared type int: b1
bool.c:6: Test expression for if is assignment expression: i = 3
bool.c:6: Test expression for if not bool, type int: I = 3
bool.c:8: Operand of ! is non-boolean (int): !I
bool.c:8: Right operand of || is non-boolean (char *): !i || s
bool.c:10: Test expression for if not bool, type char *: s
    Not reported without +predboolptr.
bool.c:12: Use of == with bool variables (risks inconsistency because
           of multiple true values): b1 == b2

Finished LCLint checking --- 7 code errors found
```

Figure 3. Boolean checking.

3.4 Primitive C Types

LCLint supports stricter checking of primitive C types. The `char` and `enum` types can be checked as distinct types, and the different numeric types can be type-checked strictly.

Two types have *compatible type* if their types are the same.
ANSI C, 3.1.2.6.

3.4.1 Characters

The primitive `char` type can be type-checked as a distinct type. If `char` is used as a distinct type, common errors involving assigning `int`s to `char`s are detected.

The `+charint` flag can be used for checking legacy programs where `char` and `int` are used interchangeably. If `charint` is on, `char` types indistinguishable from `int`s. To keep `char` and `int` as distinct types, but allow chars to be used to index arrays, use `+charindex`.

[8] To change the names of TRUE and FALSE, use `-booltrue` and `-boolfalse`. The LCLint distribution includes an implementation of `bool`, in `lib/bool.h`. However, it isn't necessary to use this implementation to get the benefits of boolean checking.

3.4.2 Enumerators

Standard C treats user-declared `enum` types just like integers. An arbitrary integral value may be assigned to an `enum` type, whether or not it was listed as an enumerator member. LCLint checks each user-defined `enum` type as distinct type. An error is reported if a value that is not an enumerator member is assigned to the `enum` type, or if an `enum` type is used as an operand to an arithmetic operator.

Two types need not be identical to be compatible.
ANSI C, footnote to 3.1.2.6.

If the `enumint` flag is on, `enum` and `int` types may be used interchangeably. Like `charindex`, if the `enumindex` flag is on, `enum` types may be used to index arrays.

3.4.3 Numeric Types

LCLint reports where numeric types are used in dangerous or inconsistent ways. With the strictest checking, LCLint will report an error anytime numeric types do not match exactly. If the `relaxquals` flag is on, only those inconsistencies that may corrupt values are reported. For example, if an `int` is assigned to a variable of type `long` (or passed as a `long` formal parameter), LCLint will not report an error if `relax-quals` is on since a `long` must have at least enough bits to store an `int` without data loss. On the other hand, an error would be reported if the `long` were assigned to an `int`, since the `int` type may not have enough bits to store the `long` value.

Similarly, if a `signed` value is assigned to an `unsigned`, LCLint will report an error since an `unsigned` type cannot represent all `signed` values correctly. If the `ignore-signs` flag is on, checking is relaxed to ignore all sign qualifiers in type comparisons (this is not recommended, since it will suppress reporting of real bugs, but may be necessary for quickly checking certain legacy code).

3.4.4 Arbitrary Integral Types

Some types are declared to be integral types, but the concrete type may be implementation dependent. For example, the standard library declares the types `size_t`, `ptr_diff` and `wchar_t`, but does not constrain their types other than limiting them to integral types. Programs may rely on them being integral types (e.g., can use `+` operator on two `size_t` operands), but should not rely on a particular representation (e.g., `long unsigned`).

LCLint supports three different kinds of arbitrary integral types:

`/*@integraltype@*/`
: An arbitrary integral type. The actual type may be any one of `short`, `int`, `long`, `unsigned short`, `unsigned`, or `unsigned long`.

`/*@unsignedintegraltype@*/`
: An arbitrary unsigned integral type. The actual type may be any one of `unsigned short`, `unsigned`, or `unsigned long`.

`/*@signedintegraltype@*/`
: An arbitrary signed integral type. The actual type may be any one of `short`, `int`, or `long`.

LCLint reports an error if the code depends on the actual representation of a type declared as an arbitrary integral. The `match-any-integral` flag relaxes checking and allows an arbitrary integral type is allowed to match any integral type.

Other flags set the arbitrary integral types to a concrete type. These should only be used if portability to platforms that may use different representations is not important. The `long-integral` and `long-unsigned-integral` flags set the type corresponding to `/*@integraltype@*/` to be `unsigned long` and `long` respectively. The `long-`

`unsigned-unsigned-integral` flag sets the type corresponding to `/*@unsignedintegraltype@*/` to be `unsigned long`. The `long-signed-integral` flag sets the type corresponding to `/*@signedintegraltype@*/` to be `long`.

4. Function Interfaces

Functions communicate with their calling environment through an interface. The caller communicates the values of actual parameters and global variables to the function, and the function communicates to the caller through the return value, global variables and storage reachable from the actual parameters. By keeping interfaces narrow (i.e., restricting the amount of information visible across a function interface), we can understand and implement functions independently.

A function prototype documents the interface to a function. It serves as a contract between the function and its caller. In early versions of C, the function "prototype" was very limited. It described the type returned by the function but nothing about its parameters. The main improvement provided by ANSI C was the ability to add information on the number and types of parameter to a function. LCLint provides the means to express much more about a function interface: what global variable the function may use, what values visible to the caller it may modify, if a pointer parameter may be a null pointer or point to undefined storage, if storage pointed to by a parameter is deallocated before the function returns, if the function may create new aliases to a parameter, can the caller modify or deallocate the return value, etc.

The extra interface information places constraints on both how the function may be called and how it may be implemented. LCLint reports places where these constraints are not satisfied. Typically, these indicate bugs in the code or errors in the interface documentation.

This section describes syntactic comments that may be added to a function declaration to document what global variables the function implementation may use and what values visible to its caller it may modify. Sections 4-7 describe annotations may be added to parameters to constrain valid arguments to a function and how these arguments may be used after the call and to the return value to constrain results.

4.1 Modifications

The modifies clause lists what values visible to the caller may be modified by a function. Modifies clauses limit what values a function may modify, but they do not require that listed values are always modified. The declaration,

```
int f (int *p, int *q) /*@modifies *p@*/;
```

declares a function `f` that may modify the value pointed to by its first argument but may not modify the value of its second argument or any global state.

LCLint checks that a function does not modify any caller-visible value not encompassed by its modifies clause and does modify all values listed in its modifies clause on some possible execution of the function. Figure 4 shows an example of modifies checking done by LCLint.

4.1.1 Special Modifications

A few special names are provided for describing function modifications:

`internalState`
: The function modifies some internal state (that is, the value of a `static` variable). Even though a client cannot access the internal state directly, it is important to know that something may be modified by the function call both for clear documentation and for checking undefined order of evaluation (Section 10.1) and side-effect free parameters (Section 8.2.1).

`fileSystem`

The function modifies the file system. Any modification that may change the system state is considered a file system modification. All functions that modify an object of type pointer to `FILE` also modify the file system. In addition, functions that do not modify a `FILE` pointer but modify some state that is visible outside this process also modify the file system (e.g., `rename`). The flag `mod-file-system` controls reporting of undocumented file system modifications.

`nothing`

The function modifies nothing (i.e., it is side-effect free).

The syntactic comment, `/*@*/` in a function declaration or definition (after the parameter list, before the semi-colon or function body) denotes a function that modifies nothing and does not use any global variables (see Section 4.2).

modify.c

```
void setx (int *x, int *y)
  /*@modifies *x@*/
{
4  *y = *x;
}

void sety (int *x, int *y)
  /*@modifies *y@*/
{
   setx (y, x);
 }
```

No errors for sety *– the call to* setx *modifies the value pointed to by its first parameter (y) as documented by the modifies clause.*

Running LCLint

```
> lclint modify.c +checks
LCLint 2.4 --- 10 Apr 98

modify.c:4: Undocumented modification of *y: *y = *x
modify.c:5: Suspect object listed in modifies of setx
            not modified: *x
  modify.c:1: Declaration of setx

Finished LCLint checking --- 2 code errors found
```

The `+checks` *flag is a mode flag for selecting moderately strict checking. It turns on* `mustmod` *checking, so the second error concerning missing documented modifications is reported.*

Figure 4. Modifies checking.

4.1.2 Missing Modifies Clauses

LCLint is designed so programs with many functions that are declared without modifies clauses can be checked effectively. Unless `modnomods` is in on, no modification errors are reported checking a function declared with no modifies clause.

A function with no modifies clause is an *unconstrained* function since there are no documented constraints on what it may modify. When an unconstrained function is called, it is checked differently from a function declared with a modifies clause. To prevent spurious errors, no modification error is reported at the call site unless the `mod-uncon` flag is on. Flags control whether errors involving unconstrained functions are reported for other checks that depend on modifications (side-effect free macro parameters (Section 8.2.1), undefined evaluation order (Section 10.1), and likely infinite loops (Section 10.2.1).)

4.1.3 Limitations

Determining whether a function modifies a particular parameter or global is in general an undecidable[9] problem. To enable useful checking, certain simplifying assumptions are necessary. LCLint assumes an object is modified when it appears on the left hand side of an assignment or it is passed to a function as a parameter which may be modified by that function (according to the called

[9] This means that theoreticians can prove that no algorithm exists that solves the problem correctly for all possible programs.

function's modifies clause). Hence, LCLint will report spurious modification errors for assignments that do not change the value of an object or modifications that are always reversed before a procedure returns. The `/*@-mods@*/` and `/*@=mods@*/` control comments can be used around these modifications to suppress the message.

4.2 Global Variables

Another aspect of a function's interface, is the global variables it uses. A globals list in a function declaration lists external variables that may be used in the function body. LCLint checks that global variables used in a procedure match those listed in its globals list. A global is used in a function if it appears in the body directly, or it is in the globals list of a function called in the body. LCLint reports if a global that is used in a procedure is not listed in its globals list, and if a listed global is not used in the function implementation. Figure 5 shows an example function definition with a globals list and associated checking done by LCLint.

globals.c	Running LCLint

`int glob1, glob2;`
 	
`3 int f (void) /*@globals glob1;@*/`
`{`
`5 return glob2;`
`}`
	> lclint globals.c +checks
	LCLint 2.4 --- 10 Apr 98
	
	globals.c:5: Undocumented use of global glob2
	globals.c:3: Global glob1 listed but not used
	
	Finished LCLint checking --- 2 code errors found

Figure 5. Globals checking.

4.2.1 Controlling Globals Checking

Whether on not an error is reported for a use of a global variable in a given function depends on the scope of the variable (file `static` or external), the checking annotation used in the variable declaration or the implicit annotation if no checking annotation is used, whether or not the function is declared with a globals list, and flag settings.

A global or file static variable declaration may be preceded by an annotation to indicate how the variable should be checked. In order of decreasing checks, the annotations are:

`/*@checkedstrict@*/`
: Strictest checking. Undocumented uses and modifications of the variable are reported in all functions whether or not they have a globals list (unless `check-strict-globs` is off).

`/*@checked@*/`
: Undocumented use of the variable is reported in a function with a globals list, but not in a function declared with no globals (unless `glob-noglobs` is on).

`/*@checkmod@*/`
: Undocumented uses of the variable are not reported, but undocumented modifications are reported. (If `mod-globs-nomods` is on, errors are reported even in functions declared with no modifies clause or globals list.)

`/*@unchecked@*/`
: No messages are reported for undocumented use or modification of this global variable.

If a variable has none of these annotations, an implicit annotation is determined by the flag settings.

Different flags control the implicit annotation for variables declared with global scope and variables declared with file scope (i.e., using the `static` storage qualifier). To set the implicit annotation for global variables declared in *`context`* (`globs` for external variables or `statics` for file static variable) to be *`annotation`* (`checked`, `checkmod`, `checkedstrict`) use `imp`*`<annotation><context>`*. For example, `+imp-checked-strict-statics` makes the implicit checking on unqualified file static variables `checkedstrict`. (See Appendix C for a complete list of globals checking flags.)

4.3 Declaration Consistency

LCLint checks that function declarations and definitions are consistent. The general rule is that the *first* declaration of a function implies all later declarations and definitions. If a function is declared in a header file, the first declaration processed is its first declaration (if it is declared in more than one header file an error is reported if `redecl` is set). Otherwise, the first declaration in the file defining the function is its first declaration.

Later declarations may not include variables in the globals list that were not included in the first declaration. The exception to this is when the first declaration is in a header file and the later declaration or definition includes file static variables. Since these are not visible in the header file, they can not be included in the header file declaration. Similarly, the modifies clause of a later declaration may not include objects that are not modifiable in the first declaration. The later declaration may be more specific. For example, if the header declaration is:

```
extern void setName (employee e, char *s) /*@modifies e@*/;
```

the later declaration could be,

```
void setName (employee e, char *) /*@modifies e->name@*/;
```

If `employee` is an abstract type, the declaration in the header should not refer to a particular implementation (i.e., it shouldn't rely on there being a `name` field), but the implementation declaration can be more specific.

This rule also applies to file static variables. The header declaration for a function that modifies a file static variable should use `modifies internalState` since file static variables are not visible to clients. The implementation declaration should list the actual file static variables that may be modified.

5. Memory Management

About half the bugs in typical C programs can be attributed to memory management problems. Memory management bugs are notoriously difficult to detect through traditional techniques. Often, the symptom of the bug is far removed from its actual source. Memory management bugs often only appear sporadically and some bugs may only be apparent when compiler optimizations are turned on or the code is compiled on a different platform. Run-time tools offer some help, but are cumbersome to use and limited to detecting errors that occur when test cases are run. By detecting these errors statically, we can be confident that certain types of errors will never occur and provide verified documentation on the memory management behavior of a program.

LCLint can detect many memory management errors at compile time including:

- using storage that may have been `free`d (Section 5.2)
- failing to deallocate memory (Section 5.2)
- returning a pointer to stack-allocated storage (Section 5.2.6)
- undocumented or dangerous aliasing or storage sharing (Section 6)
- passing or returning storage that is not completely defined (Section 7.1)
- dereferencing a null pointer (Section 7.2)

Most of these checks rely heavily on annotations added to programs to document assumptions related to memory management and pointer values. By documenting these assumptions for function interfaces, variables, type definitions and structure fields, memory management bugs can be detected at their source — where an assumption is violated. In addition, precise documentation about memory management decisions makes it easier to change code.

Yea, from the table of my memory I'll wipe away all trivial fond records, all saws of books, all forms, all pressures past, that youth and observation copied there.
Hamlet prefers garbage collection (Shakespeare, Hamlet. Act I, Scene v)

5.1 Storage Model[10]

This section describes execution-time concepts for describing the state of storage more precisely than can be done using standard C terminology. Certain uses of storage are likely to indicate program bugs, and are reported as anomalies.

LCL assumes a CLU-like object storage model.[11] An *object* is a typed region of storage. Some objects use a fixed amount of storage that is allocated and deallocated automatically by the compiler.
Other objects use dynamic storage that must be managed by the program.

Storage is *undefined* if it has not been assigned a value, and *defined* after it has been assigned a value. An object is *completely defined* if all storage that may be reached from it is defined. What storage is reachable from an object depends on the type and value of the object. For example, if `p` is a pointer to a structure, `p` is completely defined if the value of `p` is `NULL`, or if every field of the structure `p` points to is completely defined.

When an expression is used as the left side of an assignment expression we say it is *used as an lvalue*. Its location in memory is used, but not its value. Undefined storage may be used as an lvalue since only its location is needed. When storage is used in any other way, such as on the right side of an assignment, as an operand to a primitive operator (including the indirection operator,

[10] This section is largely based on [Evans96]. It semi-formally defines some of the terms needed to describe memory management checking; if you are satisfied with an intuitive understanding of these terms, this section may be skipped.
[11] This is similar to the LISP storage model, except that objects are typed.

`*`),[12] or as a function parameter, we say it is *used as an rvalue*. It is an anomaly to use undefined storage as an rvalue.

A *pointer* is a typed memory address. A pointer is either *live* or *dead*. A live pointer is either `NULL` or an address within allocated storage. A pointer that points to an object is an *object* pointer. A pointer that points inside an object (e.g., to the third element of an allocated block) is an *offset* pointer. A pointer that points to allocated storage that is not defined is an *allocated* pointer. The result of dereferencing an allocated pointer is undefined storage. Hence, it is an anomaly to use it as an rvalue. A dead (or "dangling") pointer does not point to allocated storage. A pointer becomes dead if the storage it points to is deallocated (e.g., the pointer is passed to the `free` library function.) It is an anomaly to use a dead pointer as an rvalue.

There is a special object *null* corresponding to the `NULL` pointer in a C program. A pointer that may have the value `NULL` is a *possibly-null* pointer. It is an anomaly to use a possibly-null pointer where a non-null pointer is expected (e.g., certain function arguments or the indirection operator).

5.2 Deallocation Errors

There are two kinds of deallocation errors with which we are concerned: deallocating storage when there are other live references to the same storage, or failing to deallocate storage before the last reference to it is lost. To handle these deallocation errors, we introduce a concept of an obligation to release storage. Every time storage is allocated, it creates an obligation to release the storage. This obligation is attached to the reference to which the storage is assigned.[13] Before the scope of the reference is exited or it is assigned to a new value, the storage to which it points must be released. Annotations can be used to indicate that this obligation is transferred through a return value, function parameter or assignment to an external reference.

'Tis in my memory lock'd, and you yourself shall keep the key of it.
Ophelia prefers explicit deallocation (Hamlet. Act I, Scene iii)

5.2.1 Unshared References

The `only` annotation is used to indicate a reference is the only pointer to the object it points to. We can view the reference as having an obligation to release this storage. This obligation is satisfied by
transferring it to some other reference in one of three ways:

- pass it as an actual parameter corresponding to a formal parameter declared with an `only` annotation
- assign it to an external reference declared with an `only` annotation
- return it as a result declared with an `only` annotation

After the release obligation is transferred, the original reference is a dead pointer and the storage it points to may not be used.

All obligations to release storage stem from primitive allocation routines (e.g., `malloc`), and are ultimately satisfied by calls to `free`. The standard library declared the primitive allocation and deallocation routines.

The basic memory allocator, `malloc`, is declared:[14]

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

[12] Except `sizeof`, which does not need the value of its argument.
[13] If the storage is not assigned to a reference, an internal reference is created to track the storage.
[14] The full declaration of `malloc` also includes a `null` annotation (Section 7.2) to indicate that the result may be `NULL` (as it is when the requested storage cannot be allocated) and an `out` annotation (Section 7.1) to indicate that the result points to undefined storage.

It returns an object that is referenced only by the function return value.

The deallocator, `free`, is declared:[15]

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

The parameter to `free` must reference an unshared object. Since the parameter is declared using `only`, the caller may not use the referenced object after the call, and may not pass in a reference to a shared object. There is nothing special about `malloc` and `free` — their behavior can be described entirely in terms of the provided annotations.

only.c

```
1 extern /*@only@*/ int *glob;

/*@only@*/ int *
f (/*@only@*/ int *x, int *y,
   int *z)
  /*@globals glob;@*/
{
8 int *m = (int *)
9    malloc (sizeof (int));

11 glob = y;      Memory leak
12 free (x);
13 *m = *x;       Use after free
14 return z;      Memory leak detected
}
```

Running LCLint

```
> lclint only.c
LCLint 2.4 --- 10 Apr 98

only.c:11: Only storage glob not released before
             assignment: glob = y
  only.c:1: Storage glob becomes only
only.c:11: Implicitly temp storage y assigned to only:
             glob = y
only.c:13: Dereference of possibly null pointer m: *m
  only.c:8: Storage m may become null
only.c:13: Variable x used after being released
  only.c:12: Storage x released
only.c:14: Implicitly temp storage z returned as only: z
only.c:14: Fresh storage m not released before return
  only.c:9: Fresh storage m allocated

Finished LCLint checking --- 6 code errors found
```

Figure 6. Deallocation errors.

5.2.2 Temporary Parameters

The `temp` annotation is used to declare a function parameter that is used temporarily by the function. An error is reported if the function releases the storage associated with a `temp` formal parameter or creates new aliases it that are visible after the function returns. Any storage may be passed as a `temp` parameter, and it satisfies its original memory constraints after the function returns.

5.2.3 Owned and Dependent References

In real programs it is sometimes necessary to have storage that is shared between several possibly references. The `owned` and `dependent` annotations provide a more flexible way of managing storage, at the cost of less checking. The `owned` annotation denotes a reference with an obligation to release storage. Unlike `only`, however, other external references marked with `dependent`

[15] The full declaration of `free` also has `out` and `null` annotations on the parameter to indicate that the argument may be `NULL` and need not point to defined storage. According to [ANSI, 4.10.3.2], `NULL` may be passed to `free` without an error. On some UNIX platforms, passing `NULL` to free causes a program crash so the UNIX version of the standard library (Appendix F) specifies `free` without the `null` annotation on its parameter. To check that allocated objects are completely destroyed (e.g., all unshared objects inside a structure are deallocated before the structure is deallocated), LCLint checks that any parameter passed as an `out only void *` does not contain references to live, unshared objects. This makes sense, since such a parameter could not be used sensibly in any way other than deallocating its storage.

annotations may share this object. It is up to the programmer to ensure that the lifetime of a `dependent` reference is contained within the lifetime of the corresponding `owned` reference.

5.2.4 Kept Parameters

The `keep` annotation is similar to `only`, except the caller may use the reference after the call. The called function must assign the `keep` parameter to an `only` reference, or pass it as a `keep` parameter to another function. It is up to the programmer to make sure that the calling function does not use this reference after it is released. The `keep` annotation is useful for adding an object to a collection (e.g., a symbol table), where it is known that it will not be deallocated until the collection is.

5.2.5 Shared References

If LCLint is used to check a program designed to be used in a garbage-collected environment, there may be storage that is shared by one or more references and never explicitly released. The `shared` annotation declares storage that may be shared arbitrarily, but never released.

5.2.6 Stack References

Local variables that are not allocated dynamically are stored on a call stack. When a function returns, its stack frame is deallocated, destroying the storage associated with the function's local variables. A memory error occurs if a pointer into this storage is live after the function returns. LCLint detects errors involving stack references exported from a function through return values or assignments to references reachable from global variables or actual parameters. No annotations are needed to detect stack reference errors, since it is clear from a declaration if storage is allocated on the function stack.

Figure 7 gives and example of errors reported involving stack-allocated storage.

5.2.7 Inner Storage

An annotation always applies to the outermost level of storage. For example,

```
/*@only@*/ int **x;
```

declares `x` as an unshared pointer to a pointer to an `int`. The `only` annotation applies to `x`, but not to `*x`. To apply annotations to inner storage a type definition may be used:

```
typedef /*@only@*/ int *oip;
/*@only@*/ oip *x;
```

Now, x is an `only` pointer to an `oip`, which is an `only` pointer to an `int`.

stack.c

```
int *glob;

/*@dependent@*/ int *
  f (int **x)
{
  int sa[2] = { 0, 1 };
  int loc = 3;

9 glob = &loc;
10 *x = &sa[0];

12 return &loc;
}
```

A `dependent` *annotation is used on the return value. Without this, several other errors would be reported, since the result would have an implicit* `only` *annotation.*

Running LCLint

```
> lclint stack.c
LCLint 2.4 --- 10 Apr 98

stack.c: (in function f)
stack.c:12: Stack-allocated storage &loc reachable
              from return value: &loc
stack.c:12: Stack-allocated storage *x reachable from
              parameter x
  stack.c:10: Storage *x becomes stack
stack.c:12: Stack-allocated storage glob reachable
              from global glob
  stack.c:9: Storage glob becomes stack

Finished LCLint checking --- 3 code errors found
```

Figure 7. Stack references.

When annotations are used in type definitions, they may be overridden in instance declarations. For example,

```
/*@dependent@*/ oip x;
```

makes `x` a `dependent` pointer to an `int`.

Another way to apply annotations to inner storage is to use a special clause (see Section 7.4).

5.3 Implicit Memory Annotations

Since it is important that LCLint can check unannotated programs effectively, the meaning of declarations with no memory annotations is chosen to minimize the number of annotations needed to get useful checking on an unannotated program.

An implicit memory management annotation may be assumed for declarations with no explicit memory management annotation. Implicit annotations are checked identically to the corresponding explicit annotation, except error messages indicate that they result from an implicit annotation.

Unannotated function parameters are assumed to be `temp`. This means if memory checking is turned on for an unannotated program, all functions that release storage referenced by a parameter or assign a global variable to alias the storage will produce error messages. (Controlled by `paramimptemp`.)

Unannotated return values, structure fields and global variables are assumed to be `only`. With implicit annotations (on by default), turning on memory checking for an unannotated program will produce errors for any function that does not return unshared storage or assignment of shared storage to a global variable or structure field.[16] (Controlled by `retimponly`, `structimponly` and `globimponly`. The `allimponly` flag sets all of the implicit only flags.)

[16] If an exposure qualifier is used (see Section 6.2), the implied `dependent` annotation is used instead of the more generally implied `only` annotation.

implicit.c

```
typedef struct {
   only char *name;
   int   val;
} *rec;

extern only rec rec_last ;

extern only rec
   rec_create (temp char *name,
               int val) ;
```

Annotations in `italics` *are not present in the code, but may be implied.*

Implicit `only` *annotation on mutable structure field if* `structimponly` *is on.*

Implicit `only` *annotation on mutable global variables if* `globimponly` *is on.*

Implicit `only` *annotation on mutable function result if* `retimponly` *is set. Implicit* `temp` *annotation on mutable parameter if* `paramimptemp` *is set.*

Figure 8. Implicit annotations.

5.4 Reference Counting

Another approach to memory management is to add a field to a type to explicitly keep track of the number of references to that storage. Every time a reference is added or lost the reference count is adjusted accordingly; if it would become zero, the storage is released. Reference counting it difficult to do without automatic checking since it is easy to forget to increment or decrement the reference count, and exceedingly difficult to track down these errors.

LCLint supports reference counting by using annotations to constrain the use of reference counted storage in a manner similar to other memory management annotations.
A reference counted type is declared using the `refcounted` annotation. Only pointer to `struct` types may be declared as `reference counted`, since reference counted storage must have a field to count the references. One field in the structure (or integral type) is preceded by the `refs` annotation to indicate that the value of this field is the number of live references to the structure.

For example (in `rstring.h`),

```
typedef /*@abstract@*/ /*@refcounted@*/ struct {
  /*@refs@*/ int refs;
  char *contents;
} *rstring;
```

declares `rstring` as an abstract, reference-counted type. The `refs` field counts the number of references and the `contents` field holds the contents of a string.

All functions that return `refcounted` storage must increase the reference count before returning. LCLint cannot determine if the reference count was increased, so any function that directly returns a reference to `refcounted` storage will produce an error. This is avoided, by using a function to return a new reference (e.g., `rstring_ref` in Figure 9).

A reference counted type may be passed as a `temp` or `dependent` parameter. It may not be passed as an `only` parameter. Instead, the `killref` annotation is used to denote a parameter whose reference is eliminated by the function call. Like `only` parameters, an actual parameter corresponding to a `killref` formal parameter may not be used in the calling function after the call. LCLint checks that the implementation of a function releases all `killref` parameters, either by passing them as `killref` parameters, or assigning or returning them without increasing the reference count.

rstring.c

```
# include "rstring.h"

static rstring rstring_ref (rstring r)
{
  r->refs++;
6 return r;
}

rstring rstring_first (rstring r1, rstring r2)
{
  if (strcmp (r1->contents, r2->contents) < 0)
    {
13    return r1;
    }
  else
    {
17    return rstring_ref (r2);
    }
}
```

Running LCLint

```
> lclint rstring.c
LCLint 2.4 --- 10 Apr 98

rstring.c: (in function rstring_first)
rstring.c:13: Reference counted storage
   returned without modifying reference
   count: r1

Finished LCLint checking --- 1 code
  error found
```

No error is reported for line 6 since the reference count was incremented. No error is reported for line 17, since `rstring_ref` *returns a new reference.*

Figure 9. Reference counting.

6. Sharing

Errors involving unexpected sharing of storage can cause serious problems. Undocumented sharing may lead to unpredictable modifications, and some library calls (e.g., `strcpy`) have undefined behavior if parameters share storage. Another class of sharing errors occurs when clients of an abstract type may obtain a reference to mutable storage that is part of the abstract representation. This exposes the representation of the abstract type, since clients may modify an instance of the abstract type indirectly through this shared storage.

6.1 Aliasing

LCLint detects errors involving dangerous aliasing of parameters. Some of these errors are already detected through the standard memory annotations (e.g., `only` parameters may not be aliases.) Two additional annotations are provided for constraining aliasing of parameters and return values.

6.1.1 Unique Parameters

The `unique` annotation denotes a parameter that may not be aliased by any other storage reachable from the function implementation — that is, any storage reachable through the other parameters or global variables used by the function. The `unique` annotation places similar constraints on function parameters as the `only` annotation, but it does not transfer the obligation to release storage.
LCLint will report an error if a `unique` parameter may be aliased by another parameter or global variable.

unique.c

```
# include <string.h>

void
capitalize (/*@out@*/ char *s,
            char *t)
{
7  strcpy (s, t);
   *s = toupper (*s);
}
```

The `out` *qualifier is explained in Section 7.1.1.*

Running LCLint

```
> lclint unique.c
LCLint 2.4 --- 10 Apr 98

unique.c: (in function capitalize)
unique.c:7: Parameter 1 (s) to function strcpy is
   declared unique but may be aliased externally by
   parameter 2 (t)

Finished LCLint checking --- 1 code error found
```

An error is reported since the first parameter to the library function `strcpy` *is declared with unique. If a* `unique` *qualifier were added to the parameter declaration for* `s` *or* `t`*, no error would be reported.*

Figure 10. Unique parameters.

6.1.2 Returned Parameters

LCLint reports an error if a function returns a reference to storage reachable from one of its parameters (if `retalias` is on) since this may introduce unexpected aliases in the body of the calling function when the result is assigned.

The `returned` annotation denotes a parameter that may be aliased by the return value. LCLint checks the call assuming the result may be an alias to the `returned` parameter. Figure 11 shows an example use of a `returned` annotation.

returned.c

```
# include "intSet.h"

extern intSet intSet_insert (/*@returned@*/ intSet s, int x);

intSet intSet_singleton (int x)
{
7  return (intSet_insert (intSet_new (), x));
}
```

Without the `returned` *qualifier, a memory leak error would be reported for line 7, since the* `only` *storage returned by* `intSet_new` *is not released. Because of the* `returned` *qualifier on the first parameter to* `intSet_insert`*, LCLint assumes the result of* `intSet_insert` *is the same storage as its first parameter, in this case the storage returned by* `intSet_new`*. No error is reported, since the* `only` *storage is then transferred through the return value (which has an implicit* `only` *annotation, see Section 5.3).*

Figure 11. Returned parameters.

6.2 Exposure

LCLint detects places where the representation of an abstract type is exposed. This occurs if a client has a pointer to storage that is part of the representation of an instance of the abstract type. The client can then modify or examine the storage this points to, and manipulate the value of the abstract type instance without using its operations.

There are three ways a representation may be exposed:

1. Returning (or assigning to a global variable) an object that includes a pointer to a mutable component of an abstract type representation. (Controlled by `ret-expose`).
2. Assigning a mutable component of an abstract object to storage reachable from an actual parameter or a global variable that may be used after the call. This means the client may manipulate the abstract object using the actual parameter after the call. Note that if the corresponding formal parameter is declared `only`, the caller may not use the actual parameter after the call so the representation is not exposed. (Controlled by `assign-expose`).
3. Casting mutable storage to or from an abstract type. (Controlled by `cast-expose`).

Annotations may be used to allow exposed storage to be returned safely by restricting how the caller may use the returned storage.

6.2.1 Read-Only Storage

It is often useful for a function to return a pointer to internal storage (or an instance of a mutable abstract type) that is intended only as an *observer*. The caller may use the result, but should not modify the storage it points to. For example, consider a naïve implementation of the `employee_getName` operation for the abstract `employee` type:

```
typedef /*@abstract@*/ struct {
   char *name;
   int id;
} *employee;
…
char *employee_getName (employee e) { return e->name; }
```

LCLint produces a message to indicate that the return value exposes the representation. One solution would be to return a fresh copy of `e->name`. This is expensive, though, especially if we expect `employee_getName` is used mainly just to get a string for searching or printing. Instead, we could change the declaration of `employee_getName` to:

```
extern /*@observer@*/ char *employee_getName (employee e);
```

Now, the original implementation is correct. The declaration indicates that the caller may not modify the result, so it is acceptable to return shared storage.[17] LCLint checks that the caller does not modify the return value. An error is reported if observer storage is modified directly, passed as a function parameter that may be modified, assigned to a global variable or reference derivable from a global variable that is not declared with an `observer` annotation, or returned as a function result or a reference derivable from the function result that is not annotation with an `observer` annotation.

String Literals

A program that attempts to modify a string literal has undefined behavior [ANSI, Section 3.1.4]. This is not enforced by most C compilers, and can lead to particularly pernicious bugs that only appear when optimizations are turned on and the compiler attempts to minimize storage for string literals. LCLint can be used to check that string literals are not modified, by treating them as `observer` storage. If `read-only-strings` is on (default in standard mode), LCLint will report an error if a string literal is modified.

6.2.2 Exposed Storage

Sometimes it is necessary to expose the representation of an abstract type. This may be evidence of a design flaw, but in some cases is justified for efficiency reasons. The `exposed` annotation denotes storage that is exposed. It may be used on a return value for results that reference storage internal to an abstract representation, on a parameter value to indicate a parameter that may be assigned directly to part of an abstract representation,[18] or on a field of an abstract representation to indicate that external references to the storage may exist. An error is reported if `exposed` storage is released, but unlike an `observer`, no error is reported if it is modified.

Figure 12 shows examples of exposure problems detected by LCLint.

[17] Strictly, we should also check that the returned observer storage is not used again after any other calls to the abstract type module using the same parameter. LCLint does not attempt to check this, and in practice it is not usually a problem.

[18] Note that if the parameter is annotated with `only`, it is not an error to assign it to part of an abstract representation, since the caller may not use the storage after the call returns.

exposure.c

```
# include "employee.h"

char *
employee_getName (employee e)
{
6  return e->name;
}

/*@observer@*/ char *
employee_obsName (employee e)
{ return e->name; }

/*@exposed@*/ char *
employee_exposeName (employee e)
{ return e->name; }

void
employee_capName (employee e)
{
  char *name;

  name = employee_obsName (e);
23 *name = toupper (*name);
}
```

Running LCLint

```
> lclint exposure.c +checks
LCLint 2.4 --- 10 Apr 98

exposure.c: (in function employee_getName)
exposure.c:6: Function returns reference to parameter
              e: e->name
exposure.c:6: Return value exposes rep of employee:
              e->name
exposure.c:6: Released storage e->name reachable from
              parameter
  exposure.c:6: Storage e->name is released
exposure.c: (in function employee_capName)
exposure.c:23: Suspect modification of observer name:
               *name = toupper(*name)

Finished LCLint checking --- 4 code errors found
```

Three messages are reported for line 6 where a mutable field of an abstract type is returned with no sharing qualifier (without `+checks` *only the third one would be reported.)*

The error for line 23 reports a modification of an observer. If the call in line 22 were changed to call `employee_exposeName`*, no error would be reported.*

Figure 12. Exposure checking.

7. Value Constraints

LCLint can be used to constrain values of parameters, function results, global variables, and derived storage such as structure fields. These constraints are checked at *interface points* — where a function is called or returns. Section 7.1 describes how to constrain parameters, return values and structures to detect use before definition errors. A similar approach is used for restricting the use of possibly null pointers in Section 7.2. To do both well, and avoid spurious errors, information about when and if a function returns if useful. Annotations for documenting execution control are described in Section 7.3.

7.1 Use Before Definition

Like many static checkers, LCLint detects instances where the value of a location is used before it is defined. This analysis is done at the procedural level. If there is a path through the procedure that uses a local variable before it is defined, a use before definition error is reported. The `usedef` flag controls use before definition checking.

LCLint can do more checking than standard checkers though, because the annotations can be used to describe what storage must be defined and what storage may be undefined at interface points. Unannotated references are expected to be completely defined at interface points. This means all storage reachable from a global variable, parameter to a function, or function return value is defined before and after a function call.

7.1.1 Undefined Parameters

Sometimes, function parameters or return values are expected to reference undefined or partially defined storage. For example, a pointer parameter may be intended only as an address to store a result, or a memory allocator may return allocated but undefined storage. The `out` annotation denotes a pointer to storage that may be undefined.

LCLint does not report an error when a pointer to allocated but undefined storage is passed as an `out` parameter. Within the body of a function, LCLint will assume an `out` parameter is allocated but not necessarily bound to a value, so an error is reported if its value is used before it is defined.

LCLint reports an error if storage reachable by the caller after the call is not defined when the function returns. This can be suppressed by `-must-define`. After a call returns, an actual parameter corresponding to an `out` parameter is assumed to be completely defined.

When checking unannotated programs, many spurious use before definition errors may be reported If `impouts` is on, no error is reported when an incompletely-defined parameter is passed to a formal parameter with no definition annotation, and the actual parameter is assumed to be defined after the call. The `/*@in@*/` annotation can be used to denote a parameter that must be completely defined, even if `imp-outs` is on. If `imp-outs` is off, there is an implicit `in` annotation on every parameter with no definition annotation.

usedef.c

```
extern void
  setVal (/*@out@*/ int *x);
extern int
  getVal (/*@in@*/ int *x);
extern int mysteryVal (int *x);

int
dumbfunc (/*@out@*/ int *x, int i)
{
  if (i > 3)
11   return *x;
  else if (i > 1)
13   return getVal (x);
  else if (i == 0)
15   return mysteryVal (x);
  else
    {
18    setVal (x);
19    return *x;
    }
}
```

Running LCLint

```
> lclint usedef.c
LCLint 2.4 --- 10 Apr 98

usedef.c: (in function dumbfunc)
usedef.c:11: Value *x used before definition
usedef.c:13: Passed storage x not completely defined
             (allocated only): getVal (x)
usedef.c:15: Passed storage x not completely defined
             (allocated only): mysteryVal (x)
```

Not reported if `impouts` *is on since there is no* `in` *annotation on the parameter to* `mysteryVal`.

```
Finished LCLint checking --- 3 code errors found
```

No error is reported for line 18, since the incompletely defined storage `x` *is passed as an* `out` *parameter. After the call,* `x` *may be dereferenced, since* `setVal` *is assumed to completely define its* `out` *parameter.*

Figure 13. Use before definition.

7.1.2 Relaxing Checking

The `reldef` annotation relaxes definition checking for a particular declaration. Storage declared with a `reldef` annotation is assumed to be defined when it is used, but no error is reported if it is not defined before it is returned or passed as a parameter.

It is up to the programmer to check `reldef` fields are used correctly. They should be avoided in most cases, but may be useful for fields of structures that may or may not be defined depending on other constraints.

7.1.3 Partially Defined Structures

The `partial` annotated can be used to relax checking of structure fields. A structure with undefined fields may be passed as a `partial` parameter or returned as a `partial` result. Inside a function body, no error is reported when the field of a `partial` structure is used. After a call, all fields of a structure that is passed as a `partial` parameter are assumed to be completely defined.

7.1.4 Global Variables

Special annotations can be used in the globals list of a function declaration (Section 4.2) to describe the states of global variables before and after the call.

If a global is preceded by `undef`, it is assumed to be undefined before the call. Thus, no error is reported if the global is not defined when the function is called, but an error is reported if the global is used in the function body before it is defined.

annotglobs.c

```
int globnum;

struct {
  char *firstname;
  char *lastname;
  int id;
} globname;

void
initialize (/*@only@*/ char *name)
  /*@globals undef globnum,
             undef globname @*/
{
14 globname.id = globnum;
  globname.lastname = name;
16}

void finalize (void)
  /*@globals killed globname@*/
{
  free (globname.firstname);
22}
```

Running LCLint

```
> lclint annotglobs.c
LCLint 2.4 --- 10 Apr 98

annotglobs.c: (in function initialize)
annotglobs.c:14: Undef global globnum used before
                 definition
annotglobs.c:16: Global storage globname contains
   1 undefined field when call returns: firstname
annotglobs.c: (in function finalize)
annotglobs.c:22: Only storage globname.firstname
   (type char *) derived from killed global is
   not released (memory leak)

Finished LCLint checking --- 3 code errors found
```

Figure 14. Annotated globals lists.

The `killed` annotation denotes a global variable that may be undefined when the call returns. For globals that contain dynamically allocated storage, a `killed` global variable is similar to an `only` parameter (Section 5.2). An error is reported if it contains the only reference to storage that is not released before the call returns.

7.2 Null Pointers

A common cause of program failures is when a null pointer is dereferenced. LCLint detects these errors by distinguishing possibly `NULL` pointers at interface boundaries.

The `null` annotation is used to indicate that a pointer value may be `NULL`. A pointer declared with no `null` annotation, may not be `NULL`. If null checking is turned on (controlled by `null`), LCLint will report an error when a possibly null pointer is passed as a parameter, returned as a result, or assigned to an external reference with no `null` qualifier.

If a pointer is declared with the `null` annotation, the code must check that it is not `NULL` on all paths leading to a dereference of the pointer (or the pointer being returned or passed as a value with no `null` annotation). Dereferences of possibly null pointers may be protected by conditional statements or assertions (to see how `assert` is declared see Section 7.3) that check the pointer is not `NULL`.

Consider two implementations of `firstChar` in Figure 15. For `firstChar1`, LCLint reports an error since the pointer that is dereferenced is declared with a `null` annotation. For `firstChar2`, no error is reported since the true branch of the `s == NULL` if statement returns, so the dereference of `s` is only reached if `s` is not `NULL`.

null.c

```
char firstChar1 (/*@null@*/ char *s)
{
3 return *s;
}

char firstChar2 (/*@null@*/ char *s)
{
  if (s == NULL) return '\0';
9 return *s;
}
```

Running LCLint

```
> lclint null.c
LCLint 2.4 --- 10 Apr 98

null.c:3: Dereference of possibly null pointer s: *s
  null.c:1: Storage s may become null

Finished LCLint checking --- 1 code error found
```

No error is reported for line 9, since the dereference is reached only if `s` *is non-null.*

Figure 15. Null checking.

7.2.1 Predicate Functions

Another way to protect null dereference, is to declare a function using `falsenull` or `truenull` and call the function in a conditional statement before the `null`-annotated pointer is dereferenced. The `falsenull` and `truenull` annotations may only be used on return values for functions that return a boolean[19] result and whose first argument is a possibly null pointer.

A function is annotated with `truenull` is assumed to return `TRUE` if its first parameter is `NULL` and `FALSE` otherwise. For example, if `isNull` is declared as,

```
/*@truenull@*/ bool isNull (/*@null@*/ char *x);
```

we could write `firstChar2`:

[19] That is, the return type is `bool`, or `int` if `+boolint` is used.

```
char firstChar2 (/*@null@*/ char *s)
{
   if (isNull (s)) return '\0';
   return *s;
}
```

No error is reported since the dereference of `s` is only reached if `isNull(s)` is false, and since `isNull` is declared with the `truenull` annotation this means `s` must not be null.

The `falsenull` annotation is not quite the opposite of `truenull`. If a function declared with `falsenull` returns `TRUE`, it means its parameter is not `NULL`. If it returns `FALSE`, the parameter may or may not be `NULL`.

For example, we could define `isNonEmpty` to return `TRUE` if its parameter is not `NULL` and has least one character before the `NUL` terminator:

```
/*@falsenull@*/ bool isNonEmpty (/*@null@*/ char *x)
{
  return (x != NULL && *x != '\0');
}
```

LCLint does not check that the implementation of a function declared with `falsenull` or `truenull` is consistent with its annotation, but assumes the annotation is correct when code that calls the function is checked.

7.2.2 Overriding Null Types

The `null` annotation may be used in a type definition to indicate that all instances of the type may be `NULL`. For declarations of a type declared using `null`, the `null` annotation in the type definition may be overridden with `notnull`. This is particularly useful for parameters to hidden `static` operations of abstract types where the null test has already been done before the function is called, or function results of the type which are never `NULL`. For an abstract type, `notnull` may not be used for parameters to external functions, since clients should not be aware of when the concrete representation may by `NULL`. Parameters to static functions in the implementation module, however, may be declared using `notnull`, since they may only be called from places where the representation is accessible. Return values for `static` or external functions may be declared using `notnull`.

Figure 16 gives an example showing the use of `notnull`.

7.2.3 Relaxing Null Checking

An additional annotation, `relnull` may be used to relax null checking (`relnull` is analogous to `reldef` for definition checking). No error is reported when a `relnull` value is dereferenced, or when a possibly null value is assigned to an identifier declared using `relnull`.

This is generally used for structure fields that may or may not be null depending on some other constraint. LCLint does not report and error when `NULL` is assigned to a `relnull` reference, or when a `relnull` reference is dereferenced. It is up to the programmer to ensure that this constraint is satisfied before the pointer is dereferenced.

mstring.c

```
typedef /*@abstract@*/ /*@null@*/ char *mstring;
static /*@notnull@*/ mstring mstring_createNew (int x) ;

mstring mstring_space (void)
{
  mstring m = mstring_createNew (1);
  Because of notnull qualifier on mstring_createNew, can assume m is not null.
  *m = ' '; *(m + 1) = '\0';
  return m;
}
```

Figure 16. Using `notnull`.

7.3 Execution

To detect certain errors and avoid spurious errors, it is important to know something about the control flow behavior of called functions. Without additional information, LCLint assumes that all functions eventually return and execution continues normally at the call site.

The `exits` annotation is used to denote a function that never returns. For example,

```
extern /*@exits@*/ void fatalerror (/*@observer@*/ char *s);
```

declares `fatalerror` to never return. This allows LCLint to correctly analyze code like,

```
if (x == NULL) fatalerror ("Yikes!");
*x = 3;
```

Other functions may exit, but sometimes (or usually) return normally. The `mayexit` annotation denotes a function that may or may not return. This doesn't help checking much, since LCLint must assume that a function declared with `mayexit` returns normally when checking the code.

To be more precise, the `trueexit` and `falseexit` annotations may be used. Similar to `truenull` and `falsenull` (see Section 7.2.1), `trueexit` and `falseexit` mean that a function always exits if the value of its first argument is `TRUE` or `FALSE` respectively. They may be used only on functions whose first argument has a boolean type.

A function declared with `trueexit` must exit if the value of its argument is `TRUE`, and a function declared with `falseexit` must exit if the value of its argument is `FALSE`. For example, the standard library declares `assert` as[20]:

```
/*@falseexit@*/ void assert (/*@sef@*/ bool /*@alt int@*/ pred);
```

This way, code like,

```
assert (x != NULL);
*x = 3;
```

is checked correctly, since the `falseexit` annotation on `assert` means the deference of `x` is not reached is `x != NULL` is false.

[20]The `sef` annotation denotes a parameter as side-effect free (see Section 8.2.1). By declaring the argument to `assert` to be side-effect free, LCLint will report errors if the parameter to `assert` produces a side-effect. This is especially pertinent if assertions are turned off when the production version is compiled. The `bool /*@alt int@*/` type specifier for the parameter means the parameter type must match either `bool` or `int`. Alternate types are described in Section 8.2.2.

7.4 Special Clauses

Sometimes it is necessary to specify function interfaces at a lower level than is possible with the standard annotations. For example, if a function defines some fields of a returned structure but does not define all the fields. The `/*@special@*/` annotation is used to mark a parameter, global variable, or return value that is described using special clauses. The usual implicit definition rules do not apply to a special declaration.

Special clauses may be used to constrain the state of a parameter or return value before or after a call. One or more special clauses may appear in a function declaration, before the modifies or globals clauses. Special clauses may be listed in any order, but the same special clause should not be used more than once. Parameters used in special clauses must be annotated with `/*@special@*/` in the function header. In a special clause list, `result` is used to refer to the return value of the function. If `result` appears in a special clause, the function return value must be annotated with `/*@special@*/`.

The following special clauses are used to describe the definition state or parameters before and after the function is called and the return value after the function returns:

`/*@uses` *<references>*`@*/`
: References in the `uses` clause must be completely defined before the function is called. They are assumed to be defined at function entrance when the function is checked.

`/*@sets` *<references>*`@*/`
: References in the `sets` clause must be allocated before the function is called. They are completely defined after the function returns. When the function is checked, they are assumed to be allocated at function entrance and an error is reported if there is a path on which they are not defined before the function returns.

`/*@defines` *<references>*`@*/`
: References in the `defines` clause must not refer to unshared, allocated storage before the function is called. They are completely defined after the function returns. When the function is checked, they are assumed to be undefined at function entrance and an error is reported if there is a path on which they are not defined before the function returns.

`/*@allocates` *<references>*`@*/`
: References in the `allocates` clause must not refer to unshared, allocated storage before the function is called. They are allocated but not necessarily defined after the function returns. When the function is checked, they are assumed to be undefined at function entrance and an error is reported if there is a path on which they are not allocated before the function returns.

`/*@releases` *<references>*`@*/`
: References in the `releases` clause are deallocated by the function. They must correspond to storage that could be passed as an `only` parameter before the function is called, and are dead pointers after the function returns. When the function is checked, they are assumed to be allocated at function entrance and an error is reported if they refer to live, allocated storage at any return point.

Additional generic special clauses can be used to describe other aspects of the state of inner storage before or after a call. Generic special clauses have the form *`state`*`:`*`constraint`*. The state is either `pre` (before the function is called), or `post` (after the function is called). The constraint is similar to an annotation. The following constraints are supported:

Aliasing Annotations

`pre:only`, `post:only`

`pre:shared`, `post:shared`
`pre:owned`, `post:owned`
`pre:dependent`, `post:dependent`
References refer to `only`, `shared`, `owned` or `dependent` storage before (`pre`) or after (`post`) the call.

Exposure Annotations

`pre:observer`, `post:observer`
`pre:exposed`, `post:exposed`
References refer to `observer` or `exposed` storage before (`pre`) or after (`post`) the call.

Null State Annotations

`pre:isnull`, `post:isnull`
References have the value `NULL` before (`pre`) or after (`post`) the call. Note, this is not the same name or meaning as the null annotation (which means the value *may* be `NULL`.)
`pre:notnull`, `post:notnull`
References do not have the value `NULL` before (`pre`) or after (`post`) the call.

Some examples of special clauses are shown in Figure 17. The defines clause for `record_new` indicates that the `id` field of the structure pointed to by the result is defined, but the `name` field is not. So, `record_create` needs to call `record_setName` to define the name field. Similarly, the releases clause for `record_clearName` indicates that no storage is associated with the `name` field of its parameter after the return, so no failure to deallocate storage message is produced for the call to `free` in `record_free`.

special.c

```
typedef struct
{
  int id;
  /*@only@*/ char *name;
} *record;

static /*@special@*/ record record_new (void)
  /*@defines result->id@*/
{
  record r = (record) malloc (sizeof (*r));

  assert (r != NULL);
  r->id = 3;
  return r;
}

static void
   record_setName (/*@special@*/ record r, /*@only@*/ char *name)
   /*@defines r->name@*/
{
  r->name = name;
}

record record_create (/*@only@*/ char *name)
{
  record r = record_new ();
  record_setName (r, name);
  return r;
}

void record_clearName (/*@special@*/ record r)
   /*@releases r->name@*/
   /*@post:isnull r->name@*/
{
  free (r->name);
  r->name = NULL;
}

void record_free (/*@only@*/ record r)
{
  record_clearName (r);
  free (r);
}
```

Figure 17. Special Clauses.

8. Macros

Macros are commonly used in C programs to implement constants or to mimic functions without the overhead of a function call. Macros that are used to implement functions are a persistent source of bugs in C programs, since they may not behave like the intended function when they are invoked with certain parameters or used in certain syntactic contexts.

LCLint eliminates most of the potential problems by detecting macros with dangerous implementations and dangerous macro invocations. Whether or not a macro definition is checked or expanded normally depends on flag settings and control comments (see Section 8.3). Stylized macros can also be used to define control structures for iterating through many values (see Section 8.4).

8.1 Constant Macros

Macros may be used to implement constants. To get type-checking for constant macros, use the `constant` syntactic comment:

```
/*@constant null char *mstring_undefined@*/
```

Declared constants are not expanded and are checked according to the declaration. A constant with a `null` annotation may be used as `only` storage.

8.2 Function-like Macros

Using macros to imitate functions is notoriously dangerous. Consider this broken macro for squaring a number:

```
# define square(x) x * x
```

This works fine for a simple invocation like `square(i)`. It behaves unexpectedly, though, if it is invoked with a parameter that has a side effect.

For example, `square(i++)` expands to `i++ * i++`. Not only does this give the incorrect result, it has undefined behavior since the order in which the operands are evaluated is not defined. (See Section 10.1 for more information on how expressions exhibiting undefined evaluation order behavior are detected by LCLint.) To correct the problem we either need to rewrite the macro so that its parameter is evaluated exactly once, or prevent clients from invoking the macro with a parameter that has a side-effect.

Another possible problem with macros is that they may produce unexpected results because of operator precedence rules. The invocation, `square(i+1)` expands to `i+1*i+1`, which evaluates to `i+i+1` instead of the square of `i+1`. To ensure the expected behavior, the macro parameter should be enclosed in parentheses where it is used in the macro body.

Macros may also behave unexpectedly if they are not syntactically equivalent to an expression. Consider the macro definition,

```
# define incCounts()  ntotal++; ncurrent++;
```

This works fine, unless it is used as a statement. For example,

```
if (x < 3) incCounts();
```

increments `ntotal` if `x < 3` but always increments `ncurrent`.

One solution is to use the comma operator to define the macro:

```
# define incCounts()  (ntotal++, ncurrent++)
```

More complicated macros can be written using a `do` … `while` construction:

```
# define incCounts() \
   do { ntotal++; ncurrent++; } while (FALSE)
```

LCLint detects these pitfalls in macro definitions, and checks that a macro behaves as much like a function as possible. A client should only be able to tell that a function was implemented by a macro if it attempts to use the macro as a pointer to a function.

LCLint does these checks on a macro definition corresponding to a function:

- Each parameter to a macro (except those declared to be side-effect free, see Section 8.2.1) must be used exactly once in all possible executions of the macro, so side-effecting arguments behave as expected.[21] (Controlled by `macroparams`.)
- A parameter to a macro may not be used as the left-hand side of an assignment expression or as the operand of an increment or decrement operator in the macro text, since this produces non-functional behavior. (Controlled by `macroassign`.)
- Macro parameters must be enclosed in parentheses when they are used in potentially dangerous contexts. (Controlled by `macroparens`.)
- A macro definition must be syntactically equivalent to a statement when it is invoked followed by a semicolon. (Controlled by `macrostmt`.)
- The type of the macro body must match the return type of the corresponding function. If the macro is declared with type `void`, its body may have any type but the macro value may not be used.
- All variables declared in the body of a macro definition must be in the macro variable namespace, so they do not conflict with variables in the scope where the macro is invoked (which may be used in the macro parameters). By default, the macro namespace is all names prefixed by `m_`. (See Section 9.2 for information on controlling namespaces.)

At the call site, a macro is checked like any other function call.

8.2.1 Side-Effect Free Parameters

Suppose we really do want to implement `square` as a macro, but want do so in a safe way. One way to do this is to require that it is never invoked with a parameter that has a side-effect. LCLint will check that this constraint holds, if the parameter is annotated to be *side-effect free*. That is, the expression corresponding to this parameter must not modify any state, so it does not matter how many times it is evaluated. The `sef` annotation is used to denote a parameter that may not have any side-effects:

```
extern int square (/*@sef@*/ int x);
# define square(x) ((x) *(x))
```

Now, LCLint will not report an error checking the definition of `square` even though `x` is used more than once.

A message will be reported, however, if `square` is invoked with a parameter that has a side-effect. For the code fragment,

```
square (i++)
```

[21] To be completely correct, all the macro parameters should be evaluated before the macro has any side-effects. Since checking this would require extensive analysis for occasional modest gain, it was not considered worth implementing.

LCLint produces the message:

> Parameter 1 to square is declared sef, but the argument may modify i: i++

It is also an error to pass a non-`sef` macro parameter as a `sef` macro parameter in the body of a macro definition. For example,

```
extern int sumsquares (int x, int y);
# define sumsquares(x,y) (square(x) + square(y))
```

Although `x` only appears once in the definition of `sumsquares` it will be evaluated twice since `square` is expanded. LCLint reports an error when a non-`sef` macro parameter is passed as a `sef` parameter.

A parameter may be passed as a `sef` parameter without an error being reported, if LCLint can determine that evaluating the parameter has no side-effects. For function calls, the modifies clause is used to determine if a side-effect is possible.[22] To prevent many spurious errors, if the called function has no modifies clause, LCLint will report an error only if `sef-uncon` is on. Justifiably paranoid programmers will insist on setting `sef-uncon` on, and will add modifies clauses to unconstrained functions that are used in `sef` macro arguments.

8.2.2 Polymorphism

One problem with our new definition of `square` is that while the original macro would work for parameters of any numeric type, LCLint will now report an error is the new version is used with a non-integer parameter.

We can use the `/*@alt <type>,+@>` syntax to indicate that an alternate type may be used. For example,

```
extern int /*@alt float@*/ square (/*@sef@*/ int /*@alt float@*/ x);
# define square(x) ((x) *(x))
```

declares `square` for both `int`s and `float`s.

Alternate types are also useful for declaring functions for which the return value may be safely ignored (see Section 10.3.2).

8.3 Controlling Macro Checking

By default, LCLint expands macros normally and checks the resulting code after macros have been expanded. Flags and control comments may be used to control which macros are expanded and which are checked as functions or constants.

If the `fcn-macros` flag is on, LCLint assumes all macros defined with parameter lists implement functions and checks them accordingly. Parameterized macros are not expanded and are checked as functions with unknown result and parameter types (or using the types in the prototype, if one is given). The analogous flag for macros that define constants is `const-macros`. If it is on, macros with no parameter lists are assumed to be constants, and checked accordingly. The `all-macros` flag sets both `fcn-macros` and `const-macros`. If the `macro-fcn-decl` flag is set, a message reports parameterized macros with no corresponding function prototype. If the `macro-const-decl` flag is set, a similar message reports macros with no parameters that have no corresponding constant declaration.

---

[22] Note that functions which do not produce to the same result each time they are called with the same arguments should be declared to modify `internalState` so they will lead to errors if they are passed as `sef` parameters.

The macro checks described in the previous sections make sense only for macros that are intended to replace functions or constants. When `fcnmacros` or `constmacros` is on, more general macros need to be marked so they will not be checked as functions or constants, and will be expanded normally. Macros which are not meant to behave like functions should be preceded by the `/*@notfunction@*/` comment. For example,

```
/*@notfunction@*/
# define forever for(;;)
```

Macros preceded by `notfunction` are expanded normally before regular checking is done. If a macro that is not syntactically equivalent to a statement without a semi-colon (e.g., a macro which enters a new scope) is not preceded by `notfunction`, parse errors may result when `fcn-macros` or `const-macros` is on.

8.4 Iterators

It is often useful to be able to execute the same code for many different values. For example, we may want to sum all elements in an `intSet` that represents a set of integers. If `intSet` is an abstract type, there is no easy way of doing this in a client module without depending on the concrete representation of the type. Instead, we could provide such a mechanism as part of the type's implementation. We call a mechanism for looping through many values an *iterator*.

The C language provides no mechanism for creating user-defined iterators. LCLint supports a stylized form of iterators declared using syntactic comments and defined using macros.

Iterator declarations are similar to function declarations except instead of returning a value, they assign values to their `yield` parameters in each iteration. For example, we could add this iterator declaration to `intSet.h`:

```
/*@iter intSet_elements (intSet s, yield int el);@*/
```

The `yield` annotation means that the variable passed as the second actual argument is declared as a local variable of type `int` and assigned a value in each loop iteration.

Defining Iterators

An iterator is defined using a macro. Here's one (not particularly efficient) way of defining `intSet_elements`:

```
typedef /*@abstract@*/ struct {
   int nelements;
   int *elements;
} intSet;
…
# define intSet_elements(s,m_el) \
  { int m_i; \
    for (m_i = (0); m_i <= ((s)->nelements); m_i++) { \
        int m_el = (s)->elements[(m_i)];

# define end_intSet_elements }}
```

Each time through the loop, the yield parameter `m_el` is assigned to the next value. After each value has been assigned to `m_el` for one iteration, the loop terminates. Variables declared by the iterator macro (including the `yield` parameter) are preceded by the macro variable namespace prefix `m_` (see Section 8.2) to avoid conflicts with variables defined in the scope where the iterator is used.

Using Iterators

The general structure for using an iterator is,

```
iter (<params>) stmt; end_iter
```

For example, a client could use `intSet_elements` to sum the elements of an `intSet`:

```
intSet s;
int sum = 0;
...
intSet_elements (s, el) {
   sum += el;
} end_intSet_elements;
```

The actual parameter corresponding to a yield parameter, `el`, is not declared in the function scope. Instead, it is declared by the iterator and assigned to an appropriate value for each iteration.

LCLint will do the following checks for uses of stylized iterators:

- An invocation of the iterator *`iter`* must be balanced by a corresponding end, named `end_`*`iter`*.
- All actual parameters must be defined, except those corresponding to yield parameters.
- Yield parameters must be new identifiers, not declared in the current scope or any enclosing scope.

Iterators are a bit awkward to implement, but they enable compact, easily understood client code. For abstract collection types, an iterator can be used to enable clients to operate on elements of the collection without breaking data abstraction.

9. Naming Conventions

Naming conventions tend to be a religious issue. Generally, it doesn't matter too much what naming convention is followed as long as one is chosen and followed religiously. There are two kinds of naming conventions supported by LCLint. Type-based naming conventions (Section 9.1) constrain identifier names according to the abstract types that are accessible where the identifier is defined. Prefix naming conventions (Section 9.2) constrain the initial characters of identifier names according to what is being declared and its scope. Naming conventions may be combined or different conventions may be selected for different kinds of identifiers. In addition, LCLint supports checking that names do not conflict with names reserved for the standard library or implementation (Section 9.3) and that names are sufficiently distinguishable from other names.

9.1 Type-Based Naming Conventions

Generic naming conventions constrain valid names of identifiers. By limiting valid names, namespaces may be preserved and programs may be more easily understood since the name gives clues as to how and where the name is defined and how it should be used.

Names may be constrained by the scope of the name (external, file static, internal), the file in which the identifier is defined, the type of the identifier, and global constraints.

9.1.1 Czech Names

Czech[23] names denote operations and variables of abstract types by preceding the names by *<type>*_. The remainder of the name should begin with a lowercase character, but may use any other character besides the underscore. Types may be named using any non-underscore characters.

The Czech naming convention is selected by the `czech` flag. If `access-czech` is on, a function, variable, constant or iterator named *<type>*_*<name>* has access to the abstract type *<type>*.

Reporting of violations of the Czech naming convention is controlled by different flags depending on what is being declared:

`czech-fcns`
: Functions and iterators. An error is reported for a function name of the form *<prefix>*_*<name>* where *<prefix>* is not the name of an accessible type. Note that if `accessczech` is on, a type named *<prefix>* would be accessible in a function beginning with *<prefix>*_. If `access-czech` is off, an error is reported instead. An error is reported for a function name that does not have an underscore if any abstract types are accessible where the function is defined.

[23] The most renowned C naming convention is the Hungarian naming convention, introduced by Charles Simonyi [Simonyi, Charles, and Martin Heller. "The Hungarian Revolution." *BYTE*, August 1991, p. 131-38]. The names for LCLint naming conventions follow the tradition of using Central European nationalities as mnemonics for naming conventions. The LCLint conventions are similar to the Hungarian naming convention in that they encode type information in names, except that the LCLint conventions encode the names of accessible abstract types instead of the type of the declaration of return value. Prefixes used in the Hungarian naming convention are not supported by LCLint.

`czech-vars`
`czech-constants`
`czech-macros`
Variables, constants and expanded macros. An error is reported if the identifier name starts with *<prefix>_* and *prefix* is not the name of an accessible abstract type, or if an abstract type is accessible and the identifier name does not begin with *<type>_* where `type` is the name of an accessible abstract type. If `access-czech` is on, the representation of the type is visible in the constant or variable definition.

`czech-types`
User-defined types. An error is reported if a type name includes an underscore character.

Of course, this is a complete jumble to the uninitiated, and that's the joke.
Charles Simonyi, on the Hungarian naming convention

9.1.2 Slovak Names

Slovak names are similar to Czech names, except they are spelled differently. A Slovak name is of the form *<type><Name>*. The type prefix may not use uppercase characters. The remainder of the name starts with the first uppercase character.

The `slovak` flag selects the Slovak naming convention. Like Czech names, it may be used with `access-slovak` to control access to abstract representations. The `slovak-fcns`, `slovak-vars`, `slovak-constants`, and `slovak-macros` flags are analogous to the similar Czech flags. If `slovak-type` is on, an error is reported if a type name includes an uppercase letter.

9.1.3 Czechoslovak Names

Czechoslovak names are a combination of Czech names and Slovak names. Operations may be named either *<type>_* followed by any sequence of non-underscore characters, or *<type>* followed by an uppercase letter and any sequence of characters. Czechoslovak names have been out of favor since 1993, but may be necessary for checking legacy code. The `czechoslovak-fcns`, `czechoslovak-vars`, `czechoslovak-macros`, and `czechoslovak-constants` flags are analogous to the similar Czech flags. If `czechoslovak-type` is on, an error is reported if a type name contains either an uppercase letter or an underscore character.

9.2 Namespace Prefixes

Another way to restrict names is to constrain the leading character sequences of various kinds of identifiers. For example, the names of all user-defined types might begin with "`T`" followed by an uppercase letter and all file static names begin with an uppercase letter. This may be useful for enforcing a namespace (e.g., all names exported by the X-windows library should begin with "`X`") or just making programs easier to understand by establishing an enforced convention. LCLint can be used to constrain identifiers in this way to detect identifiers inconsistent with prefixes.

All namespace flags are of the form, `-`*<context>*`prefix` *<string>*. For example, the macro variable namespace restricting identifiers declared in macro bodies to be preceded by "`m_`" would be selected by `-macrovarprefix "m_"`. The string may contain regular characters that may appear in a C identifier. These must match the initial characters of the identifier name. In addition, special characters (shown in Table 1) can be used to denote a class of characters.[24] The `*` character may be used at the end of a prefix string to specify the rest of the identifier is zero or more characters matching the character immediately before the `*`. For example, the prefix string "`T&*`" matches "`T`" or "`TWINDOW`" but not "`Twin`".

[24] Namespace prefixes should probably be described by regular expressions. LCLint uses a simpler, more limited means for describing names, which is believed to be adequate for describing most useful naming conventions. If there is sufficient interest, regular expressions may be supported in a future version of LCLint.

^	Any uppercase letter, A-Z
&	Any lowercase letter, a-z
%	Any character that is not an uppercase letter (allows lowercase letters, digits and underscore)
~	Any character that is not a lowercase letter (allows uppercase letters, digits and underscore)
$	Any letter (a-z, A-Z)
/	Any letter or digit (A-Z, a-z, 0-9)
?	Any character valid in a C identifier
#	Any digit, 0-9

Table 1. Prefix character codes.

Different prefixes can be selected for the following identifier contexts:

`macro-var-prefix`	Any variable declared inside a macro body
`unchecked-macro-prefix`	Any macro that is not checked as a function or constant (see Section 8.4)
`tag-prefix`	Tags for `struct`, `union` and `enum` declarations
`enum-prefix`	Members of `enum` types
`type-prefix`	Name of a user-defined type
`file-static-prefix`	Any identifier with file static scope
`glob-var-prefix`	Any variable (not of function type) with global scope
`const-prefix`	Any constant (see Section 8.1)
`iter-prefix`	An iterator (see Section 8.4)
`proto-param-prefix`	A parameter in a function declaration prototype
`external-prefix`	Any exported identifier

If an identifier is in more than one of the namespace contexts, the most specific defined namespace prefix is used (e.g., a global variable is also an exported identifier, so if `global-var-prefix` is set, it is checked against the variable name; if not, the identifier is checked against the `external-prefix`.)

For each prefix flag, a corresponding flag named *<prefixname>*`exclude` controls whether errors are reported if identifiers in a different namespace match the namespace prefix. For example, if `macro-var-prefix-exclude` is on, LCLint checks that no identifier that is not a variable declared inside a macro body uses the macro variable prefix.

Here is a (somewhat draconian) sample naming convention:

`-unchecked-macro-prefix "~*"`	unchecked macros have no lowercase letters
`-type-prefix "T^&*"`	all type names begin with T followed by an uppercase letter. The rest of the name is all lowercase letters.
`+type-prefix-exclude`	no identifier that does no name a user-defined type name begin with the type name prefix (set above)
`-file-static-prefix "^&&&"`	file static scope variables begin with an uppercase letter and three lowercase letters
`-proto-param-prefix "p_"`	all parameters in prototypes must begin with `P_`
`-glob-var-prefix "G"`	all global variables start with `G`
`+glob-var-prefix-exclude`	no identifier that is not a global variable starts with `G`

The prefix for parameters in function prototypes is useful for making sure parameter names are not in conflict with macros defined before the function prototype. In most cases, it may be preferable to

not name prototype parameters. If the `proto-param-name` flag is set, an error is reported for any named parameter in a prototype declaration. If a `proto-param-prefix` is set, no error is reported for unnamed parameters.

It may also be useful to check the names of prototype parameters correspond to the names in definitions.[25] If `proto-param-match` is set, LCLint will report an error if the name of a definition parameter does not match the corresponding prototype parameter (after removing the `protoparamprefix`).

9.3 Naming Restrictions

Additional naming restrictions can be used to check that names do no conflict with names reserved for the standard library, and that identifier are sufficiently distinct (either for the compiler and linker, or for the programmer.) Restrictions may be different for names that are needed by the linker (*external* names) and names that are only needed during compilations (*internal* names). Names of non-`static` functions and global variables are external; all other names are internal.

9.3.1 Reserved Names

Many names are reserved for the implementation and standard library. A complete list of reserved names can be found in [vdL, p. 126-128] or [ANSI, Section 4]. Some name prefixes such as `str` followed by a lowercase character are reserved for future library extensions. Most C compilers do not detect naming conflicts, and they can lead to unpredictable program behavior. If `ansi-reserved` is on, LCLint reports errors for external names that conflict with reserved names. If `ansi-reserved-internal` is on, errors are also reported for internal names.

If the `cpp-names` flag is set, LCLint will report identifier names that are keywords or reserved words in C++. This is useful if the code may later be compiled with a C++ compiler (of course, this is not enough to ensure the meaning of the code is not changed when it is compiled as C++.)

9.3.2 Distinct Identifiers

LCLint can check that identifiers differ within a given number of characters, optionally ignoring alphabetic case and differences between characters that look similar. The number of significant characters may be different for external and internal names.

Using `+distinct-external-names` sets the number of significant characters for external names to six and makes alphabetical case insignificant for external names. This is the minimum significance acceptable in an ANSI-conforming compiler. Most modern compilers exceed these minimums (which are particularly hard to follow if one uses the Czech or Slovak naming convention). The number of significant characters can be changed using the `external-name-length` *<number>* flag. If `external-name-case-insensitive` is on, alphabetical case is ignored in comparing external names. LCLint reports identifiers that differ only in alphabetic case.

For internal identifiers, a conforming compiler must recognize at least 31 characters and treat alphabetical cases distinctly. Nevertheless, it may still be useful to check that internal names are more distinct then required by the compiler to minimize the likelihood that identifiers are confused in the program. Analogously to external names, the `internal-name-length` *<number>* flag sets the number of significant characters in an internal name and `internal-name-case-`

[25] While using header files as documentation is not generally recommended, it is common enough practice that it makes sense to check that parameter names are consistent. A discrepancy may indicate an error in the parameter order in the function prototype.

`insensitive` sets the case sensitivity. The `internal-name-look-alike` flag further restricts distinctions between identifiers. When set, similar-looking characters match — the lowercase letter "`l`" matches the uppercase letter "`I`" and the number "`1`"; the letter "`O`" or "`o`" matches the number "`0`"; "`5`" matches "`S`"; and "`2`" matches "`Z`". Identifiers that are not distinct except for look-alike characters will produce an error message. External names are also internal names, so they must satisfy both the external and internal distinct identifier checks.

names.c

```
1 char *stringrev (char *s);

3 int f (int x)
  {
5 int lookalike = 1;
6 int looka1ike = 2;

  if (x > 3)
    {
10    int x = lookalike;
       x += looka1ike;
    }

  return x;
}
```

Running LCLint

```
> lclint names.c +distinctinternalnames +internalnamelookalike
          +ansireserved
LCLint 2.4 --- 10 Apr 98

names.c:1: Name stringreverse is reserved for future ANSI library
   extensions.  Functions that begin with "str" and a lowercase
   letter may be added to <stdlib.h> or <string.h>.  (See ANSI,
   Section 4.13.7)
names.c:6: Internal identifier looka1ike is not distinguishable from
          lookalike except by lookalike characters
  names.c:5: Declaration of lookalike
names.c:10: Variable x shadows outer declaration
  names.c:3: Previous declaration of x: int

Finished LCLint checking --- 3 code errors found
```

Figure 18. Naming checks.

The decision to retain the old six-character case-insensitive restriction on significance was most painful.

ANSI C Rationale

10. Other Checks

The section describes other errors detected by LCLint that are not directly related to extra information provided in annotations. Many of the checks are significantly improved, however, because of the extra information that is known about the program.

10.1 Undefined Evaluation Order

The order in which side effects take place in a C program is not entirely defined by the code. Certain execution points are known as *sequence points* — a function call (after the arguments have been evaluated), the end of a full expression (an initializer, expression in an expression statement, the control expression of an `if`, `switch`, `while` or `do` statement, each expression of a `for` statement, and the expression in a `return` statement), and after the first operand or a `&&`, `||`, `?` or `,` operand.

All side effects before a sequence point must be complete before the sequence point, and no evaluations after the sequence point shall have taken place [ANSI, Section 2.1.2.3]. Between sequence points, side effects and evaluations may take place in any order. Hence, the order in which expressions or arguments are evaluated is not specified. Compilers are free to evaluate function arguments and parts of expressions (that do not contain sequence points) in any order. The behavior of code is undefined if it uses a value that is modified by another expression that is not required to be evaluated before or after the other use.

LCLint detects instances where undetermined order of evaluation produces undefined behavior. If modifies clauses and globals lists are used, this checking is enabled in expressions involving function calls. Evaluation order checking is controlled by the `eval-order` flag.

order.c

```
extern int glob;

extern int mystery (void);

extern int modglob (void)
   /*@globals glob@*/
   /*@modifies glob@*/ ;

int f (int x, int y[])
{
11 int i = x++ * x;

13 y[i] = i++;
14 i += modglob() * glob;
15 i += mystery() * glob;
16 return i;
}
```

Running LCLint

```
> lclint order.c +evalorderuncon
LCLint 2.4 --- 10 Apr 98

order.c:11: Expression has undefined behavior (value of right
            operand modified by left operand): x++ * x
order.c:13: Expression has undefined behavior (left operand uses
            i, modified by right operand): y[i] = i++
order.c:14: Expression has undefined behavior (value of right
   operand modified by left operand): modglob() * glob
      From the modifies clause, modglob may modify glob.
      The behavior is undefined since we don't know if glob
      is evaluated before, after or during the modification.
order.c:15: Expression has undefined behavior (unconstrained
   function mystery used in left operand may set global variable
   glob used in right operand): mystery() * glob
     Not reported without +evalorderuncon.

Finished LCLint checking --- 4 code errors found
```

Figure 19. Evaluation order

When checking systems without modifies and globals information, evaluation order checking may report errors when unconstrained functions are called in procedure arguments. Since LCLint has no annotations to constrain what these functions may modify, it cannot be guaranteed that the evaluation order is defined if another argument calls an unconstrained function or uses a global

variable or storage reachable from a parameter to the unconstrained function. Its best to add modifies and globals clauses to constrain the unconstrained functions in ways that eliminate the possibility of undefined behavior. For large legacy systems, this may require too much effort. Instead, the `-eval-order-uncon` flag may be used to prevent reporting of undefined behavior due to the order of evaluation of unconstrained functions.

10.2 Problematic Control Structures

A number of control structures that are syntactically legal may indicate likely bugs in programs. LCLint can detect errors involving likely infinite loops (Section 10.2.1), fall through cases and missing cases in `switch` statements (Section 10.2.2), `break` statements within deeply nested loops or switches (Section 10.2.3), clauses of `if`, `while` or `for` statements that are empty statements or unblocked single statements (Section 10.2.4) and incomplete if-else logic (Section 10.2.5). Although any of these may appear in a correct program, depending on the programming style used they may indicate likely bugs or style violations that should be detected and eliminated.

10.2.1 Likely Infinite Loops

LCLint reports an error if it detects a loop that appears to be infinite. An error is reported for a loop that does not modify any value used in its condition test inside the body of the loop or in the condition test itself. This checking is enhanced by modifies clauses and globals lists since they provide more information about what global variable may be used in the condition test and what values may be modified by function calls in the loop body.

Figure 20 shows examples of infinite loops detected by LCLint. An error is reported for the loop in line 14, since neither of the values used in the loop condition (`x` directly and `glob1` through the call to `f`) is modified by the body of the loop. If the declaration of `g` is changed to include `glob1` in the modifies clause no error is reported. (In this example, if we assume the annotations are correct, then the programmer has probably called the wrong function in the loop body. This isn't surprising, given the horrible choices of function and variable names!)

If an unconstrained function is called within the loop body, LCLint will assume that it modifies a value used in the condition test and not report an infinite loop error, unless `infloopsuncon` is on. If `infloopsuncon` is on, LCLint will report infinite loop errors for loops where there is no explicit modification of a value used in the condition test, but where they may be an undetected modification through a call to an unconstrained function (e.g., line 15 in Figure 20).

10.2.2 Switches

The automatic fall-through of C switch statements is almost never the intended behavior.[26] LCLint detects `case` statements with code that may fall through to the next `case`. The `casebreak` flag controls reporting of fall through cases. A single fall through case may be marked by preceding the `case` keyword with `/*@fallthrough@*/` to indicate explicitly that execution falls through to this case.

[26] Peter van der Linden estimates that default fall through is the wrong behavior 97% of the time. [vdL95, p. 37]

loop.c	Running LCLint

`extern int glob1, glob2;`
 	
`extern int f (void)`
`  /*@globals glob1@*/`
`  /*@modifies nothing@*/ ;`
 	
`extern void g (void)`
`  /*@modifies glob2@*/ ;`
 	
`extern void h (void) ;`
 	
`void upto (int x)`
`{`
`14  while (x > f()) g();`
`15  while (f() < 3) h();`
`}`
	> lclint loop.c +infloopsuncon
	LCLint 2.4 --- 10 Apr 98
	
	loop.c: (in function upto)
	loop.c:14: Suspected infinite loop. No value used in loop test (x, glob1) is modified by test or loop body.
	loop.c:15: Suspected infinite loop. No condition values modified. Modification possible through unconstrained calls: h
	
	Finished LCLint checking --- 2 code errors found
	
	An error is reported for line 14 since the only value modified by the loop test or body if `glob2` *and the value of the loop test does not depend on* `glob2`*.*
	
	The error for line 15 would not be reported if `+infloopsuncon` *wasn't used.*

Figure 20. Infinite loop checking

For switches on `enum` types, LCLint reports an error if a member of the enumerator does not appear as a case in the switch body (and there is no `default` case). (Controlled by `misscase`.)

switch.c	Running LCLint

`typedef enum {`
`  YES, NO, DEFINITELY,`
`  PROBABLY, MAYBE } ynm;`
 	
`void decide (ynm y)`
`{`
`  switch (y)`
`  {`
`  case PROBABLY:`
`  case NO: printf ("No!");`
`11  case MAYBE: printf ("Maybe");`
`  /*@fallthrough@*/`
`  case YES: printf ("Yes!");`
`14  }`
`}`
	> lclint switch.c
	LCLint 2.4 --- 10 Apr 98
	
	switch.c: (in function decide)
	switch.c:11: Fall through case (no preceding break)
	switch.c:14: Missing case in switch: DEFINITELY
	
	Finished LCLint checking --- 2 code errors found
	
	No fall through error is reported for the `NO` *case, since there are no statements associated with the previous case. The* `/*@fallthrough@*/` *comment prevents a message from being produced for the* `YES` *case.*

Figure 21. Switch checking.

10.2.3 Deep Breaks

There is no syntax provided by C (other than `goto`) for breaking out of a nested loop. All `break` and `continue` statements act only on the innermost surrounding loop or switch. This often leads to serious problems[27] when a programmer intends to break the outer loop or switch instead. LCLint optionally reports errors for `break` and `continue` statements in nested contexts.

Four types of `break` errors are reported:

[27] "Software Glitch Cripples AT&T Network", Telephony, 22 January 1990.

- `break` inside a loop (`while` or `for`) that is inside a loop. Controlled by `loploopbreak`. To indicate that a `break` is inside an inner loop, precede the `break` by `/*@innerbreak@*/`.
- `break` inside a loop that is inside a `switch` statement. Controlled by `switchloopbreak`. To mark the `break` as a loop break, precede the `break` by `/*@loopbreak@*/`.
- `break` inside a `switch` statement that is inside a loop. Controlled by `loopswitchbreak`. To mark the `break` as a switch break, precede the `break` by `/*@switchbreak@*/`.
- `break` inside a `switch` inside another `switch`. Controlled by `switchswitchbreak`. To indicate that the `break` is for the inner switch, use `/*@innerbreak@*/`.

Since `continue` only makes sense within loops, errors are only reported for `continue` statements within nested loops. (Controlled by `loploopcontinue`.) A safe inner `continue` may be preceded by `/*@innercontinue@*/` to suppress error messages locally. The `deepbreak` flag sets all nested break and continue checking flags.

LCLint reports an error if the marker preceding a `break` is not consistent with its effect. An error is reported if `innerbreak` precedes a `break` that is not breaking an inner loop, `switchbreak` precedes a `break` that is not breaking a switch, or `loopbreak` precedes a `break` that is not breaking a loop.

10.2.4 Loop and If Bodies

An empty statement after an `if`, `while` or `for` often indicates a potential bug. A single statement (i.e., not a compound block) after an `if`, `while` or `for` is not likely to indicate a bug, but make the code harder to read and edit. LCLint can report errors for if or loop statements with empty bodies or bodies that are not compound statements. Separate flags control checking for statements following an `if`, `while` or `for`:

- `[if, while, for]empty` — report errors for empty bodies (e.g., `if (x > 3) ;`)
- `[if, while, for]block` — report errors for non-block bodies (e.g., `if (x > 3) x++;`)

The `if` statement checks also apply to the body of the `else` clause. An `ifblock` error is not reported if the body of the `else` clause is an `if` statement, to allow `else if` chains.

10.2.5 Complete if-else Logic

Although it may be perfectly reasonable in many contexts, an `if-else` chain with no final `else` may indicate missing logic or forgetting to check error cases. If `elseif-complete` is on, LCLint reports errors when an `if` statement that is the body of an `else` clause does not have a matching `else` clause. For example, the code,

```
if (x == 0) { return "nil"; }
else if (x == 1) { return "many"; }
```

produces an error message since the second `if` has no matching `else` branch.

10.3 Suspicious Statements

LCLint detects errors involving statements with no apparent effects (Section 10.3.1) and statements that ignore the result of a called function (Section 10.3.2).

10.3.1 Statements with No Effects

LCLint can report errors for statements that have no effect. (Controlled by `no-effect`.) Because of modifies clauses, LCLint can detect more errors than traditional checkers. Unless the `no-effect-uncon` flag is on, errors are not reported for statements that involve calls to unconstrained functions since the unconstrained function may cause a modification.

noeffect.c	Running LCLint

extern void
  nomodcall (int *x) /*@*/;
  Recall that /@*/ is shorthand for*
  modifies nothing and use no globals.
extern void mysterycall (int *x);
 	
int noeffect (int *x, int y)
{
  y == *x;
  nomodcall (x);
  mysterycall (x);
  return *x;
}
	> lclint noeffect.c +noeffectuncon
	LCLint 2.4 --- 10 Apr 98
	
	noeffect.c:6: Statement has no effect: y == *x
	noeffect.c:7: Statement has no effect: nomodcall(x)
	noeffect.c:8: Statement has no effect (possible undetected modification through call to unconstrained function mysterycall): mysterycall(x)
	Not reported without +noeffectuncon.
	
	Finished LCLint checking --- 3 code errors found

Figure 22. Statements with no effect.

10.3.2 Ignored Return Values

LCLint reports an error when a return value is ignored. Checking may be controlled based on the type of the return value: `ret-val-int` controls reporting of ignored return values of type `int`, and `ret-val-bool` for return values of type `bool`, and `ret-val-others` for all other types. A function statement may be cast to `void` to prevent this error from being reported.

Alternate types (Section 8.2.2) can be used to declare functions that return values that may safely be ignored by declaring the result type to alternately by `void`. Several functions in the standard library are specified to alternately return `void` to prevent ignored return value errors for standard library functions (e.g., `strcpy`) where the result may be safely ignored (see Appendix F).

Figure 23 shows example of ignored return value errors reported by LCLint.

10.4 Unused Declarations

LCLint detects constants, functions, parameters, variables, types, enumerator members, and structure or union fields that are declared but never used. The flags `constuse`, `fcnuse`, `paramuse`, `varuse`, `typeuse`, `enummemuse` and `fieldus`e control whether unused declaration errors are reported for each kind of declaration. Errors for exported declarations are reported only if `topuse` is on (see Section 10.5).

The `/*@unused@*/` annotation can be used before a declaration to indicate that the item declared need not be used. Unused declaration errors are not reported for identifiers declared with `unused`.

ignore.c

```
extern int fi (void);
extern bool fb (void);
extern int /*@alt void@*/
  fv (void);

int ignore (void)
{
 8  fi ();
 9  (void) fi ();
10  fb ();
11  fv ();
12  return fv ();
}
```

Running LCLint

```
> lclint ignore.c
LCLint 2.4 --- 10 Apr 98

ignore.c: (in function ignore)
ignore.c:8: Return value (type int) ignored: fi()
ignore.c:10: Return value (type bool) ignored: fb()

Finished LCLint checking --- 2 code errors found
```

The message for line 8 would not be reported if `-retvalint` *is set; for line 10, if* `-retvalbool` *is set. No message is reported for line 9 because the result is cast to* `void`*, and no message is reported for line 11 because* `fv` *is declared to alternately return* `void`.

Figure 23. Ignored return values.

10.5 Complete Programs

LCLint can be used on both complete and partial programs. When checking complete programs, additional checks can be done to ensure that every identifier declared by the program is defined and used, and that functions that do not need to be exported are declared `static`.

LCLint checks that all declared variables and functions are defined (controlled by `compdef`). Declarations of functions and variables that are defined in an external library, may be preceded by `/*@external@*/` to suppress undefined declaration errors.

LCLint reports external declarations that are unused (controlled by `topuse`). Which declarations are reported also depends on the declaration use flags (see Section 10.4).

The `partial` flag sets flags for checking a partial system. Top-level unused declarations, undefined declarations, and unnecessary external names are not reported if `partial` is set.

10.5.1 Unnecessary External Names

LCLint can report variables and functions that are declared with global scope (i.e., without using `static`), that are not used outside the file in which they are defined. In a stand-alone system, these identifiers should usually be declared using `static` to limit their scope. If the `export-static` flag is on, LCLint will report declarations that could have file scope. It should only be used when all relevant source files are listed on the LCLint command line; otherwise, variables and functions may be incorrectly identified as only used in the file scope since LCLint did not process the other file in which they are used.

10.5.2 Declarations Missing from Headers

A common practice in C programming styles, is that every function or variable exported by *M*`.c` is declared in *M*`.h`. If the `export-header` flag is on, LCLint will report exported declarations in *M*`.c` that are not declared in *M*`.h`.

10.6 Compiler Limits

The ANSI Standard includes limits on minimum numbers that a conforming compiler must support. Whether of not a particular compiler exceeds these limits, it is worth checking that a program does not exceed them so that other compilers may safely compile it. In addition, exceeding a limit may indicate a problem in the code (e.g., it is too complex if the control nest depth limit is exceeded) that should be fixed regardless of the compiler. LCLint checks the following limits. For each limit, the maximum value may be set from the command line (or locally using a stylized comment). If the `ansi-limits` flag is on, all limits are checked with the minimum values of a conforming compiler.

`include-nest`
: Maximum nesting depth of file inclusion (`#include`). (ANSI minimum is 8)

`control-nest-depth`
: Maximum nesting of compound statements, control structures. (ANSI minimum is 15)

`num-enum-members`
: Number of members in an `enum` declaration. (ANSI minimum is 127)

`num-struct-fields`
: Number of fields in a `struct` or `union` declaration. (ANSI minimum is 127)

Since human beings themselves are not fully debugged yet, there will be bugs in your code no matter what you do.
Chris Mason, Zero-defects memo (Microsoft Secrets, Cusumano and Selby)

From	bnelson@netcom.com (Bob Nelson)
Subject	Re: NT vs. Linux
Date	Fri, 5 Jul 1996 05:11:22 GMT
Newsgroups	comp.os.linux.advocacy,comp.sys.ibm.pc.hardware, comp.os.ms-windows.win95.misc, comp.os.mswindows.nt.misc, alt.flame,alt.fan.bill-gates,alt.destroy.microsoft

---

Toni Anzlovar (toni.anzlovar@kiss.uni-lj.si) wrote:

> Why does everybody want to RUN WORD? Why does nobody want to write and edit
> text?

*Simple. A *tremendous* number of documents are written using Microsoft Word. One that is particularly ironic is the guide to LCLint -- a very popular lint tool -- often the lint of choice in the linux world.*

Appendix A Availability

The web home page for LCLint is

```
http://www.sds.lcs.mit.edu/lclint/
```

It includes this guide in HTML format, samples demonstrating LCLint, and links to related web sites.

LCLint can be downloaded from

```
http://www.sds.lcs.mit.edu/lclint/download.html
```

or obtained via anonymous `ftp` from

```
ftp://sds.lcs.mit.edu/pub/lclint/
```

Win32 and several UNIX platforms are supported. Source code is freely available.

Appendix B Communication

LCLint development is largely driven by suggestions and comments from users. We are also very interested in hearing about your experiences using LCLint in developing or maintaining programs, enforcing coding standards, or teaching courses. For general information, suggestions, and questions on LCLint send mail to `lclint@sds.lcs.mit.edu`.

To report a bug in LCLint send a message to `lclint-bug@sds.lcs.mit.edu`.

There are two mailing lists associated with LCLint:

`lclint-announce@sds.lcs.mit.edu`
: Reserved for announcements of new releases and bug fixes. (Everyone who sends mail regarding LCLint is added to this list.)

`lclint-interest@sds.lcs.mit.edu`
: Informal discussions on the use and development of LCLint. To subscribe, send a (human-readable) message to `lclint-request@sds.lcs.mit.edu` or use the form at `http://www.sds.mit.edu/lclint/lists.html`. The mailing list is archived at `http://www.sds.mit.edu/lclint/lclint-interest/`

LCLint discussions relating to checks enabled by specifications or annotations are welcome in the `comp.specification.larch` Usenet group. Messages more focused on C-specific checking would be more appropriate for the `lclint-interest` list of one of the C language groups.

Appendix C Flags

Flags can be grouped into four major categories:

- Global flags for controlling initializations and global behavior
- Message format flags for controlling how messages are displayed
- Mode selectors for coarse control of LCLint checking
- Checking flags that control checking and what classes of messages are reported.

Global flags can be used in initialization files and at the command line; all other flags may also be used in control comments.

Global Flags

Global flags can be set at the command line or in an options file, but cannot be set locally using stylized comments. These flags control on-line help, initialization files, pre-processor flags, libraries and output.

Help

On-line help provides documentation on LCLint operation and flags. When a help flag is used, no checking is done by LCLint. Help flags may be preceded by `-` or `+`.

`help`
Display general help overview, including list of additional help topics.

`help <topic>`
Display help on <*topic*>. Available topics:

`annotations`	describe annotations
`comments`	describe control comments
`flags`	describe flag categories
`flags` *<category>*	all flags pertaining to <category> (one of the categories listed by `lclint -help flags`)
`flags alpha`	all flags in alphabetical order
`flags full`	print a full description of all flags
`mail`	print information on mailing lists
`modes`	flags settings in modes
`prefixcodes`	character codes for setting namespace prefixes
`references`	print references to relevant papers and web sites
`vars`	describe environment variables
`version`	print maintainer and version information

`help <flag>`
Describe flag <*flag*>. (May list several flags.)

`warn-flags`
Display a warning when a flag is set in a surprising way. An error is reported if an obsolete (LCLint Version 1.4 or earlier) flag is set, a flag is set to its current value (i.e., the `+` or `-` may be wrong), or a mode selector flag is set after mode checking flags that will be reset by the mode were set. By default, `warn-flags` is on. To suppress flag warnings, use `-warn-flags`.

Initialization

These flags control directories and files used by LCLint. They may be used from the command line or in an options file, but may not be used as control comments in the source code. Except where noted. they have the same meaning preceded by `-` or `+`.

`tmpdir` *<directory>*
: Set directory for writing temp files. Default is `/tmp/`.

`I`*<directory>*
: Add directory to path searched for C include files. Note there is no space after the `I`, to be consistent with C preprocessor flags.

`S`*<directory>*
: Add directory to path search for `.lcl` specification files.

`f` *<file>*
: Load options file *<file>*. If this flag is used from the command line, the default `~/.lclintrc` file is not loaded. This flag may be used in an options file to load in another options file.

`nof`
: Prevents the default options files (`./.lclintrc` and `~/.lclintrc`) from being loaded. (Setting `-nof` overrides `+nof`, causing the options files to be loaded normally.)

`sys-dirs`
: Set directories for system files (default is `"/usr/include"`). Separate directories with colons (e.g., `"/usr/include:/usr/local/lib"`). Flag settings propagate to files in a system directory. If `-sys-dir-errors` is set, no errors are reported for files in system directories.

Pre-processor

These flags are used to define or undefine pre-processor constants. The `-I`*<directory>* flag is also passed to the C pre-processor.

`D<initializer>`
: Passed to the C pre-processor.

`U<initializer>`
: Passed to the C pre-processor.

Libraries

These flags control the creation and use of libraries.

`dump` *<file>*
: Save state in *<file>* for loading. The default extension `.lcd` is added if *<file>* has no extension.

`load` *<file>*
: Load state from *<file>* (created by `-dump`). The default extension `.lcd` is added if *<file>* has no extension. Only one library file may be loaded.

By default, the standard library is loaded if the `-load` flag is not used to load a user library. If no user library is loaded, one of the following flags may be used to select a different standard library. Precede the flag by `+` to load the described library (or to prevent a library from being loaded using `no-lib`). See Appendix F for information on the provided libraries.

`no-lib`
: Do not load any library. This prevents the standard library from being loaded.

`ansi-lib`

Use the ANSI standard library (selected by default).

`strict-lib`
Use strict version of the ANSI standard library.

`posix-lib`
Use the POSIX standard library.

`posix-strict-lib`
Use the strict version of the POSIX standard library.

`unix-lib`
Use UNIX version of standard library.

`unix-strict-lib`
Use the strict version of the UNIX standard library.

Output

These flags control what additional information LCLint prints. Setting `+`*`<flag>`* causes the described information to be printed; setting `-`*`<flag>`* prevents it. By default, all these flags are off.

`use-stderr`
Send error messages to standard error (instead of standard output).

`show-summary`
Show a summary of all errors reported and suppressed. Counts of suppressed errors are not necessarily correct since turning a flag off may prevent some checking from being done to save computation, and errors that are not reported may propagate differently from when they are reported.

`show-scan`
Show file names are they are processed.

`show-all-uses`
Show list of uses of all external identifiers sorted by number of uses.

`stats`
Display number of lines processed and checking time.

`time-dist`
Display distribution of where checking time is spent.

`quiet`
Suppress herald and error count. (If `quiet` is not set, LCLint prints out a herald with version information before checking begins, and a line summarizing the total number of errors reported.)

`which-lib`
Print out the standard library filename and creation information.

`limit` *`<number>`*
At most <*number*> similar errors are reported consecutively. Further errors are suppressed, and a message showing the number of suppressed messages is printed.

Expected Errors

Normally, LCLint will expect to report no errors. The exit status will be success (`0`) if no errors are reported, and failure if any errors are reported. Flags can be used to set the expected number of reported errors. Because of the provided error suppression mechanisms, these options should probably not be used for final checking real programs but may be useful in developing programs using make.

`expect` *`<number>`*
Exactly <*number*> code errors are expected. LCLint will exit with failure exit status unless <*number*> code errors are detected.

Message Format

These flags control how messages are printed. They may be set at the command line, in options files, or locally in syntactic comments. The `line-len` and `limit` flags may be preceded by `+` or `-` with the same meaning; for the other flags, `+` turns on the describe printing and `-` turns it off. The box to the left of each flag gives its default value.

`+` `show-column`
: Show column number where error is found.

`+` `show-func`
: Show name of function (or macro) definition containing error. The function name is printed once before the first message detected in that function.

`-` `show-all-conjs`
: Show all possible alternate types (see Section 8.2.2).

`-` `paren-file-format`
: Use <*file*>(<*line*>) format in messages.

`+` `hints`
: Provide hints describing an error and how a message may be suppressed for the first error reported in each error class.

`-` `force-hints`
: Provide hints for all errors reported, even if the hint has already been displayed for the same error class.

`80` `line-len` *<number>*
: Set length of maximum message line to <*number*> characters. LCLint will split messages longer than <*number*> characters long into multiple lines.

Mode Selector Flags

Mode selects flags set the mode checking flags to predefined values. They provide a quick coarse-grain way of controlling what classes of errors are reported. Specific checking flags may be set after a mode flag to override the mode settings. Mode flags may be used locally, however the mode settings will override specific command line flag settings. A warning is produced if a mode flag is used after a mode checking flag has been set.

These are brief descriptions to give a general idea of what each mode does. To see the complete flag settings in each mode, use `lclint -help modes`. A mode flag has the same effect when used with either `+` or `-`.

`weak`
: Weak checking, intended for typical unannotated C code. No modifies checking, macro checking, rep exposure, or clean interface checking is done. Return values of type `int` may be ignored. The types `bool`, `int`, `char` and user-defined `enum` types are all equivalent. Old style declarations are unreported.

`standard`
: The default mode. All checking done by `weak`, plus modifies checking, global alias checking, use all parameters, using released storage, ignored return values or any type, macro checking, unreachable code, infinite loops, and fall-through cases. The types `bool`, `int` and `char` are distinct. Old style declarations are reported.

`checks`
Moderately strict checking. All checking done by `standard`, plus must modification checking, rep exposure, return alias, memory management and complete interfaces.

`strict`
Absurdly strict checking. All checking done by `checks`, plus modifications and global variables used in unspecified functions, strict standard library, and strict typing of C operators. A special reward will be presented to the first person to produce a real program that produces no errors with `strict` checking.

Checking Flags

These flags control checking done by LCLint. They may be set locally using syntactic comments, from the command line, or in an options file. Some flags directly control whether a certain class of message is reported. Preceding the flag by `+` turns reporting on, and preceding the flag by `-` turns reporting off. Other flags control checking less directly by determining default values (what annotations are implicit), making types equivalent (to prevent certain type errors), controlling representation access, etc. For these flags, the effect of `+` is described, and the effect of `-` is the opposite (or explicitly explained if there is no clear opposite). The organization of this section mirrors Sections 3-10.

Key

To the left of each flag name is a flag descriptor encoding what kind of flag it is and its default value. The descriptions are:

plain: - A plain flag. The value after the colon gives the default setting (e.g., this flag is off.)

m:- -++ A mode checking flag. The value of the flag is set by the mode selector. The four signs give the setting in the weak, standard, checks and strict modes. (e.g., this flag is off in the weak and standard modes, and on in the checks and strict modes.)

shortcut A shortcut flag. This flag sets other flags, so it has no default value.

Types

Abstract Types

plain: - `imp-abstract`
Implicit `abstract` annotation for type declarations that do not use `concrete`.

m:-+++ `mut-rep`
Representation of mutable type has sharing semantics.

Access (Section 3.1)

plain: + `access-module`
An abstract type defined in *M*`.h` (or specified in *M*`.lcl`) is accessible in *M*`.c`.

plain: + `access-file`
An abstract type named *`type`* is accessible in files named *`type.<extension>`*.

plain: + `access-czech`
An abstract type named *`type`* may be accessible in a function named *`type_name`*. (see Section 9.1.1)

plain: - `access-slovak`
An abstract type named *`type`* may be accessible in a function named *`typeName`*. (see Section.9.1.2)

plain: - `access-czechoslovak`
An abstract type named *`type`* may be accessible in a function named *`type_name`* or *`typeName`*. (see Section 9.1.3)

shortcut `access-all`
Sets `access-module`, `access-file` and `access-czech`.

Boolean Types (Section 3.3)

These flags control the type name used to represent booleans, and whether the boolean type is abstract.

plain: - `bool`
Boolean type is an abstract type.

plain: *unset* `booltype` *`<name>`*
Set name of boolean type to <*name*>.

plain: FALSE `boolfalse` *`<name>`*
Set name of boolean false to <*name*>.

plain: TRUE `booltrue` *`<name>`*
Set name of boolean true to <*name*>.

Predicates

m:--++ `pred-bool-ptr`
Type of condition test is a pointer.

m:-+++ `pred-bool-int`
Type of condition test is an integral type.

m:++++ `pred-bool-others`
Type of condition test is not a boolean, pointer or integral type.

shortcut `pred-bool`
Sets `predboolint`, `predboolptr` and `preboolothers`.

plain: + `pred-assign`
The condition test is an assignment expression. If an assignment is intended, add an extra parentheses nesting (e.g., `if ((a = b))` ...).

Primitive Operations

m:---+ `ptr-arith`
Arithmetic involving pointer and integer.

m:++-- `ptr-negate`
Allow the operand of the `!` operator to be a pointer.

m:---+ `bitwise-signed`
An operand to a bitwise operator is not an unsigned value. This may have unexpected results depending on the signed representations.

m:-+++ `shift-signed`
The left operand to a shift operator is not an unsigned value.

m:---+ `strict-ops`
Primitive operation does not type check strictly.

m:---+ `sizeof-type`
Operand of `sizeof` operator is a type. (Safer to use expression, `int *x = sizeof (*x);` instead of `sizeof (int)`.)

Format Codes

plain: + `format-code`
Invalid format code in format string for `printflike` or `scanflike` function.

plain: + `format-type`
Type-mismatch in parameter corresponding to format code in a `printflike` or `scanflike` function.

Main

plain: + `main-type`
Type of `main` does not match expected type (function returning an `int`, taking no parameters or two parameters of type `int` and `char **`.)

Comparisons

m:-+++ `bool-compare`
Comparison between boolean values. This is dangerous since there may be multiple `TRUE` values if any non-zero value is interpreted as `TRUE`.

m:-+++ `real-compare`
Comparison involving `float` or `double` values. This is dangerous since it may produce unexpected results because floating point representations are inexact.

m:-+++ `ptr-compare`
Comparison between pointer and number.

Type Equivalence

m:+--- `void-abstract`
Allow `void *` to match pointers to abstract types. (Casting a pointer to an abstract type to a pointer to `void` is okay if `+void-abstract` is set.)

plain: + `cast-fcn-ptr`
A pointer to a function is cast to (or used as) a pointer to void (or vice versa).

m:+--- `forward-decl`
Forward declarations of pointers to abstract representation match abstract type.

m:-+++ `imp-type`
A variable declaration has no explicit type. The type is implicitly int.

plain: + `incomplete-type`
A formal parameter is declared with an incomplete type (e.g., `int[][]`).

m:+--- `char-index`
Allow `char` to index arrays.

m:---- `enum-index`
Allow members of `enum` type to index arrays.

m:+--- `bool-int`
Make `bool` and `int` are equivalent. (No type errors are reported when a boolean is used where an integral type is expected and vice versa.)

m:+--- `char-int`
Make `char` and `int` types equivalent

m:++-- `enum-int`
Make `enum` and `int` types equivalent

m:+--- `float-double`
Make `float` and `double` types equivalent

m:---- `ignore-quals`
Ignore type qualifiers (`long`, `short`, `unsigned`).

m:++-- `relax-quals`

Report qualifier mismatches only if dangerous (information may be lost since a larger type is assigned to (or passed as) a smaller one or a comparison uses `signed` and `unsigned` values.)

m:----
`ignore-signs`
Ignore signs in type comparisons (`unsigned` matches `signed`).

plain: -
`long-integral`
Allow long type to match an arbitrary integral type (e.g., `dev_t`).

m:+---
`long-unsigned-integral`
Allow unsigned long type to match an arbitrary integral type (e.g., `dev_t`).

plain: -
`match-any-integral`
Allow any integral type to match an arbitrary

plain: -
`long-unsigned-unsigned-integral`
Allow unsigned long type to match an arbitrary unsigned integral type (e.g., `size_t`).

m:+---
`long-signed-integral`
Allow long type to match an arbitrary signed integral type (e.g., `ssize_t`).

plain: +
`num-literal`
Integer literals can be used as floats.

plain: -
`char-int-literal`
A character constant may be used as an `int`.

plain: +
`zero-ptr`
Literal `0` may be used as a pointer.

plain: -
`relax-types`
Allow all numeric types to match.

Function Interfaces

Modification (Section 4.1)

plain: +
`modifies`
Undocumented modification of caller-visible state. Without `+moduncon`, modification errors are only reported in the definitions of functions declared with a modifies clause (or specified).

m:--++
`must-mod`
Documented modification is not detected. An object listed in the modifies clause for a function, is not modified by the implementation.

shortcut
`mod-uncon`
Report modification errors in functions declared without a modifies clause.(Sets `mod-nomods`, `mod-globs-nomods` and `mod-strict-globs-nomods`.)

m:---+
`mod-nomods`
Report modification errors (not involving global variables) in functions declared without a modifies clause.

m:---+
`mod-uncon-nomods`
An unconstrained function is called in a function body where modifications are checked. Since the unconstrained function may modify anything, there may be undetected modifications in the checked function.

m:---+
`mod-internal-strict`
A function that modifies `internalState` is called from a function that does not list `internalState` in its modifies clause.

m:---+
`mod-file-sys`
A function modifies the file system but does not list `fileSystem` in its modifies clause.

Global Variables (Section 4.2)

Errors involving the use and modification of global and file static variables are reported depending on flag settings, annotations where the global variable is declared, and whether or not the function where the global is used was declared with a globals clause.

plain: + `globs`
Undocumented use of a checked global variable in a function with a globals list.

m:++++ `glob-use`
A global listed in the globals list is not used in the implementation.

m:---+ `glob-noglobs`
Use of a checked global in a function with no globals list.

m:---+ `internal-globs`
Undocumented use of internal state (should have `globals internalState`).

m:---+ `internal-globs-noglobs`
Use of internal state in function with no globals list.

m:-+++ `glob-state`
A function returns with global in inconsistent state (null or undefined)

m:--++ `all-globs`
Report use and modification errors for globals not annotated with `unchecked`.

m:++++ `check-strict-globs`
Report use and modification errors for `checkedstrict` globals.

Modification of Global Variables

m:-+++ `mod-globs`
Undocumented modification of a checked global variable.

m:---+ `mod-globs-unchecked`
Undocumented modification of an `unchecked` global variable.

m:---+ `mod-globs-nomods`
Undocumented modification of a checked global variable in a function with no modifies clause.

m:---+ `mod-strict-globs-nomods`
Undocumented modification of a `checkedstrict` global variable in a function declared with no modifies clause.

Globals Lists and Modifies Clauses

m:---+ `warn-missing-globs`
Global variable used in modifies clause is not listed in globals list. (The global is added to the globals list.)

m:---+ `warn-missing-globs-noglobs`
Global variable used in modifies clause of a function with no globals list.

m:--++ `globs-imp-mods-nothing`
A function declared with a globals list but no modifies clause is assumed to modify nothing.

m:---- `mods-imp-noglobs`
A function declared with a modifies clause but no globals list is assumed to use no globals.

Implicit Checking Qualifiers

m:----
`imp-checked-globs`
Implicit `checked` qualifier on global variables with no checking annotation.

m:----
`imp-checked-statics`
Implicit `checked` qualifier file static scope variables with no checking annotation.

m:----
`imp-checkmod-globs`
Implicit `checkmod` qualifier on global variables with no checking annotation.

m:----
`imp-checkmod-statics`
Implicit `checkmod` qualifier file static scope variables with no checking annotation.

m:---+
`imp-checkedstrict-globs`
Implicit `checked` qualifier on global variables with no checking annotation.

m:---+
`imp-checkedstrict-statics`
Implicit `checked` qualifier file static scope variables with no checking annotation.

m:--++
`imp-checkmod-internals`
Implicit `checkmod` qualifier on function scope static variables with no checking annotation.

m:-+++
`imp-globs-weak`

Global Aliasing

shortcut
`glob-alias`
Function returns with global aliasing external state (sets `checkstrict-glob-alias`, `checked-glob-alias`, `checkmod-glob-alias` and `unchecked-glob-alias`).

m:-+++
`checkstrict-glob-alias`
Function returns with a `checkstrict` global aliasing external state.

m:-+++
`checked-glob-alias`
Function returns with a `checked` global aliasing external state.

m:-+++
`checkmod-glob-alias`
Function returns with a `checkmod` global aliasing external state.

m:--++
`unchecked-glob-alias`
Function returns with an `unchecked` global aliasing external state.

Declaration Consistency *(Section 4.3)*

m:-+++
`incon-defs`
Identifier redeclared or redefined with inconsistent type.

m:-+++
`incon-defs-lib`
Identifier defined in a library is redefined with inconsistent type

m:----
`overload`
Standard library function overloaded.

m:-+++
`match-fields`
A `struct` or `enum` type is redefined with inconsistent fields or members.

Memory Management

Reporting of memory management errors is controlled by flags setting checking and implicit annotations and code annotations.

Deallocation Errors (Section 5.2)

m:-+++ `use-released`
: Storage used after it may have been released.

m:---+ `strict-use-released`
: An array element used after it may have been released.

Inconsistent Branches

m:-+++ `branch-state`
: Storage has inconsistent states of alternate paths through a branch (e.g., it is released in the true branch of an if-statement, but there is no else branch.)

m:---+ `strict-branch-state`
: Storage through array fetch has inconsistent states of alternate paths through a branch. Since array elements are not checked accurately, this may lead to spurious errors.

m:--++ `dep-arrays`
: Treat array elements as `dependent` storage. Checking of array elements cannot be done accurately by LCLint. If `dep-arrays` is not set, array elements are assumed to be independent, so code that releases the same element more than once will produce no error. If `dep-arrays` is set, array elements are assumed to be dependent, so code that releases the same element more that once will produce an error, but so will code that releases different elements correctly will produce a spurious error.

Memory Leaks

m:-+++ `must-free`
: Allocated storage was not released before return or scope exit Errors are reported for `only`, `fresh` or `owned` storage.

m:-+++ `comp-destroy`
: All only references derivable from `out only` parameter of type `void *` must be released. (This is the type of the parameter to `free`, but may also be used for user-defined deallocation functions.)

m:---+ `strict-destroy`
: Report complete destruction errors for array elements that may have been released. (If `strict-destroy` is not set, LCLint will assume that if any array element was released, the entire array was correctly released.)

Transfer Errors

A transfer error is reported when storage is transferred (by an assignment, passing a parameter, or returning) in a way that is inconsistent.

shortcut `mem-trans`
Sets all memory transfer errors flags.

m:-+++ `only-trans`
Only storage transferred to non-only reference (memory leak).

m:-+++ `ownedtrans`
Owned storage transferred to non-owned reference (memory leak).

m:-+++ `fresh-trans`
Newly-allocated storage transferred to non-only reference (memory leak).

m:-+++ `shared-trans`
Shared storage transferred to non-shared reference

m:-+++ `dependent-trans`
Inconsistent `dependent` transfer. Dependent storage is transferred to a non-dependent reference.

m:-+++ `temp-trans`
Temporary storage (associated with a `temp` formal parameter) is transferred to a non-temporary reference. The storage may be released or new aliases created.

m:-+++ `kept-trans`
Kept storage transferred to non-temporary reference.

m:-+++ `keep-trans`
Keep storage is transferred in a way that may add a new alias to it, or release it.

m:-+++ `refcount-trans`
Reference counted storage is transferred in an inconsistent way.

m:-+++ `newref-trans`
A new reference transferred to a reference counted reference (reference count is not set correctly).

m:-+++ `immediate-trans`
An immediate address (result of `&`) is transferred inconsistently.

m:-+++ `static-trans`
Static storage is transferred in an inconsistent way.

m:-+++ `expose-trans`
Inconsistent exposure transfer. Exposed storage is transferred to a non-`exposed`, non-`observer` reference.

m:-+++ `observer-trans`
Inconsistent `observer` transfer. Observer storage is transferred to a non-observer reference.

m:-+++ `unqualified-trans`
Unqualified storage is transferred in an inconsistent way.

Initializers

m:--++ `only-unq-global-trans`
Only storage transferred to an unqualified global or static reference. This may lead to a memory leak, since the new reference is not necessarily released.

m:--++ `static-init-trans`
Static storage is used as an initial value in an inconsistent way.

m:--++ `unqualified-init-trans`
Unqualified storage is used as an initial value in an inconsistent way.

Derived Storage

m:-+++ `comp-mem-pass`

Storage derivable from a parameter does not match the alias kind expected for the formal parameter.

Stack References

m:++++ `stack-ref`
A stack reference is pointed to by an external reference when the function returns. Since the call frame will be destroyed when the function returns the return value will point to dead storage. (Section 5.2.6)

Implicit Memory Annotations (Section 5.3)

plain: + `glob-imp-only`
Assume unannotated global storage is only.

plain: + `param-imp-temp`
Assume unannotated parameter is `temp`.

plain: + `ret-imp-only`
Assume unannotated returned storage is `only`.

plain: + `struct-imp-only`
Assume unannotated structure or union field is `only`.

shortcut `code-imp-only`
Sets `glob-imp-only`, `ret-imp-only` and `struct-imp-only`.

m:-+++ `mem-imp`
Report memory errors for unqualified storage.

m:---- `pass-unknown`
Passing a value as an unannotated parameter clears its annotation. This will prevent many spurious errors from being report for unannotated programs, but eliminates the possibility of detecting many errors.

Sharing

Aliasing (Section 6)

m:-+++ `alias-unique`
An actual parameter that is passed as a `unique` formal parameter is aliased by another parameter or global variable.

m:-+++ `may-alias-unique`
An actual parameter that is passed as a `unique` formal parameter may be aliased by another parameter or global variable.

m:-+++ `must-not-alias`
An alias has been added to a `temp`-qualifier parameter or global that is visible externally when the function returns.

m:--++ `ret-alias`
A function returns an alias to parameter or global.

Exposure (Section 6.2)

shortcut `rep-expose`
The internal representation of an abstract type is visible to the caller. This means clients may have access to a pointer into the abstract representation. (Sets `assign-expose`, `ret-expose`, and `cast-expose`.)

m:`--++` `assign-expose`
Abstract representation is exposed by an assignment or passed parameter.

m:`--++` `cast-expose`
Abstract representation is exposed through a cast.

m:`--++` `ret-expose`
Abstract representation is exposed by a return value.

Observer Modifications

plain: `+` `mod-observer`
Possible modification of observer storage.

m:`---+` `mod-observer-uncon`
Storage declared with observer may be modified through a call to an unconstrained function.

String Literals (Section 6.2.1)

m:`--++` `read-only-trans`
Report memory transfer errors for initializations to read-only string literals

m:`-+++` `read-only-strings`
String literals are read-only (ANSI semantics). An error is reported if a string literal may be modified or released.

Use Before Definition (Section 7.1)

m:`-+++` `use-def`
The value of a location that may not be initialized on some execution path is used.

m:`----` `imp-outs`
Allow unannotated pointer parameters to functions to be implicit out parameters.

m:`-+++` `comp-def`
Storage derivable from a parameter, return value or global variable is not completely defined.

m:`-+++` `union-def`
No field of a union is defined. (No error is reported if at least one union field is defined.)

m:`-+++` `must-define`
Parameter declared with `out` is not defined before return or scope exit.

Null Pointers (Section 7.2)

m:`-+++` `null`
A possibly null pointer may be dereferenced, or used somewhere a non-null pointer is expected.

Macros (Section 8)

These flags control expansion and checking of macro definitions and invocations.

Macro Expansion

These flags control which macros are checked as functions or constants, and which are expanded in the pre-processing phase. Macros preceded by `/*@notfunction@*/` are never expanded regardless of these flag settings. These flags may be used in source-file control comments.

plain: - `fcn-macros`
: Macros defined with parameter lists are not expanded and are checked as functions.

plain: - `const-macros`
: Macros defined without parameter lists are not expanded and are checked as constants.

shortcut `all-macros`
: Sets `fcn-macros` and `const-macros`.

plain: - `lib-macros`
: Macros defining identifiers declared in a loaded library are not expanded and are checked according to the library information.

Macro Definitions

These flags control what errors are reported in macro definitions.

m:-+++ `macro-stmt`
: Macro definition is not syntactically equivalent to function. This means if the macro is used as a statement (e.g., `if (test) macro();`) unexpected behavior may result. One fix is to surround the macro body with `do { … } while (FALSE)`.

m:-+++ `macro-params`
: A macro parameter is not used exactly once in all possible invocations of the macro.

m:-+++ `macro-assign`
: A macro parameter is used as the left side of an assignment expression.

m:-+++ `macro-parens`
: A macro parameter is used without parentheses (in potentially dangerous context).

m:---+ `macro-empty`
: Macro definition of a function is empty.

m:-+++ `macro-redef`
: Macro is redefined. There is another macro defined with the same name.

m:-+++ `macro-unrecog`
: An unrecognized identifier appears in a macro definition. Since the identifier may be defined where the macro is used, this could be okay, but LCLint will not be able to check the unrecognized identifier appropriately.

Corresponding Declarations

m:++++ `macro-match-name`
An `iter` or `constant` macro is defined using a different name from the one used in the previous syntactic comment

shortcut `macro-decl`
A macro definition has no corresponding declaration. (Sets `macrofcndecl` and `macroconstdecl`.)

m:-+++ `macro-fcn-decl`
Macro definition with parameter list has no corresponding function prototype. Without a prototype, the types of the macro result and parameters are unknown.

m:-+++ `macro-const-decl`
A macro definition without parameter list has no corresponding constant declaration.

plain: + `next-line-macros`
A constant or iter declaration is not immediately followed by a macro definition.

Side-Effect Free Parameters (Section 8.2.1)

These flags control error reporting for parameters with inconsistent side-effects in invocations of checked function macros and function calls.

m:-+++ `sef-params`
An actual parameter with side-effects is passed as a formal parameter declared with `sef`.

m:--++ `sef-uncon`
An actual parameter involving a call to an unconstrained function (declared without modifies clause) that may modify anything is passed as a `sef` parameter.

Iterators

plain: - `has-yield`
An iterator has been declared with no parameters annotated with `yield`.

Naming Conventions

plain: + `name-checks`
Turns all name checking on or off without changing other settings.

Type-Based Naming Conventions (Section 9.1)

Czech Naming Convention

shortcut `czech`
Selects complete Czech naming convention (sets `access-czech`, `czech-fcns`, `czech-vars`, `czech-consts`, `czech-macros`, and `czech-types`).

plain: + `access-czech`
Allow access to abstract types following Czech naming convention. The representation of an abstract type named *t* is accessible in the definition of a function or constant named *t_name*.

plain: - `czech-fcns`
Function or iterator name is not consistent with Czech naming convention.

plain: - `czech-vars`
Variable name is not consistent with Czech naming convention.

plain: - `czech-macros`
Expanded macro name is not consistent with Czech naming convention.

plain: - `czech-consts`

Constant name is not consistent with Czech naming convention.

plain: - `czech-types`
Type name is not consistent with Czech naming convention. Czech type names must not use the underscore character.

Slovak Naming Convention

shortcut `slovak`
Selects complete Slovak naming convention (sets `access-slovak`, `slovak-fcns`, `slovak-vars`, `slovak-consts`, `slovak-macros`, and `slovak-types`).

plain: - `access-slovak`
Allow access to abstract types following Slovak naming convention. The representation of an abstract type named *t* is accessible in the definition of a function or constant named *tName*.

plain: - `slovak-fcns`
Function or iterator name is not consistent with Slovak naming convention.

plain: - `slovak-macros`
Expanded macro name is not consistent with Slovak naming convention.

plain: - `slovak-vars`
Variable name is not consistent with Slovak naming convention.

plain: - `slovak-consts`
Constant name is not consistent with Slovak naming convention.

plain: - `slovak-types`
Type name is not consistent with Slovak naming convention. Slovak type names may not include uppercase letters.

Czechoslovak Naming Convention

shortcut `czechoslovak`
Selects complete Czechoslovak naming convention (sets `access-czechoslovak`, `czechoslovak-fcns`, `czechoslovak-vars`, `czechoslovak-consts`, `czechoslovak-macros`, and `czechoslovak-types`).

plain: - `access-czechoslovak`
Allow access to abstract types by Czechoslovak naming convention. The representation of an abstract type named *t* is accessible in the definition of a function or constant named *t_name* or *tName*.

plain: - `czechoslovak-fcns`
Function name is not consistent with Czechoslovak naming convention.

plain: - `czechoslovak-macros`
Expanded macro name is not consistent with Czechoslovak naming convention.

plain: - `czechoslovak-vars`
Variable name is not consistent with Czechoslovak naming convention.

plain: - `czechoslovak-consts`
Constant name is not consistent with Czechoslovak naming convention.

plain: - `czechoslovak-types`
Type name is not consistent with Czechoslovak naming convention. Czechoslovak type names may not include uppercase letters or the underscore character.

Namespace Prefixes (Section 9.2)

`macro-var-prefix` *<prefix string>*
Set namespace prefix for variables declared in a macro body. (Default is `m_`.)

plain: + `macro-var-prefix-exclude`
A variable declared outside a macro body starts with the `macro-var-prefix`.

`tag-prefix` *<prefix string>*
Set namespace prefix of `struct`, `union` or `enum` tag identifiers.

plain: - `tag-prefix-exclude`

An identifier that is not a tag starts with the `tagprefix`.

`enum-prefix` *`<prefix string>`*

Set namespace prefix for `enum` members.

plain: - `enum-prefix-exclude`

An identifier that is not an `enum` member starts with the `enumprefix`.

`file-static-prefix` *`<prefix string>`*

Set namespace prefix for file `static` declarations.

plain: - `file-static-prefix-exclude`

An identifier that is not file static starts with the `filestaticprefix`.

`global-prefix` *`<prefix string>`*

Set namespace prefix for global variables.

plain: - `global-prefix-exclude`

An identifier that is not a global variable starts with the `globalprefix`.

`type-prefix` *`<prefix string>`*

Set namespace prefix for user-defined types.

plain: - `type-prefix-exclude`

An identifier that is not a type name starts with the `typeprefix`.

`external-prefix` *`<prefix string>`*

Set namespace prefix for external identifiers.

plain: - `external-prefix-exclude`

An identifier that is not external starts with the `externalprefix`.

`local-prefix` *`<prefix string>`*

Set namespace prefix for local variables.

plain: - `local-prefix-exclude`

An identifier that is not a local variable starts with the `localprefix`.

`unchecked-macro-prefix` *`<prefix string>`*

Set namespace prefix for unchecked macros.

plain: - `unchecked-macro-prefix-exclude`

An identifier that is not the name of an unchecked macro starts with the `uncheckedmacroprefix`.

`const-prefix` *`<prefix string>`*

Set namespace prefix for constants.

plain: - `const-prefix-exclude`

An identifier that is not a constant starts with the `constantprefix`.

`iter-prefix` *`<prefix string>`*

Set namespace prefix for iterators.

plain: - `iter-prefix-exclude`

An identifier that is not an `iter` starts with the `iterprefix`.

`proto-param-prefix` *<prefix string>*
Set namespace prefix for parameters in function prototypes.

plain: - `proto-param-prefix-exclude`
An identifier that is not a parameter in a function prototype starts with the `protoprarmprefix`.

m:--++ `proto-param-name`
A parameter in a function prototype has a name (can interfere with macro definitions).

m:---+ `proto-param-match`
The name of a parameter in a function definition does not match the corresponding name of the parameter in a function prototype (after removing the `protoparamprefix`).

Naming Restrictions (Section 9.3)

m:-+++ `shadow`
Declaration reuses name visible in outer scope.

Reserved Names

m:--++ `ansi-reserved`
External name conflicts with name reserved for the compiler or standard library.

m:---+ `ansi-reserved-internal`
Internal name conflicts with name reserved for the compiler or standard library.

m:--++ `cpp-names`
Internal or external name conflicts with a C++ reserved word. (Will cause problems if program is compiled with a C++ compiler.)

Distinct External Names

plain: - `distinct-external-names`
An external name is not distinguishable from another external name using `externalnamelen` significant characters.

plain: 6 `external-name-len` *<number>*
Sets the number of significant characters in an external name (ANSI default minimum is 6). Sets `+distinct-external-names`.

plain: - `external-name-case-insensitive`
Make alphabetic case insignificant in external names. According to ANSI standard, case need not be significant in an external name. If `+distinct-external-names` is not set, sets `+distinct-external-names` with unlimited external name length.

Distinct Internal Names

m:---- `distinct-internal-names`
An internal name is not distinguishable from another internal name using `internalnamelen` significant characters. (Also effected by `internal-name-case-insensitive` and `internal-name-lookalike`.)

plain: 31 `internal-name-len` *<number>*
Set the number of significant characters in an internal name. Sets `+distinct-internal-names`.

plain: - `internal-name-case-insensitive`
Set whether case is significant an internal names (`-internal-name-case-insensitive` means case is significant). If `+distinct-internal-names` is not set, sets `+distinct-internal-names` with unlimited internal name length.

plain: - `internal-name-lookalike`
Set whether similar looking characters (e.g., "`1`" and "`l`") match in internal names.

Other Checks

Undefined Evaluation Order (Section 10.1)

m:-+++ `eval-order`
Behavior of an expression is unspecified or implementation-dependent because sub-expressions contain interfering side effects that may be evaluated in any order.

m:---+ `eval-order-uncon`
An expression may be undefined because a sub-expression contains a call to an unconstrained function (no modifies clause) that may modify something that may be modified or used by another sub-expression.

Problematic Control Structures (Section 10.2)

m:-+++ `inf-loops`
Likely infinite loop is detected (Section 10.2.1).

m:--++ `inf-loops-uncon`
Likely infinite loop is detected. Loop test or body calls an unconstrained function that may produce an undetected modification.

m:---+ `elseif-complete`
There is no finals else following an else if construct (Section 10.2.5).

m:-+++ `case-break`
There is a non-empty case in a switch not followed by a `break` (Section 10.2.2).

m:-+++ `miss-case`
A switch on an `enum` type is missing a case for a member of the enumerator.

m:---- `loop-exec`
Assume all loops execute at least once. This effects use-before-definition and memory checking. It should probably not be used globally, but may be used surrounding a particular loop that is known to always execute to prevent spurious messages.

Deep Break (Section 10.2.3)

shortcut `deep-break`
Report errors for `break` statements inside a nested `while`, `for` or `switch`. (Sets all nested break and continue flags.)

m:--++ `loop-loop-break`
There is a `break` inside a `while`, `for` or iterator loop that is inside a `while`, `for` or iterator loop. Mark with `/*@innerbreak@*/` to suppress the message.

m:--++ `switch-loop-break`
There is a `break` inside a `while`, `for` or iterator loop that is inside a `switch` statement. Mark with `/*@loopbreak@*/`.

m:---+ `loop-switch-break`
There is a `break` inside a `switch` statement that is inside a `while`, `for` or iterator loop. Mark with `/*@switchbreak@*/`.

m:---+ `switch-switch-break`
There is a `break` inside a `switch` statement that is inside another `switch` statement. Mark with `/*@innerbreak@*/`.

m:---+ `loop-loop-continue`
There is a `continue` inside a while, for or iterator loop that is inside a while, for or iterator loop. Mark with `/*@innercontinue@*/`.

Loop and if Bodies (Section 10.2.4)

shortcut `all-empty`
An if, while or for statement has no body (sets `if-empty`, `while-empty` and `for-empty`.)

shortcut `all-block`
The body of an `if`, `while` or `for` statement is not a block (sets `if-block`, `while-block` and `for-block`.)

m:--++ `while-empty`
A while statement has no body.

m:---+ `while-block`
The body of a `while` statement is not a block

m:---+ `for-empty`
A `for` statement has no body.

m:---+ `for-block`
The body of a `for` statement is not a block.

m:++++ `if-empty`
An if statement has no body.

m:---+ `ifblock`
The body of an `if` statement is not a block.

Suspicious Statements (Section 10.3)

m:-+++ `unreachable`
Code is not reached on any possible execution.

m:-+++ `noeffect`
Statement has no effect.

m:---+ `noeffect-uncon`
Statement involving call to unconstrained function may have no effect.

m:-+++ `noret`
There is a path with no `return` in a function declared to return a non-`void` value.

Ignored Return Values (Section 10.3.2)

These flags control when errors are reported for function calls that do not use the return value. Casting the function call to `void` or declaring the called function to return `/*@alt void@*/`.

m:-+++ `ret-val-bool`
: Return value of type `bool` ignored.

m:-+++ `ret-val-int`
: Return value of type `int` ignored.

m:++++ `ret-val-other`
: Return value of type other than `bool` or `int` ignored.

shortcut `ret-val`
: Return value ignored (Sets `retvalbool`, `retvalint`, `retvalother`.)

Unused Declarations (Section 10.4)

These flags control when errors are reported for declarations that are never used. The `unused` annotation can be used to prevent unused errors from being report for a particular declaration.

m:---+ `top-use`
: An external declaration is not used in any file.

m:-+++ `const-use`
: Constant never used.

m:-+++ `enum-mem-use`
: Member of enumerator never used.

m:++++ `var-use`
: Variable never used.

m:-+++ `param-use`
: Function parameter never used.

m:++++ `fcn-use`
: Function is never used.

m:++++ `type-use`
: Defined type never used.

m:-+++ `field-use`
: Field of structure or union type is never used.

m:---+ `unused-special`
: Declaration in a special file (corresponding to `.l` or `.y` file) is unused.

Complete Programs (Section 10.5)

m:--++ `decl-undef`
: Function, variable, iterator or constant declared but never defined.

shortcut `partial`
: Check as partial system (sets `-decl-undef`, `-export-local` and prevents checking of macros in headers without corresponding `.c` files.)

Exports

m:---+ `export-local`
: A declaration is exported but not used outside this module. (Declaration can use the `static` qualifier.)

m:--++ `export-header`
: A declaration (other than a variable) is exported but does not appear in a header file.

m:--++ `export-header-var`
: A variable declaration is exported but does not appear in a header file.

Unrecognized Identifiers

plain: + `unrecog`
An unrecognized identifier is used.

plain: + `sys-unrecog`
Report unrecognized identifiers that start with the system prefix, __ (two underscores).

plain: - `repeat-unrecog`
Report multiple messages for unrecognized identifiers. If `repeatunrecog` is not set, an error is reported only the first time a particular unrecognized identifier appears in the file.

Multiple Definition and Declarations

plain: + `redef`
A function or variable is defined more than once.

m:--++ `redecl`
An identifier is declared more than once.

m:-+++ `nested-extern`
An `extern` declaration is used inside a function body.

ANSI C Conformance

m:--++ `noparams`
A function is declared without a parameter list prototype.

m:---+ `old-style`
Function definition is in old style syntax. Standard prototype syntax is preferred.

m:-+++ `exit-arg`
Argument to `exit` has implementation defined behavior. The only valid arguments to `exit` are `EXIT_SUCCESS`, `EXIT_FAILURE` and `0`. An error is reported if LCLint can detect statically that the argument to `exit` is not one of these.

plain: + `use-var-args`
Report if `<varargs.h>` is used (should use `stdarg.h`).

Limits (Section 10.6)

shortcut `ansi-limits`
Check for violations of standard limits (Sets `control-nest-depth`, `string-literal-len`, `include-nest`, `num-struct-fields`, and `num-enum-members`).

m:---+ 15 `control-nest-depth` *<number>*
Set maximum nesting depth of compound statements, iteration control structures, and selection control structures (ANSI minimum is 15).

m:---+ 509 `string-literal-len` *<number>*
Set maximum length of string literals (ANSI minimum is 509).

m:---+ 127 `num-struct-fields` *<number>*
Set maximum number of fields in a `struct` or `union` (ANSI minimum is 127).

m:---+ 127 `num-enum-members` *<number>*
Set maximum number of members of an `enum` type (ANSI minimum is 127).

m:--++ 8 `include-nest` *<number>*
Set maximum number of nested `#include` files (ANSI minimum is 8).

Header Inclusion (Appendix F)

plain: + `skip-ansi-headers`
Prevent inclusion of header files in a system directory with names that match standard ANSI headers. The symbolic information in the standard library is used instead. In effect only if a library that includes the ANSI library is used. The ANSI headers are: `assert`, `ctype`, `errno`, `float`, `limits`, `locale`, `math`, `setjmp`, `signal`, `stdarg`, `stddef`, `stdio`, `stdlib`, `strings`, `string`, `time`, and `wchar`.

plain: + `skip-posix-headers`

Prevent inclusion of header files in a system directory with names that match standard POSIX headers. The symbolic information in the standard library is used instead. In effect only if a library that includes the POSIX library is used. The POSIX headers are: `dirent`, `fcntl`, `grp`, `pwd`, `termios`, `sys/stat`, `sys/times`, `sys/types`, `sys/utsname`, `sys/wait`, `unistd`, and `utime`.

plain: + `warn-posix-headers`
Report use of a POSIX header when checking a program with a non-POSIX library.

plain: - `skip-sys-headers`
Prevent inclusion of all header files in system directories.

plain: + `sys-dir-expand-macros`
Expand macros in system directories regardless of other settings, except for macros corresponding to names defined in a load library.

m:---+ `sys-dir-errors`
Report errors in files in system directories (set by `-sys-dirs`).

global: - `single-include`
Optimize header inclusion to only include each header file once.

global: - `never-include`
Use library information instead of including header files.

Comments

These flags control how syntactic comments are interpreted (see Appendix E).

plain: @ `comment-char` *<char>*
Set the marker character for syntactic comments. Comments beginning with `/*`*<char>* are interpreted by LCLint.

plain: - `noaccess`
Ignore access comments.

plain: - `nocomments`
Ignore all stylized comments.

plain: + `sup-counts`
Actual number of errors does not match number in `/*@i`*<n>*`@*/`

plain: + `lint-comments`
Interpret traditional lint comments (`/*FALLTHROUGH*/`, `/*NOTREACHED*/`, `/*PRINTFLIKE*/`).

m:-+++ `warn-lint-comments`
Print a warning and suggest an alternative when a traditional lint comment is used.

plain: + `unrecog-comments`
Stylized comment is unrecognized.

Parsing

plain: - `continue-comment`
A line continuation marker (\) appears inside a comment on the same line as the comment close. Preprocessors should handle this correctly, but it causes problems for some preprocessors.

plain: + `nest-comment`
A comment open sequence (`/*`) appears inside a comment. This usually indicates that an earlier comment was not closed.

plain: + `duplicate-quals`
Report duplicate type qualifiers (e.g., `long long`). Duplicate type qualifiers not supported by ANSI, but some compilers (e.g., `gcc`) do support duplicate qualifiers.

plain: + `gnu-extensions`
Support some `GNU` (`gcc`) and Microsoft language extensions.

Array Formal Parameters

These flags control reporting of common errors caused by confusion about the semantics of array formal parameters.

plain: + `sizeof-formal-array`
The `sizeof` operator is used on a parameter declared as an array. (In many instances this has unexpected behavior, since the result is the size of a pointer to the element type, not the number of elements in the array.)

plain: + `fixed-formal-array`
An array formal parameter is declared with a fixed size (e.g., `int x[20]`). This is likely to be confusing, since the size is ignored.

plain: - `formal-array`
A formal parameter is declared as an array. This is probably not a problem, but can be confusing since it is treated as a pointer.

General Checks

These flags should probably not be set globally since they turn off general checks that should always be done. They may be used locally to suppress spurious errors.

plain: + `abstract`
A data abstraction barrier is violated.

plain: + `control`
A control flow error is detected.

plain: + `syntax`
Parse error.

plain: - `try-to-recover`
Try to recover from a parse error. If `trytorecover` is not set, LCLint will abort checking after a parse error is detected. If it is set, LCLint will attempt to recover, but LCLint does performs only minimal error recovery. It is likely that trying to recover after a parse error will lead to an internal assertion failing.

plain: + `type`
Type mismatch.

Flag Name Abbreviations

Within a flag name, abbreviations may be used. Table 2 shows the flag name abbreviations. The expanded and short forms are interchangeable in flag names.

For example, `globsimpmodsnothing` and `globalsimpliesmodifiesnothing` denote the same flag. Abbreviations in flag names allow pronounceable, descriptive names to be used without making flag names excessively long (although one must admit even `globsimpmodsnothing` is a bit of a mouthful.)

To make flag names more readable, the space, dash (`-`), and underscore (_) characters may be used inside a flag name. So, `globals-implies-modifies-nothing`, `glob_imps_modsnothing` and `globsimpmodsnothing` are equivalent.

Expanded Form	Short Form
`constant`	`const`
`declaration`	`decl`
`function`	`fcn`
`global`	`glob`
`implicit`, `implied`	`imp`
`iterator`	`iter`
`length`	`len`
`modifies`	`mods`
`modify`	`mod`
`memory`	`mem`
`parameter`	`param`
`pointer`	`ptr`
`return`	`ret`
`variable`	`var`
`unconstrained`, `unconst`	`uncon`

Table 2. Flag name abbreviations.

Copyright © 1996 United Feature Syndicate, Inc.
Redistribution in whole or in part prohibited

Appendix D Annotations

The grammar below is the C syntax from [K&R,A13] modified to show the syntax of syntactic comments. Only productions effected by LCLint annotations are shown. In the annotations, the `@` represents the comment marker char, set by `-commentchar` (default is `@`).

Functions

direct-declarator:
direct-declarator `(` *parameter-type-list*$_{opt}$ `)` *specials*$_{opt}$ *globals*$_{opt}$ *modifies*$_{opt}$
| *direct-declarator* `(` *identifier-list*$_{opt}$ `)` *specials*$_{opt}$ *globals*$_{opt}$ *modifies*$_{opt}$

specials: (Section 7.4)
`/*@`*special-tag specitem,*$^{+}$ `;`$_{opt}$ `@*/`

special-tag: `uses` | `sets` | `defines` | `allocates` | `releases` | *state-tag*`:`*state-clause*

state-tag: `pre` | `post`

state-clause: `only` | `shared` | `owned` | `dependent` | `observer` | `exposed`
| `isnull` | `notnull`

globals: (Section 4.2)
`/*@globals` *globitem,*$^{+}$ `;`$_{opt}$ `@*/`
| `/*@globals` *declaration-list*$_{opt}$ `;`$_{opt}$ `@*/`

globitem:
globannot* *identifier*
| `internalState`
| `fileSystem`

globannot: `undef` | `killed`

modifies: (Section 4.1)
`/*@modifies` *moditem,*$^{+}$ `;`$_{opt}$ `@*/`
| `/*@modifies nothing ;`$_{opt}$ `@*/`
| `/*@*/` *(Abbreviation for no globals and modifies nothing.)*

moditem:
expression
| `internalState`
| `fileSystem`

Iterators (Section 8.4)

The globals and modifies clauses for an iterator are the same as those for a function, except they are not enclosed by a comment, since the iterator is already a comment.

direct-declarator:
`/*@iter` *identifier* `(` *parameter-type-list*$_{opt}$ `)` *iter-globals*$_{opt}$ *iter-modifies*$_{opt}$ `@*/`

iter-globals:
`globals` *declaration-list*$_{opt}$ `;`$_{opt}$

iter-modifies:
`modifies` *moditem*,$^{+}$ `;`$_{opt}$
| `modifies nothing ;`$_{opt}$

Constants (Section 8.1)

external-declaration:
`/*@constant` *declaration* `;`$_{opt}$ `@*/`

Alternate Types (Section 8.2.2)

Alternate types may be used in the type specification of parameters and return values.

extended-type:
type-specifier alt-type$_{opt}$

alt-type:
`/*@alt` *basic-type*,$^{+}$ `@*/`

Declarator Annotations

General annotations appear after *storage-class-specifier*s and before *type-specifier*s. Multiple annotations may be used in any order. Here, annotations are without the surrounding comment. In a declaration, the annotation would be surrounded by `/*@` and `@*/`. In a globals or modifies clause or iterator or constant declaration, no surrounding comments would be used since they are within a comment.

Type Definitions (Section 3)

A type definition may use any either `abstract` or `concrete`, either `mutable` or `immutable`, and `refcounted`. Only a pointer to a `struct` may be declared with `refcounted`. Mutability annotations may not be used with concrete types since concrete types inherit their mutability from the actual type.

`abstract`
Type is abstract (representation is hidden from clients).
`concrete`
Type is concrete (representation is visible to clients).

`immutable`
Instances of the type cannot change value. (Section 3.2)
`mutable`
Instances of the type can change value. (Section 3.2)
`refcounted`
Reference counted type. (Section 5.4)

Global Variables (Section 4.2.1)

One check annotation may be used on a global or file-static variable declaration.

`unchecked`
Weakest checking for global use.
`checkmod`
Check modification by not use of global.
`checked`
Check use and modification of global.
`checkedstrict`
Check use of global, even in functions with no global list.

Memory Management (Section 1)

`dependent`
A reference to externally-owned storage. (Section 5.2.2)
`keep`
A parameter that is kept by the called function. The caller may use the storage after the call, but the called function is responsible for making sure it is deallocated. (Section 5.2.4)
`killref`
A `refcounted` parameter. This reference is killed by the call. (Section 5.4)
`only`
An unshared reference. Associated memory must be released before reference is lost. (Section 5.2)
`owned`
Storage may be shared by dependent references, but associated memory must be released before this reference is lost. (Section 5.2.2)
`shared`
Shared reference that is never deallocated. (Section 5.2.5)
`temp`
A temporary parameter. May not be released, and new aliases to it may not be created. (Section 5.2.2)

Aliasing (Section 6)

Both alias annotations may be used on a parameter declaration.

`unique`
Parameter that may not be aliased by any other reference visible to the function. (Section 6.1.1)
`returned`
Parameter that may be aliased by the return value. (Section 6.1.2)

Exposure (Section 6.2)

`observer`
Reference that cannot be modified. (Section 6.2.1)

`exposed`
Exposed reference to storage in another object. (Section 6.2)

Definition State (Section 7.1)

`out`
Storage reachable from reference need not be defined.

`in`
All storage reachable from reference must be defined.

`partial`
Partially defined. A structure may have undefined fields. No errors reported when fields are used.

`reldef`
Relax definition checking. No errors when reference is not defined, or when it is used.

Global State (Section 7.1.2)

These annotations may only be used in globals lists. Both annotations may be used for the same variable, to mean the variable is undefined before and after the call.

`undef`
Variable is undefined before the call.

`killed`
Variable is undefined after the call.

Null State (Section 7.2)

`null`
Possibly null pointer.

`notnull`
Non-null pointer.

`relnull`
Relax null checking. No errors when `NULL` is assigned to it, or when it is used as a non-null pointer.

Null Predicates (Section 7.2.1)

A null predicate annotation may be used of the return value of a function returning a boolean type, taking a possibly-null pointer for its first argument.

`truenull`
If result is `TRUE`, first parameter is `NULL`.

`falsenull`
If result is `TRUE`, first parameter is not `NULL`.

Execution (Section 7.3)

The `exits`, `mayexit` and `neverexits` annotations may be used on any function. The `trueexit` and `falseexit` annotations may only be used on functions whose first argument is a boolean.

`exits`
: Function never returns.

`mayexit`
: Function may or may not return.

`trueexit`
: Function does not return if first parameter is `TRUE`.

`falseexit`
: Function does not return if first parameter if `FALSE`.

`neverexit`
: Function always returns.

Side-Effects (Section 8.2.1)

`sef`
: Corresponding actual parameter has no side effects.

Declaration

These annotations can be used on a declaration to control unused or undefined error reporting.

`unused`
: Identifier need not be used (no unused errors reported.) (Section 10.4)

`external`
: Identifier is defined externally (no undefined error reported.) (Section 10.5)

Case

`fallthrough`
: Fall-through case. No message is reported if the previous case may fall-through into the one immediately after the `fallthrough`.

Break (Section 10.2.3)

These annotations are used before a `break` or `continue` statement.

`innerbreak`
: Break is breaking an inner loop or switch.

`loopbreak`
: Break is breaking a loop.

`switchbreak`
: Break is breaking a switch.

`innercontinue`
: Continue is continuing an inner loop.

Unreachable Code

This annotation is used before a statement to prevent unreachable code errors.

`notreached`
: Statement may be unreachable.

Special Functions (Appendix E)

These annotations are used immediately before a function declaration.

`printflike`
: Check variable arguments like `printf` library function.

`scanflike`
: Check variable arguments like `scanf` library function.

Appendix E Control Comments

Error Suppression

Several comments are provided for suppressing messages. In general, it is usually better to use specific flags to suppress a particular error permanently, but the general error suppression flags may be more convenient for quickly suppressing messages for code that will be corrected or documented later.

`ignore`
`end`

No errors will be reported in code regions between `/*@ignore@*/` and `/*@end@*/`. These comments can be used to easily suppress an unlimited number of messages, but are dangerous since if real errors are introduced in the `ignore`...`end` region they will not be reported. The `ignore` and `end` comments must be matched — a warning is printed if the file ends in an ignore region or if `ignore` is used inside ignore region.

`i`

No errors will be reported from an `/*@i@*/` comment to the end of the line.

`i`*`<n>`*

No errors will be reported from an `/*@i`*`<n>`*`@*/` (e.g., `/*@i3@*/`) comment to the end of the line. If there are not exactly *n* errors suppressed from the comment point to the end of the line, LCLint will report an error. This is more robust than `i` or `ignore` since a message is generated if the expected number errors is not present. Since errors are not necessarily detected until after this file is processed (for example, and unused variable error), suppress count errors are reported after all files have been processed. The `-supcounts` flag may be used to suppress these errors. This is useful when a system if being rechecked with different flag settings.

`t`
`t`*`<n>`*

Like `i` and `i`*`<n>`*, except controlled by `+tmpcomments` flag. These can be used to temporarily suppress certain errors. Then, `-tmpcomments` can be set to find them again.

Type Access

Control comments may also be used to override type access settings. The syntax `/*@access` *`<type>`*`,`$^{+}$`@*/` allows the following code to access the representation of *`<type>`*. Similarly, `/*@noaccess` *`<type>`*`,`$^{+}$`@*/` restricts access to the representation of *`<type>`*. The type in a `noaccess` comment must have been declared as an abstract type. Type access applies from the point of the comment to the end of the file or the next access control comment for this type.

Macro Expansion

The `/*@notfunction@*/`indicates that the next macro definition is not intended to be a function, and should be expanded in line instead of checked as a macro function definition.

Special Types

These syntactic comments are used to represent arbitrary integral types. Syntactically, they replace the implicit `int` type.

`/*@integraltype@*/`
: An arbitrary integral type. The actual type may be any one of `short`, `int`, `long`, `unsigned short`, `unsigned`, or `unsigned long`.

`/*@unsignedintegraltype@*/`
: An arbitrary unsigned integral type. The actual type may be any one of `unsigned short`, `unsigned`, or `unsigned long`.

`/*@signedintegraltype@*/`
: An arbitrary signed integral type. The actual type may be any one of `short`, `int`, or `long`.

Traditional Lint Comments

Some of the control comments supported by most standard UNIX lints are supported by LCLint so legacy systems can be checked more easily. These comments are not lexically consistent with LCLint comments, and their meanings are less precise (and may vary between different lint programs), so we recommend that LCLint comments are used instead except for checking legacy systems already containing standard lint comments.

These standard lint comments supported by LCLint:

`/*FALLTHROUGH*/` (alternate misspelling, `/*FALLTHRU*/`)
: Prevents errors for fall-through cases. Same meaning as `/*@fallthrough@*/`.

`/*NOTREACHED*/`
: Prevents errors about unreachable code (until the end of the function). Same meaning as `/*@notreached@*/`.

`/*PRINTFLIKE*/`
: Arguments similar to the `printf` library function (there didn't seem to be much of a consensus among standard lints as to exactly what this means). LCLint supports:

  `/*@printflike@*/`
  : Function takes zero or more arguments of any type, an unmodified `char *` format string argument and zero of more arguments of type and number dictated by the format string. Format codes are interpreted identically to the `printf` standard library function. May return a result of any type. (LCLint interprets `/*PRINTFLIKE*/` as `/*@printflike@*/`.)

  `/*@scanflike@*/`
  : Like `printflike`, except format codes are interpreted as in the `scanf` library function.

`/*ARGSUSED*/`
: Turns off unused parameter messages for this function. The control comment, `/*@-paramuse@*/` can be used to the same effect, or `/*@unused@*/` can be used in individual parameter declarations.

LCLint will ignore standard lint comments if `-lint-comments` is used. If `+warn-lint-comments` is used, LCLint generates a message for standard lint comments and suggest replacements.

Appendix F Libraries

Libraries can be used to record interface information. A library containing information about the Standard C Library is used to enable checking of library calls. Program libraries can be created to enable fast checking of single modules in a large program.

Standard Libraries

In order to check calls to library functions, LCLint uses an annotated standard library. This contains more information about function interfaces then is available in the system header files since it uses annotations. Further, it contains only those functions documented in the ANSI Standard. Many systems include extra functions in their system libraries; programs that use these functions cannot be compiled on other systems that do not provide them. Certain types defined by the library are treated as abstract types (e.g., a program should not rely on how the `FILE` type is implemented). When checking source code, LCLint does include system headers corresponding to files in the library, but instead uses the library description of the standard library.

The LCLint distribution includes several different standard libraries: the ANSI standard library, the POSIX standard library[28], and an ad hoc UNIX library. Each library comes in two versions: the standard version and the strict version.

ANSI Library

The default behavior of LCLint is to use the ANSI standard library (loaded from `ansi.lcd`). This library is based on the standard library described in the ANSI C Standard. It includes functions and types added by Amendment 1 to the ANSI C Standard.

POSIX Library

The POSIX library is selected by the `+posixlib` flag. The POSIX library is based on the IEEE Std 1003.1-1990.

UNIX Library

The UNIX library is selected by the `+unixlib` flag. This library is an ad hoc attempt to capture additional functionality provided by many UNIX platforms. Unfortunately, UNIX systems vary widely and very few are consistent with the ANSI Standard.

The differences between the standard library and the POSIX library are:

- In the UNIX library, `free` is declared with a non-null parameter. ANSI C specifies that `free` should handle the argument `NULL`, but several UNIX platforms crash if `NULL` is passed to `free`.
- Extra variables, constants and functions are included in the UNIX library. Some declarations are not part of the POSIX library, but are believed to be available on many UNIX systems. See `lib/unix.h` for a list of the UNIX-only declarations.

Code checked using the UNIX library can probably be ported to some UNIX systems without difficulty. To enhance the likelihood that a program is portable, the POSIX library should be used instead.

[28] POSIX library was contributed by Jens Schweikhardt.

Strict Libraries

Stricter versions of the libraries are used is the `-ansi-strict`, `posix-strict-lib` or `unix-strct-lib` flag is used. These libraries use a stricter interpretation of the library. They will detect more errors in some programs, but may to produce many spurious errors for typical code.

The differences between the standard libraries and the strict libraries are:

- The standard libraries declare the printing functions (`fprintf`, `printf`, and `sprintf`) that may return error codes to return `int` or `void`. This prevents typical programs from leading to deluge of ignored return value errors, but may mean some relevant errors are not detected. In the strict library, they are declared to return `int`, so ignored return value errors will be reported (depending on other flag settings). Programs should check that this return value is non-negative.
- The standard libraries declare some parameters and return values to be alternate types (`int` or `bool`, or `int` or `char`). The ANSI standard specifies these types as `int` to be compatible with older versions of the library, but logically they make more sense as `bool` or `char`. In the strict library, the stronger type is used. The parameter to `assert` is `int` or `bool` in the standard library, and `bool` in the strict library. The parameter to the character functions `isalnum`, `isalpha`, `iscntrl`, `isdigit`, `isgraph`, `islower`, `isprint`, `ispunct`, `isspace`, `isupper`, `isxdigit`, `tolower` and `toupper` is `char` or `int` in the standard library and char in the strict library. The type of the return value of the character classification functions (all of the previous character functions except `tolower` and `toupper`) is `bool` or `int` in the standard library and `bool` in the strict library. The type of the first parameter to `ungetc` is `char` or `int` in the standard library and `char` in the strict library (`EOF` should not be passed to `ungetc`). The second parameter to `strchr` and `strrchr` is `char` or `int` in the standard library and `char` in the strict library.
- The global variables `stdin`, `stdout` and `stderr` are declared as `unchecked` variables (see Section 4.2.1) in the standard libraries. In the strict libraries, they are `checked`.
- The global variable `errno` is declared `unchecked` in the standard libraries, but declared `checkedstrict` in the strict libraries.

If no library flag is used, LCLint will load the standard library, `standard.lcd`. If `+nolib` is set, no library is loaded. The library source files can easily be modified, and new libraries created to better suit a particular application.

Generating the Standard Libraries

The standard libraries are generated from header files included in the LCLint distribution. Some libraries are generated from more than one header file. Since the POSIX library includes the ANSI library, the headers for the ANSI and POSIX libraries are combined to produce the POSIX library. Similarly, the UNIX library is composed of the ANSI, POSIX and UNIX headers. The header files include some sections that are conditionally selected by defining `STRICT`.

The commands to generate the standard libraries are:

```
lclint -nolib ansi.h -dump ansi
lclint -nolib -DSTRICT ansi.h -dump ansistrict
lclint -nolib ansi.h posix.h -dump posix
lclint -nolib -DSTRICT ansi.h posix.h -dump posixstrict
lclint -nolib ansi.h posix.h unix.h -dump unix
lclint -nolib -DSTRICT ansi.h posix.h unix.h -dump unixstrict
```

User Libraries

To enable running LCLint on large systems, mechanisms are provided for creating libraries containing necessary information. This means source files can be checked independently, after a library has been created. The command line option `-dump` *`library`* stores information in the file *`library`* (the default extension, `.lcd`[29], is added). Then, `-load` *`library`* loads the library. The library contains interface information from the files checked when the library was created.

Header File Inclusion

The standard behavior of LCLint on encountering

```
#include <X.h>
```

is to search for a file named *`X`*`.h` on the include search path (set using `–I`) and then the system base include path (read from the `include` environment variable if set or using a default value, usually `/usr/include`). If *`X`*`.h` is the name of a header file in a loaded standard library (either ANSI or POSIX) and *`X`*`.h` is found in a directory that is a system directory (as set by the `-sysdirs` flag; the default is `/usr/include`), *`X`*`.h` will not be included if `skip-ansi-headers` or `skip-posix-headers` (depending on whether *`X`*`.h` is an ANSI or POSIX header file) is on (both are on by default). To force all headers to be included normally, use `-skip-ansi-headers`.

Sometimes headers in system directories contain non-standard syntax that LCLint is unable to parse. The `+skip-sys-headers` flag may be used to prevent any include file in a system directory from being included.

LCLint is fast enough that it can be run on medium-size (10,000 line) programs without performance concerns. It takes about one second to process a thousand source lines on a DEC Alpha. Libraries can be used to enable efficient checking of small modules in large programs. To further improve performance, header file inclusion can be optimized.

When processing a complete system in which many files include the same headers, a large fraction of processing time is wasted re-reading header files unnecessarily. If you are checking a 100-file program, and every file includes `utils.h`, LCLint will have to process `utils.h` 100 times (as would most C compilers). If the `+single-include` flag is used, each header file is processed only once. Single header file processing produces a significant efficiency improvement when checking large programs split into many files, but is only safe if the same header file included in different contexts always has the same meaning (i.e., it does not depend on preprocessor variable defined differently at different inclusion sites).

When processing a single file in a large system, a large fraction of the time is spent processing included header files. This can be avoided if the information in the header files is stored in a library instead. If `+never-include` is set, inclusion of files ending in `.h` is prevented. Files with different suffixes are included normally. To do this the header files must not include any expanded macros. That is, the header file must be processed with `+all-macros`, and there must be no `/*@notfunction@*/` control comments in the header. Then, the `+never-include` flag may be used to prevent inclusion of header files. Alternately, non-function macros can be moved to a different file with a name that does not end in `.h`. Remember, that this file must be included directly from the `.c` file, since if it is included from a `.h` file indirectly, that `.h` file is ignored so the other file is never included.

---

[29] In earlier versions of LCLint, the default extension `.lldmp` was used. This has been shortened to `.lcd`.

These options can be used for significant performance improvements on large systems. The performance depends on how the code is structured, but checking a single module in a large program is several times faster if libraries and `+noinclude` are used.

Preprocessing Constants

LCLint defines the preprocessor constant `__LCLINT__` (two underscores on each side) when preprocessing source files. If you want to include code that is processed only when LCLint is used, surround the code with `# ifdef __LCLINT__` … `#endif`.

Appendix G Specifications

Another way of providing more information about programs is to use formal specifications. Although this document has largely ignored specifications, LCLint was originally designed to use the information in LCL specifications instead of source-code annotations. This document focuses on annotations since it takes less effort to add annotations to source code than to maintain an additional specification file. Annotations can express everything that can be expressed in LCL specifications that is relevant to LCLint checking. However, LCL specifications can provide more precise documentation on program interfaces than is possible with LCLint annotations. This appendix (extracted from [Evans94]) is a very brief introduction to LCL Specifications. For more information, consult [GH93].

The Larch family of languages is a two-tiered approach to formal specification. A specification is built using two languages — the *Larch Shared Language* (LSL), which is independent of the implementation language, and a *Larch Interface Language* designed for the specific implementation language. An LSL specification defines *sorts*, analogous to abstract types in a programming language, and *operators*, analogous to procedures. It expresses the underlying semantics of an abstraction.

The interface language specifies an interface to an abstraction in a particular programming language. It captures the details of the interface needed by a client using the abstraction and places constraints on both correct implementations and uses of the module. The semantics of the interface are described using primitives and sorts and operators defined in LSL specifications. Interface languages have been designed for several programming languages.

LCL [GH93, Tan95] is a Larch interface language for Standard C. LCL uses a C-like syntax. Traditionally, a C module *M* consists of a source file, *M*.c, and a header file, *M*.h. The header file contains prototype declarations for functions, variables and constants exported by *M*, as well as those macro definitions that implement exported functions or constants, and definitions of exported types. When using LCL, a module includes two additional files — *M*.lcl, a formal specification of *M*, and *M*.lh, which is derived by LCLint (if the lh flag is on) from *M*.lcl. Clients use *M*.lcl for documentation, and should not need to look at any implementation file. The derived file, *M*.lh, contains include directives (if *M* depends on other specified modules), prototypes of functions and declarations of variables as specified in *M*.lcl. The file *M*.h should include *M*.lh and retain the implementation aspects of the old *M*.h, but is no longer used for client documentation.

The LCLint release package includes a grammar for LCL and examples of LCL specifications.

Specification Flags

These flags are relevant only when LCLint is used with LCL specifications.

Global Flags

`lcs`
Generate `.lcs` files containing symbolic state of `.lcl` files (used for imports). By default `.lcs` files are generated for each `.lcl` file processed. Use `-lcs` to prevent generation of `.lcs` files.

`lh`
Generate `.lh` files. By default, `-lh` is set and no `.lh` files are generated. Use `+lh` to enable `.lh` file generation.

`i <file>`
Set LCL initialization file to *<file>*. The LCL initialization file is read if any `.lcl` files are listed on the command line. The default file is `lclinit.lci`, found on the `LARCH_PATH`.

`lclexpect` *<number>*
Exactly *<number>* specification errors are expected. Specification errors are errors detected when checking the specifications. They do not depend on the source code.

Implicit Globals Checking Qualifiers

m:-++- `imp-checked-spec-globs`
Implicit `checked` qualifier on global variables specified in an LCL file with no checking annotation.

m:---- `imp-checkmod-spec-globs`
Implicit `checkmod` qualifier on global variables specified in an LCL file with no checking annotation.

m:---+ `imp-checkedstrict-spec-globs`
Implicit `checked` qualifier on global variables specified in an LCL file with no checking annotation.

Implicit Annotations

plain: - `spec-glob-imp-only`
Implicit `only` annotation on global variable declaration in an LCL file with no allocation annotation.

plain: - `spec-ret-imp-only`
Implicit `only` annotation on return value declaration in an LCL file with no allocation annotation.

plain: - `spec-struct-imp-only`
Implicit `only` annotation on structure field declarations in an LCL file with no allocation annotation.

shortcut `spec-imp-only`
Sets `spec-glob-imp-only`, `spec-ret-imp-only` and `spec-struct-imp-only`.

Macro Expansion

plain: + `spec-macros`
Macros defining specified identifiers are not expanded and are checked according to the specification.

Complete Programs and Specifications

m:-+++ `spec-undef`

Function, variable, iterator or constant specified but never defined.

plain: - `spec-undecl`

Function, variable, iterator or constant specified but never declared.

plain: - `need-spec`

There is information in the specification that is not duplicated in syntactic comments. Normally, this is not an error, but it may be useful to detect it to make sure checking incomplete systems without the specifications will still use this information.

shortcut `export-any`

An error is reported for any identifier that is exported but not specified. (Sets all export flags below.)

m:---+ `export-const`

Constant exported but not specified.

m:---+ `export-var`

Variable exported but not specified.

m:---+ `export-fcn`

Function exported but not specified.

m:---+ `export-iter`

Iterator exported but not specified.

m:---+ `export-macro`

An expanded macro exported but not specified

m:---+ `export-type`

Type definition exported but not specified

Appendix H Emacs

LCLint can be used most productively with the `emacs` text editor. The release package includes emacs files for running LCLint and editing code with annotations.

Editing Abbreviations

An additional file, `emacs/lclint-abbrevs` contains abbreviations for LCLint syntactic comments and annotations. If it is loaded, the comment surrounding an LCLint annotation will be added automatically. For example, typing "`only`" and a space, will produce "`/*@only@*/ `". Abbreviations are provided for each LCLint syntactic comment. The abbreviation of `/*@null@*/` is `nll` (not `null`), since it is often necessary to type `NULL`.

Abbreviations are loaded and used when a `.c` or `.h` file is edited by adding these lines to your `.emacs` file:

```
(quietly-read-abbrev-file "<directory>/lclint-abbrevs")
(setq c-mode-hook (function (lambda nil (abbrev-mode 1))))
```

References

LCLint

[Evans94] David Evans. *Using specifications to check source code*. MIT/LCS/TR 628, Laboratory for Computer Science, MIT, June 1994.

SM Thesis. Describes research behind LCLint, focusing on how specifications can be exploited to do lightweight checking. Includes case studies using LCLint.

[EGHT94] David Evans, John Guttag, Jim Horning and Yang Meng Tan. *LCLint: A tool for using specifications to check code*. SIGSOFT Symposium on the Foundations of Software Engineering, December 1994.

Introduction to LCLint. Shows how LCLint is used to find errors in a sample program.

[Evans96] David Evans. *Static Detection of Dynamic Memory Errors*. SIGPLAN Conference on Programming Language Design and Implementation (PLDI '96), Philadelphia, PA., May 1996.

Describes approach for exploiting annotations added to code to detect a wide class of errors. Focuses on checks described in Sections 1-7 of this guide.

Larch

[GH93] Guttag, John V. and Horning, James J., with Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing, Larch: Languages and Tools for Formal Specification, Springer-Verlag, Texts and Monographs in Computer Science, 1993.

Overview of the Larch family of specification languages and related tools. Includes a chapter on LCL, the Larch C interface language, on which LCLint is based.

[Tan95] Tan, Yang Meng. Formal Specification Techniques for Engineering Modular C, Kluwer International Series in Software Engineering, Volume 1, Kluwer Academic Publishers, Boston, 1995.

Modified and updated version of MIT Ph D thesis, previously published as MIT/LCS/TR-619, 1994. Includes presentation of the semantics of LCL and a case study using LCL.

C

[ANSI] American National Standard for Information Systems, Programming Language, C. ANSI X3.159-1989. (Believed to be identical to ISO/IEC 9899:1990).

Specification for C programming language. LCLint aims to be consistent with this document.

[Hat95] Hatton, Les. Safer C: Developing Software for High-integrity and Safety-critical Systems. McGraw-Hill International Series in Software Engineering, 1995.

A broad work on all aspects of developing safety-critical software, focusing on the C language. Provides good justification for the use of C in safety-critical systems, and the necessity of tool-supported programming standards. LCLint users will be interested to see how many of the errors listed as only being dynamically detectable can be detected statically by LCLint.

[KR88] Kernighan, Brian W. and Ritchie, Dennis M. The C Programming Language, second edition. Prentice Hall, New Jersey, 1988.

Standard reference for ANSI C. If you haven't heard of this one, you probably didn't get this far (unless you started at the back).

[vdL94] Van der Linden, Peter. Expert C Programming: Deep C Secrets. SunSoft Press, Prentice Hall, New Jersey, 1994.

Filled with useful information on the darker corners of C, as well as lots of industry anecdotes and humor. LCLint's reserved name checking is loosely based on the list of reserved names in this book.

Abstract Types

[LG86] Liskov, Barbara. and Guttag, John V. Abstraction and Specification in Program Development,
MIT Press, Cambridge, MA, 1986.

Describes a programming methodology using abstract types and specified interfaces. Much of the methodology upon which LCLint is based comes from this book. Uses the CLU programming language.