ANALYZING "C" PROGRAMS FOR COMMON ERRORS

by

Dennis M. Frederick

B.S., University of Missouri, 1970

A REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

COMPUTER SCIENCE

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1988

Approved by:
David A. Gustafson
Dr. Gustafson

CHAPTER 1

INTRODUCTION

The cost of error detection and correction can be a significant contributor to the life cycle cost of software. Some researchers estimate that 40% of the life cycle cost of a software system is in the testing phase [WOLV84]. Clearly, detecting as many errors as possible as early as possible will lower the overall cost of a software development project.

The focus of this investigation is the development of an error detection tool for the C Programming Language which will contribute to a decrease in those costs for projects coded in C. This tool will detect errors involving logical and semantic errors in statements that are syntactically and semantically acceptable to the C compiler.

The C Programming Language has a number of pitfalls which can lead to errors. C is a powerful, high-level language with a rich repertoire of operators, data types and control flow constructs. However, the very richness that makes it attractive also results in a complexity that can be troublesome to both beginning and expert programmers alike. In my experience teaching C Language and consulting on C Language projects, I have noted several errors made frequently by beginners and occasionally by experienced programmers. These errors could be detected by the proposed tool.

There are already several types of tools available for error detection each embodying a slightly different approach to the problem. These include **Dynamic Trace** tools, **Core Dump Analysis** tools, **Static Analysis** tools, and **Goal Analysis** tools. An example of each of these types is explained in some detail in the paragraphs that follow.

Dynamic Trace tools like "**ctrace**" [UNIX84] allow step by step execution of C programs with outputs to show the values of important variables along the way. **Ctrace** takes the program source file as input. It inserts statements which will print the text of all executable statements and the values of all referenced variables. The output is then directed to standard output which is normally redirected to a temporary file. The C compiler is then invoked and the newly inserted print statements are compiled as part of the program. Upon execution, the text of each executable statement is printed as it is executed. The values of any variables referenced by the statement being executed are also printed. When **ctrace** detects loops in a program, the loop is traced once, then tracing is stopped until the loop is exited or a different sequence of statements within the loop is executed. A warning message is printed every 1000 times through a loop to help detect infinite loops. Tools of this type are particularly useful for uncovering problems with control flow.

Core Dump Analysis tools like "**sdb**" [KATS81] are useful for examining "core image" files produced by aborted program executions. **Sdb** (currently implemented for C and Fortran 77) has a variety of features and capabilities which make it valuable. When using **sdb** with a C program, one must compile the source with the "-g" option. **Sdb** can then be invoked after program execution has aborted and produced a "core dump". It will reveal the procedure name and line number in which the error occurred. Upon request **sdb** will also output a stack trace that consists of a list of the called procedures which led to the error. Included in that listing are the values of the input parameters at the time of the error. **Sdb** also has extensive capabilities for displaying variables of the program. Values of the variables can be displayed in a variety of user-selected formats and variable addresses are readily available upon request. Another useful feature of **sdb** is its ability to do breakpoint debugging. Once **sdb** is invoked, breakpoints can be set at any point in the program. Execution is initiated and continues until just before the breakpoint is reached. **Sdb** then halts the program and gives access to the core image examination capabilities already mentioned. Rounding out **sdb's** capabilities is the ability to single step the program and examine the core image after each statement. All these features together make **sdb** a very useful debugging tool.

Static Analysis tools like "**lint**" [JOHN81] examine program source files for a variety of possible misuses of the language. **Lint** pursues many types of errors which may lead to improper execution even though the program may compile with no errors. Several of these are discussed below. **Lint** will issue warnings about variables and functions which are declared but never again referred to in the program. While these variables are usually left over from previous versions of the program and are probably harmless, they are flagged in the interest of encouraging good programming practice. Also, **lint** attempts to detect variables which are used in a program before being assigned a value. **Lint** also attempts to analyze the control flow of a program. It will complain about portions of a program which are apparently unreachable and loops which cannot be entered from the top or exited from the bottom. Finally, **lint** discourages older forms of the compound assignment operators as well as what are referred to as "strange constructions" such as tests that can never succeed (or fail) and statements that have no effect whatsoever.

Goal Analysis tools like "**Proust**" [SOLL85] (a debugging aid for Pascal) are useful in a teaching environment where specific programming assignments are given and the student's trial solution is analyzed for its effectiveness in solving the problem. **Proust** uses an interesting strategy for uncovering bugs in the proposed solution. First, it retrieves a description of the particular problem assignment from a library of such descriptions written in **Proust's** own problem-description language. The problem description is a paraphrase of the English language problem assignment given to the student. The problem description contains descriptions of each of the sub-problems or tasks that must be performed successfully for the overall problem to be solved. Next, **Proust** draws from a library of valid solutions to each task, attempting to find a match with the student's approach to that task. If a match is found, **Proust** infers that the task is implemented correctly. If a match is not found, **Proust** checks a database of possible bugs to see if it can explain the discrepancies. It then reports an English language description of the bug and will sometimes go so far as to suggest input data that will demonstrate the presence of the bug. Processing continues in this manner until all the tasks of the problem have been analyzed.

All these tools are valuable members of the family of software development tools. However, none of them addresses the particular errors I have targeted for this investigation.

Errors of this class occur in statements that are syntactically and semantically correct. The errors escape detection by the C compiler and can be very difficult to find. Nonetheless, they are usually abuses of the language resulting in statements that probably do not accurately reflect the programmers intentions and will almost always cause execution errors. In limited cases these "errors of intent" do not cause execution errors, but reflect such bad programming practice and make a program so vulnerable to failure over its life cycle that they should be reported as errors. The specific errors under consideration in this investigation involve misuse of the assignment and comparison operators, unintentional use of the null statement, errors of omission in switch/case statements, improper parameter specification in scanf() function calls and the use of uninitialized pointers. There may be other errors of this class equally worthy of detection effort.

CHAPTER 2

REQUIREMENTS

In order to find these "errors of intent" the tool must have several general capabilities. It must be able to identify those constructs which have the potential for harboring the errors being sought. Also, it must be able to filter out and discard statements not of interest to the tool. The tool needs the ability to make judgments concerning the appropriateness of constructs used in the suspect statements. These judgments will vary depending on the particular type of error being pursued at any given time. Finally it must be able to report the errors detected and the line numbers on which they were found.

The following paragraphs contain descriptions of the particular errors targeted by this tool along with more information concerning the capabilities required for the detection of each error.

ASSIGNMENT VS COMPARISON

One of the most common errors arises from confusion over the assignment and equality comparison operators. Some high level languages (notably Pascal and Ada) use := for assignment and = for comparison, whereas, C Language uses

= for assignment and == for comparison. This inevitably leads to misuse of these operators by programmers already familiar with the other languages. Even programmers not already familiar with using the equal sign for comparison, find it a natural choice for that use. Furthermore, from my experience in the classroom, this error is very difficult for individuals to detect in their own programs.

For example, if a programmer has momentarily forgotten the operator definitions, a construct like:

```
if ( z = 4 )
{ .
  .
  .
}
or

while ( x = y )
 { .
   .
   .
 }
```

can appear so reasonable and correct that programmers will almost always look elsewhere for an error. One can conceive of instances where a programmer might intentionally control a **while** loop with the assignment of one variable to another knowing the result will eventually be zero (false) thereby causing the loop to terminate. However, this is considered bad programming practice and will be reported as a possible error by the proposed tool.

There are, of course, some legitimate uses of the assignment operator within control flow constructs. For example:

```
while ( (c = getchar()) != EOF )
  { .
    .
    .
  }
```

which is a very useful technique for assigning a character from standard input to a variable and then comparing it with the end of file marker in the same statement. This tool will not flag legitimate constructs such as this as errors.

Inappropriate use of the assignment operator is likely to occur within the test/comparison portion of the control flow constructs (**if, while, for, do-while**) therefore, my tool will analyze those statements for this class of errors. The tool must have the ability to distinguish between appropriate and inappropriate assignments and issue warnings only after encountering the latter.

THE NULL STATEMENT (;)

Another very frequent error is the unintentional termination of a control flow construct with the null statement (;). Most lines of code in a C Language program end with a semicolon. However, the control flow constructs (**if, while,**

and for) do not. If one places a semicolon after one of these constructs the behavior of the program can be altered significantly. For instance:

```
for ( i = 0; i < 10; i++)
 {
 name[i] = tempname[i];
 }
   .
   .
   .
```

will copy the first ten elements of the **tempname** array to the **name** array, whereas:

```
for ( i = 0; i < 10; i++);
 {
 name[i] = tempname[i];
 }
   .
   .
   .
```

will execute the null statement ten times, then copy the eleventh element of the **tempname** array to the **name** array.

There are legitimate uses of the null statement following control constructs as in:

```
while ( (c = getchar()) <= ' ');
```

which is a good technique for skipping over white space in an input line. However, a less confusing way to code this would be:

```
while ( (c = getchar()) <= ' ')
  ;
```

which shows more explicitly that the null statement is intended as the object of the **while** statement.

This tool will check for a new line between the control flow construct and the null statement and issue a warning if one does not appear.

OMISSION OF BREAK STATEMENTS

Another common error involves the omission of **break** statements in the cases of a **switch** statement. C handles this type of control flow construct differently from other languages in that execution proceeds from one **case** to the next until a **break** statement or the end of the **switch** is encountered. Programmers familiar with other languages will often forget to include **break** statements where they are needed, thereby producing a control flow different from that intended.

Consider the following code fragment for processing command line options and setting flags based on the options received. Assume the **-h** option has been entered on the command line. The statements associated with **case 'h'** will be executed.

```
while (argc > 1 && (*++argv)[0] == '-') {
  for (s = argv[0]+1; *s != ' '; s++) {
    switch (*s) {
     case 'h':
          help = TRUE; /* provide usage info */
          break;
     case 'f':
          file = TRUE; /* log output in a file */
          break;
     default:
          printf("Invalid Option %c0, *s);
          argc = 0;
          break;
  }
}
   argc--;
}
```

If the **break** statement in case **'h'** is omitted, execution will continue with the statements of case **'f'**. Therefore, both the file and help flags will be set whenever the **-h** option is selected on the command line.

There are times when omission of **break** statements is intentional and the manner in which C handles **cases** becomes a convenient way to accomplish a logical ORing of **case** matches. This is illustrated in the following example.

```
switch(x) {
  case 'a':
  case 'e':
  case 'i':
  case 'o':
  case 'u':
        vowel++;
        break;
  default:
        consonant++;
        break;
}
```

This code fragment increments the vowel counter when either an **a**, e, **i**, **o**, or **u** is contained in **x**. My error checker will not issue warnings about constructs like this. It will assume the programmer intentionally omitted the **break** statements in order to create a logical OR structure in the switch. All other occurrences of missing **break** statements will be flagged as errors.

SCANF() FUNCTION CALLS

Yet another common error is improper specification of the input parameters in **scanf()** function calls. The **scanf()** function assumes that each of its input parameters is a valid address for storage of the information being scanned in. However, there are many ways to specify an address in C Language. Therefore, it is not surprising that confusion over the proper usage often leads to syntactically correct but numerically erroneous address specifications. Consider the following example.

```
main()
{
  char first[15];
  char last[15];
  char job[7];
  char *p;
  float sal;
  p = last;
  scanf("%s %s %c %d",first, p, &job[0], &sal);
}
```

In this example, **first**, **p**, **&job[0]**, and **&sal**, are all valid address references. However, in the slightly modified version below, **first[0]**, **p**, **job**, and **sal** are all syntactically acceptable while only the reference to **job** would produce the intended result.

```
main()
{
  char first[15];
  char last[15];
  char job[7];
  char *p;
  float sal;
  p = last;
  scanf("%s %s %c %d",first[0], *p, job, sal);
}
```

To uncover problems of this nature the tool must produce a symbol table of all names used in the program. The table contains an indication of whether the name is an array, a pointer, or a single-element variable. Then all **scanf()** function calls are analyzed to determine whether the parameters are specified correctly. And, of course, the tool issues a warning message upon encountering an erroneous specification.

POINTERS

Another error that is frequently made in C Language involves the use of pointers which have not been initialized or which have been initialized incorrectly. In the following example the pointer **p** has been declared but never initialized so its use in the **scanf()** function call will produce unpredictable results.

```
main()
{
  char aircraft[10];
  char *p;
  scanf("%s", p );
}
```

And in the example below the pointer **p** has been initialized improperly (i.e. initialization should be **p = &x;**).

```
main()
{
  int x;
  int *p;
  p = x;
  scanf("%d", p );
}
```

My tool checks to see that each pointer declared is assigned a valid address before it is used in the program.

These then are the errors the debugging tool will attempt to uncover. The statements containing them are syntactically and semantically acceptable to the compiler. Therefore, it must go beyond the level of checking for illegal syntax and semantics. It must be capable of assessing and making judgments about the programmers intentions, and issuing warnings if the constructs in question are suspected of harboring errors. And like all tools of this type it must be careful to complain only when there is a high probability that an error has actually occurred, lest it gain a reputation for issuing unnecessary warnings.

CHAPTER 3

DESIGN

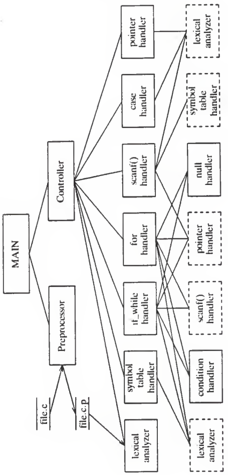
The tool I propose to uncover these errors is called **dust**. To identify these errors, **dust** must scan the source code, extract information about variables, ignore correct statements, and further investigate suspect statements. To accomplish this, **dust** consists of a preprocessor, a controller, a lexical analyzer, a symbol table handler, and a separate module for each error type being detected. The modules directly involved in error detection are: the if_while handler, the for handler, the condition handler, the null handler, the case handler, the scanf() handler, and the pointer handler. Overall organization of the program is shown in the hierarchy chart in Figure 3-1.

THE PREPROCESSOR

The purpose of this module is to strip out **#include** statements and comments from the source file and to perform the substitutions called for by the user's **#define** statements. The reason for deleting the **#include** statements is to limit the scope of the code being diagnosed to the source file.

The preprocessor works in the following manner. First, it takes the source file as input and executes a system command that removes the **#includes**,

Figure 3-1: Module Hierarchy



placing the output in a temporary file called "file.c.x". Next, it invokes the C preprocessor which draws its input from the temporary file, performs the **#define** substitutions and strips out the comments. The output is placed in another file called "file.c.p" and the temporary file "file.c.x" is removed. At this point the file "file.c.p" is ready for input to the lexical analyzer.

THE CONTROLLER

The next module invoked is the controller. This module directs the activities of all the other modules. It reads tokens from the input delivered by the lexical analyzer until it detects a token of interest to one of the other modules. If the token indicates the beginning of a statement related to one of the error types, control is passed to the appropriate error detection module. If the token indicates a declaration, control is passed to the symbol table handler. When control is passed to one of the modules below the controller, that module assumes the task of calling the lexical analyzer to obtain additional tokens as needed. When the task performed by the error detection module or symbol handler is completed, control is returned to the controller.

One special purpose task is built into the controller to handle the special problem presented by **do-while** statements. The closing **while** of a **do-while** statement is always terminated by a semicolon. If special steps were not taken, this could cause the tool to produce erroneous null statement error messages. Therefore, the controller keeps track of any **do-while** statements which have been opened by a **do** statement, but not as yet closed off by the corresponding **while** statement. This information is made available to other modules through a global variable. In this way, if a semicolon-terminated **while** statement is encountered, the null handler will know whether it is a null statement error, or merely the expected closing **while** of a **do-while** statement.

The tool, because of it modularity, is easy to expand. Additional error checks could be very easily incorporated into the tool by adding more error detection modules under the controller.

THE LEXICAL ANALYZER

This module has two very simple responsibilities. Its primary task is to break the source code into tokens. A secondary task is to keep track of the current line number.

It obtains input from "file.c.p" produced by the preprocessor. Output consists of a numeric token and a type indicator (i.e. keyword, constant, identifier or operator) for each call to the module.

THE SYMBOL TABLE HANDLER

This module builds a symbol table and answers queries from the other modules about entries in the table. Symbol table entries for each identifier will include the name, an assigned identifier number, variable type (i.e. array, structure, single-element, pointer, etc.) and, in the case of pointers, an indication of whether is has been assigned a value. The formation of this table is crucial to the operation of the error detection modules which rely on information about the identifiers.

THE IF_WHILE HANDLER

This module handles the error checking of **if** and **while** statements. Execution is triggered when the controller detects the presence of an **if** or **while** keyword. The module then brings in additional tokens from the lexical analyzer to complete the statement. Then it breaks the statement into its component parts: namely, the keyword segment, the conditional (e.g. (x = = y), (a <= b), ((c=getchar()) != EOF)) and the object statement of the construct (if, and only if, it appears on the same line). Then it passes the conditional segment to the condition handler, and the object statement to the null handler. Also, if this module detects the presence of a **scanf()** function call, it calls the scanf() handler. The scanf() handler performs its error checking and then returns control to the if_while handler to continue analyzing the statement. In addition, if it detects a pointer it calls the pointer handler which performs its error check and returns control to the if_while handler.

Input to this module consists of tokens from the lexical analyzer. Output is in the form of pointers to the conditional and object segments of the statement.

THE FOR HANDLER

This module handles the error checking of **for** statements. Execution is triggered when the controller detects the presence of a **for** keyword. The module then brings in additional tokens from the lexical analyzer to complete the statement. Like the if_while handler, it then breaks the statement into its component parts: namely, the keyword segment, the conditional (e.g. (x == y), (a <= b), ((c=getchar()) != EOF)) and the object statement of the construct (if, and only if, it appears on the same line). Then it passes the conditional segment to the condition handler, and the object statement to the null handler. Like the if_while handler it calls the scanf() handler when it detects a **scanf()** function call, and the pointer handler when it finds a pointer.

Input to this module consists of tokens from the lexical analyzer. Output is in the form of pointers to the conditional and object segments of the statement.

THE CONDITION HANDLER

This module detects assignment operators appearing in the conditional construct that were really intended to be relational operators. As a by-product of this error checking, the module will also have some knowledge of possible operator precedence errors. Since it is convenient, these errors will be reported in addition to misuses of the assignment operator.

The condition handler performs a series of steps in pursuit of errors. First, it does a quick scan through the conditional construct to see if there are any assignment or relational operators present. If there are no assignment operators and no relational operators, error message 3 will be issued. If there is an assignment operator but no relational operator, the module assumes the assignment operator was intended to be a relational operator (most likely the equality relation) and, therefore, issues error message 1. If a relational operator is found, an additional check is performed. Specifically, the module checks to make sure the assignment operator has been forced to a lower precedence than the relational operator by the proper use of parentheses. If it has not, error message 2 is issued.

Input to this module is a pointer to the conditional segment of the control construct. Output is in the form of three possible error messages generated under the conditions described above. They are:

Error message 1: "line #: - Misuse of assignment operator (=) in **"if" (while, for)** - try (==)".

Error message 2: "line #: - Operator precedence error involving assignment in **"if" (while, for)**".

Error message 3: "line #: - No relational operators in **"if" (while, for)**".

THE NULL HANDLER

This module determines whether a control construct has been terminated with a null statement on the same line of code. It receives as input the character string beginning with the first character after the closing parenthesis of the conditional construct and ending with the new line character. If it finds only a semicolon (possibly surrounded by spaces or tabs) followed by a new line, an error message will be output warning that a possible misuse of the null statement has occurred. A non-null statement followed by a semicolon will be considered acceptable on the same line of code, although to the purist this might also be considered bad programming practice.

Input to this module is a pointer to the object statement associated with the conditional. Output, in the event of an error, is the warning: "line #: - Null statement (;) after **"if" (while, for)**".

THE SCANF() HANDLER

This module is responsible for analyzing the arguments to scanf() function calls to determine whether they provide valid address specifications. It will acquire the variable type from the symbol table for each identifier in the function call. Then it will decide whether the syntax used in the argument list is appropriate. If the syntax is not appropriate it will issue a warning message.

This module receives as input a pointer to the beginning of the scanf() argument list. Output upon detection of an error is the warning: "line #: - Incorrect address specification for "variable" in scanf()".

THE CASE HANDLER

This module reviews the statements associated with the **case** labels of **switch/case** statements to insure that a **break** statement is among them. If one or more **cases** do not contain **breaks** the module will issue a warning message. An exception to this case is when two or more **case** labels appear sequentially with no statements in between. This is a convenient way to form an "or" construct in this language and will not be flagged as an error. However, the last **case** in the sequence is still required to have a **break** statement associated with it.

Input to this module is a pointer to the first **case** label in the **switch** statement. Output on detection of an error is the message: "line #: - No break at end of "case".

THE POINTER HANDLER

This module keeps track of pointer initializations, and "first uses" of each pointer. The pointer handler will update the symbol table when a pointer is first initialized. Then later in the program when the pointer appears again, it will check to verify that it has been initialized. If the pointer is used without first being assigned a value, the module issues an error message. Input to the module is the identity of the pointer variable in question. Output in the event of an error is the message: "line #: - Possible uninitialized pointer - "variable".

THE USER INTERFACE

This section describes how the user executes the tool, how options are specified, and how certain features of the tool will help a novice user. The name of the error detection tool is "**dust**". The simplest form of the command line calling for the application of **dust** to a source file would consist of the name **dust** followed by the source file specification.

Example:

$ dust program.c

By default all error checks are activated. The user can selectively suppress any of the checks by selecting the appropriate command line option. The options and their meanings are described below.

Options:

-h	**HELP! Print usage information.**
-a	**Suppress check for inappropriate assignment.**
-b	**Suppress check for break statements**
-n	**Suppress check for null statements.**
-p	**Suppress check for uninitialized pointers.**
-s	**Suppress check of scanf() function arguments.**

Multiple options can be placed behind a single dash or each can be given its own dash prefix. Furthermore, the order in which the options appear on the command line is unimportant as long as all options appear before the source file is specified. In the following examples the "-a" and "-n" options are selected to suppress the checks for inappropriate assignments and unintentional null statements. They are all equivalent usages.

Examples:

$ dust -an program.c

$ dust -na program.c

$ dust -a -n program.c

$ dust -n -a program.c

Selecting the "-h" option will cause a short help message to be printed on the terminal. The help message includes a description of the tool along with a list of all the available options and their meanings.

The user interface will issue error messages about invalid options, no source file specification, or a source file specification that does not end in ".c".

CHAPTER 4

IMPLEMENTATION

Dust was implemented using approximately 1600 lines of C - Language code. Each of the error checks is performed by a separate function called from the mainline program, or in some cases called from another error checking routine. Details of the code can be found in the appendices.

Several of the author's C programs were run through **dust** with very encouraging results. In some cases, errors were deliberately planted in the programs to test the tool's effectiveness. **Dust** consistently detected and reported all the errors it was designed to uncover and did not report errors where none existed.

Average CPU times were computed for programs of various sizes. Overall, **dust** consumed **1.92** seconds per **100** lines-of-code processed running on a Digital Equipment Corporation VAX 11/780. Large programs tended to have a lower time to lines-of-code ratio than small ones.

Several example runs of **dust** appear below. The sample program used here was constructed specifically to demonstrate some of the error checking capabilities of **dust** - it performs no real computing function. Some features of the user interface are also demonstrated by intentional omission or faulty entry of command line arguments.

```
$ pr -tn test3.c
  1    main()
  2    {
  3    int x = 2, y = 3;
  4      if ( x = y );
  5      while ( x|y )
  6        ;
  7      while ( x >= y );
  8      for (x = 0; x = y;x++ );
  9    }

$ dust
You must specify a source file to be checked!
For help use: dust -h

$ dust -h

This command searches C Language programs for a variety of errors.

It assumes your program has compiled successfully, but is not running properly.
By default, all the error checks are activated.  To selectively
suppress any of the checks, use the appropriate command line option(s).

OPTIONS:

        -h  HELP!
        -a  Suppress check for inappropriate assignments.
        -b  Suppress check for breaks in switch/case statements.
        -n  Suppress check for unintentional null statements.
        -p  Suppress check for uninitialized pointers.
        -s  Suppress check for improper scanf() function arguments.

EXAMPLES:

        $ dust program.c

        $ dust -an program.c

$ dust test3
Source file name must end with ".c"

$ dust test5.c
Can't open "test5.c" for reading.

$ dust test3.c
line  4: - Misuse of assignment operator ( = ) in "if" ( try == ).
```

```
line  4: - Null statement (;) after "if".
line  5: - No relational operators in "while".
line  7: - Null statement (;) after "while".
line  8: - Misuse of assignment operator ( = ) in "for" ( try <= ).
line  8: - Null statement (;) after "for".


$ dust -z test3.c
Invalid Option z
For help use: dust -h

$ dust -n test3.c
line  4: - Misuse of assignment operator ( = ) in "if" ( try == ).
line  5: - No relational operators in "while".
line  8: - Misuse of assignment operator ( = ) in "for" ( try <= ).

$
```

CHAPTER 5

CONCLUSION

The debugging tool **dust** is a useful addition to the family of software development tools for C - Language. **Dust** finds errors in usage of the language commonly made by beginning programmers and occasionally made by experienced programmers. The errors it finds are typically difficult to detect without the aid of such a tool. The tool does not report errors where none exist, and it consistently finds all the errors it was designed to uncover.

The tool could be extended to include some additional checks not included in this design due to time constraints. One addition could be a check of the arguments to printf() function calls (similar to the scanf() check already implemented). Another could be a check for the presence of nested comments which are illegal in C, but which are detected by the compiler only indirectly. Another useful check would be a simple check for the presence of a semicolon at the end of each statement (ignoring, of course, those statements which should *not* end in a semicolon). The absence of a semicolon where it is needed usually generates a long list of compiler syntax errors which do not always pinpoint the line of code where the semicolon has been omitted. An explicit check of this kind would save time and should be fairly easy to implement.

BIBLIOGRAPHY

1. [SOLL85] "PROUST: An Automatic Debugger for Pascal Programs", Elliot Solloway and W. Lewis Johnson, Byte Magazine April, 1985 pg179-190.

2. [UNIX84] "UNIX System V User Reference Manual (Release 2.0)", AT&T Bell Laboratories, 1984.

3. [WOLV84] "Software Costing", R. W. Wolveton from Handbook of Software Engineering edited by C. R. Vick and C. V. Ramamoorthy: Von Nostrand Reinhold Co. Inc, New York 1984.

4. [JOHN81] "Lint, a C Program Checker", S. C. Johnson from Documents for Unix Vol 1, January 1981, Bell Telephone Laboratories, Inc.

5. [KATS81] "SDB - A Symbolic Debugger", H. P. Katseff from Documents for Unix Vol 1, January 1981, Bell Telephone Laboratories, Inc.

APPENDIX A

USER'S MANUAL

NAME

dust - check for common C programming errors

SYNOPSIS

dust [**-habnps**] file.c

DESCRIPTION

Dust is a debugging tool for C - Language programs. It searches for several common abuses of the language which are not detected by the compiler. Specifically, it detects inappropriate uses of the assignment operator and operator precedence errors in control constructs, unintended null statements, omission of break statements from switch/case constructs, improper address specifications in scanf() function calls, and uninitialized pointers. It does not do the normal syntax and semantics checking done by the compiler. In fact, **dust** assumes the program being tested has already compiled successfully, but is not operating correctly.

By default, all the error checks are activated. However, the user can suppress checks by selecting the appropriate option(s) below.

-h	HELP! Print usage information.
-a	Suppress check for inappropriate assignments.
-b	Suppress check for break statements.
-n	Suppress check for null statements.
-p	Suppress check for uninitialized pointers.
-s	Suppress check of scanf() function arguments.

APPENDIX B

FUNCTION DESCRIPTIONS

This appendix contains detailed descriptions of each function in the program. Descriptions include the purpose of the function, the inputs and outputs, and an explanation of the more important features of the code itself.

lex()

Purpose:

The lexical analyzer (lex()) reads source code from the file "file.c.p" produced by the preprocessor and forms the characters into the tokens of the language (keywords, identifiers, operators etc.).

The lex() function calls several subordinate functions to form tokens while some are formed within lex() itself. Tokens consisting of a single character are formed by lex(), with the exception of single character identifiers which are formed by key_id(). Tokens comprised of more than one character are formed by functions subordinate to lex(). As soon as the first character of the token is brought in by getch(), a decision is made as to which function will process the remaining characters. If the first character is a letter or underscore, control is passed to the key_id() function. If the first character is not a letter, control passes to the function associated with the case matched by that character. (See the large switch/case statement in lex()).

When the EOF is encountered, lex() takes care of closing and removing the working input file (filname.c.p) and exiting the command. The saves all the other functions that call lex() from the trouble of checking each time to see if EOF has been encountered.

Lex() keeps track of the value and type of the last token (in Lastok and Lastype). Also, it keeps track of special tokens which are of interest to the case_hand function (in Lastcasetok and Lastcasetype). Namely, the last token which was not a semicolon and not a newline. The case_hand function uses this information when making its decision about whether a break statement has been omitted from a case.

Input:

Characters are read in using calls to the getch() function which returns a single character from the input file each time it is called. Characters can also be "put back" on the input stream, by calls to the ungetch(c) function. Successive calls to getch() will get the characters that were "put back" before getting new characters from the file.

Output:

The external variables Token and Type are assigned values by lex(). If the token is a keyword or operator, Token is assigned the value given by the #define statement for that operator or keyword. If the token is a constant, Token is given the value of the constant. (Note: lex() does not attempt to form floating point, or character constants into single tokens. For example, a floating point constant would be broken into three tokens. Namely, the integer part, the decimal point, and the fractional part.) If the token is an identifier, lex() calls a hashing function which converts the identifier character string to an integer between 0 and 2000. That integer is then assigned to Token. Collisions (i.e. different character strings hashing to the same value) are handled by the symbol table handler to make sure that each identifier gets its own entry in the table. Type is assigned one of the following values: 'o' if the token is an operator, and 'c' if its a constant.

Important variables:

Token .... integer value for token.

Type .... char value for type of token.

Lastok .... integer value for previous token.

Lastype .... char value for type of previous token.

Lastcasetok .... integer value for previous token of interest to case handler.

Lastcasetype ... char value for type of previous token of interest to case handler.

Charbuff[] ..... char array for storage of input characters as token is being formed.

Charpos .... integer index into Charbuff[] array.

Other internal functions called:

what_type()
key_id()
numproc()
exclamproc()
percentproc()
amperproc()
starproc()
plusproc()
minusproc()
slashproc()
lessproc()
equalproc()
greatproc()
xorproc()
pipeproc()
getch()

key_id()

Purpose:

The purpose of key_id() is to determine whether the character string it processes is a keyword or an identifier and then assign the appropriate value to Token and Type. It is called whenever the first character of a new token is a letter or underscore. The first thing it does is bring in the remainder of the string (all letters, digits or underscores) and put them in the Charbuff[] array. It then ungetch()'s the last character getch()'ed (since we know it is not part of this token - i.e. not a letter, digit, or underscore) and null terminates the string in Charbuff[]. It then resets Charpos to point to the beginning of the string and calls findkey() to see if the token is a keyword. If it is a keyword, Type gets the value KEYWD and Token get the value keytable[n].keynum (n = return value of findkey). The structure keytable is a table consisting of character strings which are the keywords, and their corresponding numeric values (defined by #define statements). If the token is not a keyword, Type gets the value ID and Token gets an integer value representing the character string returned by the function pchash().

Input:

Characters are brought in by calls to getch() and placed in the character array Charbuff[].

Output:

Type is assigned the value KEYWD if the token is a keyword. Type is assigned the value ID if the token is an identifier. Token is assigned the value corresponding to the keyword found (keytable[n].keynum) or an integer value returned by pchash() representing the identifier.

Important variables:

Token .... integer value for token.

Type .... char value for type of token.

Charbuff[] ..... char array for storage of input characters as token is being formed.

Charpos .... integer index into Charbuff[] array.

keytable[n].keynum integer value defined for the keyword of index n.

n .... holds value returned from findkey (the index into the keyword table).

Other internal functions called:

what_type()
getch()
ungetch()
findkey()
pchash()

findkey()

Purpose:

The findkey() function does a binary search through keytable (indirectly accessed by local structure tab) to find the keyword (if any) which matches the string in Charbuff[] (accessed through local variable word). This search uses the strcmp() function for string comparisons. If the first string is lexically "less than" the second, a negative value is returned. If the first string is lexically "greater than" the second, a positive value is returned. If the first string is "equal to" the second, zero is returned. The search progresses by setting the high, low, and mid values depending on the returns from strcmp(). Therefore, the stings in the lookup table must be in ascending alphabetic order for this binary search algorithm to work.

Input:

Input parameters are a pointer to Charbuff[], a copy of the structure keys, and the size of the keytable.

Output:

Returns the value of the index into keytable for the keyword matched, or the value -1 if a keyword match was not found (in which case the token must be an identifier).

Important variables:

Charbuff[] ..... char array for storage of input characters as token is being formed.

low .... low boundary of search

mid .... midpoint of search

high .... high boundary of search

cond .... integer that holds return value of strcmp().

Other internal functions called:

none

getch()

Purpose:

The purpose of getch() is to bring in a single character from the file associated with the FILE pointer inptr each time it is called (inptr points to filename.c.p). It first looks to see if there is a character in the buffer Buf (the result of an ungetch()), before going to the file for a character.

Input:

A single character from filename.c.p.

Output:

Returns the character gotten.

Important variables:

Buf[] .... external character array to buffer input characters.

Bufp .... external index into the Buf[] array.

Other internal functions called:

none

ungetch()

Purpose:

The purpose of ungetch() is to "put a character back on the input stream". It actually puts the character back into the buffer Buf[], but this is transparent (and irrelevant) to the calling function. The next call to getch() gets the character "put back" by the previous ungetch() (if any).

Input:

The character to be "put back" is delivered to ungetch() as an input parameter.

Output:

One character to the buffer Buf[].

Important variables:

Buf[] .... external character array to buffer input characters.

Bufp .... external index into the Buf[] array.

c .... temporary storage for character to be "put back".

Other internal functions called:

none

what_type()

Purpose:

The purpose of what_type is to determine whether a character is a letter (a - z, A - Z, or the underscore) or a digit (0 - 9).

Input:

The character to be checked for type is input as a parameter.

Output:

Output is a return value of either LETTER or DIGIT.

Important variables:

c .... temporary storage for character to be "put back".

Other internal functions called:

none

numproc()

Purpose:

The purpose of numproc() is to process tokens that consist entirely of digits. It is called whenever the first character of the token is a digit (0-9). In this case the token must be a number since identifiers cannot begin with a digit.

Input:

Characters are read in using calls to the getch() function which returns a single character from the input file each time it is called. Character are read into Charbuf[] until a non-digit is found. Then the non-digit is ungetch()'ed and the string in Charbuf[] is null terminated.

Output:

Type is assigned the value CONST. Token is assigned the actual numeric value of the token (returned by atoi(Charbuff) - ascii to integer conversion).

Important variables:

Token .... integer value for token.

Type .... char value for type of token.

Charbuff[] ..... char array for storage of input characters as token is being formed.

Charpos .... integer index into Charbuff[] array.

Other internal functions called:

getch()
ungetch()

exclamproc() percentproc() amperproc() starproc() plusproc() minusproc() slashproc() lessproc() equalproc() greatproc() xorproc() pipeproc()

Purpose:

The purpose of these functions is to process the operator tokens. The first character brought in by lex() determines which function will be called (see the large switch/case statement in lex()). The functions all consist of if/else if constructs which determine what the following characters are and therefore what the operator is. Any unwanted characters getch()'ed (i.e. characters which cannot be a part of any operator beginning with the first character already received) are ungetch()'ed immediately and the string in Charbuff[] is null terminated.

Input:

Characters are read in using calls to the getch() function which returns a single character from the input file each time it is called. Characters are read into Charbuf[] as the token is formed.

Output:

Type is assigned the value OP. Token is assigned the value for the operator as specified in the #define statements.

Important variables:

Token .... integer value for token.

Type .... char value for type of token.

Charbuff[] ..... char array for storage of input characters as token is being formed.

Charpos .... integer index into Charbuff[] array.

c .... temporary storage for character to be "put back".

Other internal functions called:

getch()
ungetch()

symhand()

Purpose:

The purpose of the symhand function is to determine the variable type (single element variable, array, pointer, etc.) and call the make_entry() function to make the actual table entry.

Input:

Tokens are brought in using calls to the lex() function which returns a single token from the input file each time it is called.

Output:

none

Important variables:

Token .... integer value for token.

token .... local integer value for token.

vartype .... type of variable

init .... initialize flag

Other internal functions called:

make_entry()

make_entry()

Purpose:

The purpose of the make_entry function is to make entries in the symbol table. Each entry includes the token value, the variable name, the type of variable, the block number in which the variable is declared, and the initialized flag (YES or NO).

Input:

vartype, token, name, and init from calling function

Output:

Important variables:

token .... local integer for token value

vartype .... type of variable

name .... name of variable

init .... local initialize flag

Symtable[].newtoken Symbol table entry for token value

Symtable[].id .. Symbol table entry for variable name

Symtable[].block Symbol table entry for code block

Symtable[].vartype Symbol table entry for variable type

Symtable[].init Symbol table entry for init flag

Other internal functions called:

none

find_entry()

Purpose:

The purpose of the find_entry function is to find entries in the symbol table. Information drawn from the symbol table on the "searched for" variable includes the token value, the variable name, the type of variable, the block number in which the variable is declared, and the initialized flag (YES or NO).

Input:

token from calling function

Output:

Passes back information on variable to calling function.

Important variables:

token .... local integer for token value

vartype .... type of variable

name .... name of variable

init .... local initialize flag

Symtable[].newtoken Symbol table entry for token value

Symtable[].id .. Symbol table entry for variable name

Symtable[].block Symbol table entry for code block

Symtable[].vartype Symbol table entry for variable type

Symtable[].init Symbol table entry for init flag

Other internal functions called:

none

if_while_hand()

Purpose:

The purpose of the if_while_hand is to process statements that begin with the if, while, or do keywords. Processing for do statements consists of pushing the do onto a stack (Dostack) to be popped later by the corresponding while of the do/while. This is to prevent the closing while's of do/while statements from being mistakenly flagged as "null statement following while" errors. Processing of if and while statements consists of stripping out the conditional segment of the statement (the test between the outermost parentheses) and placing the tokens in the local buffer tokbuf[] and typebuf[] for the token values and types respectively. The function expects the next character after the if or while to be an open parenthesis. When it encounters this it increments parencnt, then enters a while loop which brings in the rest of the characters out to and including the closing parenthesis of the conditional. This is accomplished by incrementing parencnt for every open parenthesis and decrementing it for every close parenthesis. When parencnt reaches zero we have found the closing parenthesis of the conditional. It then calls cond_hand() to evaluate the conditional and null_hand to look for null statement errors (if the appropriate flags are turned on).

Input:

Tokens are brought in using calls to the lex() function which returns a single token from the input file each time it is called. Token values and types are read into the local buffers tokbuf[] and typebuf[]. Conditional type (if, while, do) is received as an input parameter to the function.

Output:

tokbuf[], typebuf[], and condtype are passed to the cond_hand() function. condtype is passed to the null_hand function.

Important variables:

condtype .... type of conditional (if, while, do).

tokbuf[] .... local buffer for tokens in conditional segment.

typebuf[] .... local buffer for token types in conditional segment.

Dostack .... stack for do's and {'s - popped by while's and }'s.

Doptr .... index into Dostack

parencnt .... counter for open and close parentheses.

Token .... integer value for token.

Type .... char value for type of token.

Assignments ... flag for activating check for misused assignments (cond_hand).

Null_stmnts .... flag for activating check for null statements after if/while.

keytable[] ...... to retrieve name corresponding to condtype

Other internal functions called:

lex()
cond_hand()
null_hand()

for_hand()

Purpose:

The purpose of the for_handler is to process statements that begin with the for keyword. Processing consists of stripping out the conditional segment of the for (the test between the semicolons) and placing the tokens in the local buffers tokbuf[] and typebuf[] for for token values and token types respectively. The for_hand function brings in tokens after the for until it encounters the first semicolon and puts an open parenthesis into the token buffer in place of the it. Then it gets all tokens up to the next semicolon and puts them in the buffer. A closing parenthesis is then put in the buffer in place of the semicolon. The semicolons are replaced by open and close parenthesis so that the string delivered to cond_hand will look the same whether it came from an if, while, or for.

Input:

Tokens are brought in using calls to the lex() function which returns a single token from the input file each time it is called. Token values and types are read into the local buffers tokbuf[] and typebuf[]. Conditional type (for) is received as an input parameter to the function.

Output:

tokbuf[], typebuf[], and condtype are passed to the cond_hand() function. condtype is passed to the null_hand function.

Important variables:

condtype .... type of conditional (if, while, do).

tokbuf[] .... local buffer for tokens in conditional segment.

typebuf[] .... local buffer for token types in conditional segment.

Token .... integer value for token.

Type .... char value for type of token.

Assignments ... flag for activating check for misused assignments (cond_hand).

Null_stmnts .... flag for activating check for null statements after if/while.

keytable[] ...... to retrieve name corresponding to condtype

Other internal functions called:

lex()
cond_hand()
null_hand()

cond_hand()

Purpose:

The purpose of the cond_hand function is to analyze the conditional (test) segment of if, while, and for statements. It first scans through the tokens in the conditional (using tokbuf[] and typebuf[] to count the number of assignment and conditional operators. (See first switch/case statement in function). It does not count assignment operators preceded by a single quote since these are character constants ('='). If there are no relational operators and no assignment operators then no test is being performed in the conditional and an error message is issued. If there are assignment operators but no conditionals, a misuse of the assignment operator has probably occurred (i.e. = instead of ==) and a different error message is issued. If there are both assignments and relationals then the statement is checked for proper operator precedence (i.e. the assignment should be forced to higher precedence than the relational by the use of parentheses). (Refer to second switch/case in function).

If an assignment is encountered, all the tokens are read up to and including the next relational operator. An open parentheses decrements parencnt. A close parentheses increments parencnt. If further assignments are encountered, assigncnt is decremented. If after the relational operator is encountered, the parencnt is less than an or equal to zero then we did not have an unmatched close parenthesis between the assignment and the relational operator. This means we have an operator precedence error.

If a relational operator is encountered, all the tokens are read up to and including the next assignment. An open parentheses increments parencnt. A close parentheses decrements parencnt. If after the assignment operator is encountered, the parencnt is less than an or equal to zero then we did not have an unmatched open parenthesis between the relational and the assignment operator. This means we have an operator precedence error.

This process continues until either assigncnt or relopcnt goes to zero.

Input:

tokbuf[], typebuf[], and condtype are passed in from if_while_hand and for_hand.

Output:

Appropriate error messages if errors are detected.

Important variables:

condtype .... type of conditional (if, while, do).

tokbuf[] .... local buffer for tokens in conditional segment.

typebuf[] .... local buffer for token types in conditional segment.

assigncnt .... counter for assignment operators.

relopcnt .... counter for relational operators.

optype[] .... array for storing type of op (ASSIGN or RELOP)

Other internal functions called:

none

null_hand()

Purpose:

The purpose of the null_hand() function is to check for the presence of null statements on the same line as if, while, or for constructs. A null statement appearing on the line following the if, while, or for is acceptable. This function is called from the if_while_hand and the for_hand functions. First null_hand checks condtype to see if the construct is a for. If it is, we must first arrive at the closing parenthesis of the for. (If its an if or a while, we're already there). Therefore we call lex() as long as the token is not a close parenthesis. Once we arrive at the close parenthesis we make an additional call to lex() to move to the first token beyond the close parenthesis of the for. We then make additional calls to lex() until we encounter either a newline or semicolon, incrementing loopcnt each time through the loop. (A loopcnt value of zero means no statements were encountered before the newline or semicolon). If we encountered a newline we will increment the loopcnt. If we encountered a semicolon with no statement since the closing parenthesis of the for, we may have a null statement error. If condtype is while and we think we may have a null statement error, we must first check to see if there is a do on the Dostack. If there is not a do on the Dostack, we have a null statement error and we issue the appropriate error message. If there is a do on the Dostack then this null-terminated while is appropriate. We simply pop the do off the Dostack and continue. If condtype is either if or for and loopcnt is zero, we have a null statement error and we issue the appropriate error message.

Input:

condtype (if, while, for, do) from if_while_hand or for_hand.

Output:

Appropriate error messages if errors are detected.

Important variables:

condtype .... type of conditional (if, while, do).

Dostack .... stack for do's and {'s - popped by while's and }'s.

Doptr .... index into Dostack

Token .... integer value for token.

Type .... char value for type of token.

loopcnt .... counter for statements before newline or semicolon keytable[] ...... to retrieve name corresponding to condtype

Other internal functions called:

lex()

case_hand()

Purpose:

The purpose of the case_hand function is to check for missing breaks in switch/case statements. If the last non-newline, non-semicolon token (before the case token was encountered) was either a colon or a break or if the switchflag was on (meaning this is the first case in the switch) then there is no error. In this case we simply turn the Switchflag off (it may already be off, but that's okay). However, if the last non-newline, non-semicolon token was not a colon or break and the flag was off, we have a missing break statement in the switch. The function lex() as mentioned earlier, keeps track of the last tokens of interest to this function.

Input:

none

Output:

Appropriate error messages if errors are detected.

Important variables:

Lastok .... integer value for previous token.

Lastype .... char value for type of previous token.

Switchflag ...... flag to indicate if this is the first case in switch.

Other internal functions called:

none

scanf_hand()

Purpose:

The purpose of the scanf_hand function is to check for incorrect address specifications in scanf() function calls. It checks the type of the variables in the scanf() argument list, (i.e. single variable, array, pointer, pointer array) and determines whether the syntax used to specify the address is appropriate. If not, a warning message is issued.

Input:

Tokens are brought in using calls to the lex() function which returns a single token from the input file each time it is called. Token values and types are read into the local buffers tokbuf[] and typebuf[].

Output:

Appropriate error messages if errors are detected.

Important variables:

Lastok .... integer value for previous token.

Lastype .... char value for type of previous token.

Charbuff .... character input buffer.

Token .... integer value for token.

Type .... char value for type of token.

tokbuf .... local token buffer

typebuf .... local token type buffer

vartype .... type of variable

name .... name of variable

Other internal functions called:

find_entry() is used to find variables in the symbol table.

point_hand()

Purpose:

The purpose of the point_hand function is to check for uninitialized pointers.

Input:

A token and a variable name are passed in from calling function.

Output:

Appropriate error messages if errors are detected.

Important variables:

token .... integer value for token.

vartype .... type of variable

name .... name of variable

init .... initialize flag

Other internal functions called:

find_entry()

set_init()

Purpose:

The purpose of the set_init function is to set the initialized flag associated with a pointer whenever that pointer is initialized. This flag can then be checked whenever needed to see if the pointer is initialized.

Input:

A token and a variable name are passed in from calling function.

Output:

none

Important variables:

token .... integer value for token.

name .... name of variable

init .... initialize flag

Other internal functions called:

none()

APPENDIX C

DUST SOURCE CODE LISTING

```
/*
 * NAME: dust.c
 * PROGRAMMER: Dennis Frederick
 * DATE: June 7, 1987
 *
 *
 *
 *
 *
 * INPUTS: filename.c
 *
 *
 *
 *
 *
 *
 * OUTPUTS. error messages
 *
 *
 *
 *
 *
 *
 * OPTIONS:
 *
 *     -h    HELP! Print usage information.
 *     -a    Suppress check for inappropriate assignment.
 *     -b    Suppress check for break statements in switch/case statements.
 *     -n    Suppress check for unintentional null statements.
 *     -p    Suppress check for uninitialized pointers.
 *     -s    Suppress check for improper scanf() function arguments.
 *
 *
 *
 *
 *
 */

/*
 * DEFINES FOR C LANGUAGE KEYWORDS
 */

#define ASM         0
#define AUTO        1
#define BREAK       2
#define CASE        3
#define CHAR        4
#define CONTINUE    5
#define DEFAULT     6
#define DO          7
#define DOUBLE      8
#define ELSE        9
#define ENTRY       10
#define ENUM        11
#define EXTERN      12
#define FLOAT       13
```

```
#define FOR         14
#define FORTRAN     15
#define FSCANF      16
#define GOTO        17
#define IF          18
#define INT         19
#define LONG        20
#define MAIN        21
#define REGISTER    22
#define RETURN      23
#define SCANF       24
#define SHORT       25
#define SIZEOF      26
#define SSCANF      27
#define STATIC      28
#define STRUCT      29
#define SWITCH      30
#define TYPEDEF     31
#define UNION       32
#define UNSIGNED    33
#define VOID        34
#define WHILE       35

/*
 * DEFINES FOR INPUT TOKENS
 */

#define NOT         100
#define NE          101
#define REM         102
#define REMEQ       103
#define AND         104
#define LOGAND      105
#define ANDEQ       106
#define OPENPAR     107
#define CAST        108
#define CLSPAR      109
#define STAR        110
#define STAREQ      111
#define PLUS        112
#define PPLUS       113
#define PLUSEQ      114
#define COMMA       115
#define MINUS       116
#define MMINUS      117
#define MINUSEQ     118
#define ARROW       119
#define DOT         120
#define SLASH       121
#define SLASHEQ     122
#define LT          123
#define SHFTL       124
#define SHFTLEQ     125
#define LE          126
#define EQ          127
#define EQEQ        128
#define GT          129
#define GE          130
#define SHFTR       131
#define SHFTREQ     132
```

```
#define QUEST        133
#define OPENBRAK     134
#define CLSBRAK      135
#define XOR          136
#define XOREQ        137
#define OR           138
#define OREQ         139
#define LOGOR        140
#define TWOSCOMP     141
#define OPENBRACE    142
#define CLSBRACE     143
#define SEMI         144
#define COLON        145
#define DBLQT        146
#define SNGLQT       147
#define POUND        148
#define BKSLASH      149
#define NL           150
#define OPENCOMM     151
#define CLSCOMM      152
#define ATSIGN       153
#define GRAVE        154
#define DOLLAR       155

#define OP          'o' /* operator token id */
#define KEYWD       'k' /* keyword token id */
#define ID          'i' /* identifier token id */
#define CONST       'c' /* constant token id */

#define ASSIGN      'a' /* assignment operator id */
#define RELOP       'r' /* relational operator id */
#define LOGOP       'l' /* logical operator id */


#define BUFSIZE     100 /* input buffer size */
#define HASHSIZE    2000 /* hash table size */

#define LETTER      'a' /* letter id */
#define DIGIT       '0' /* digit id */
#define NKEYS       36  /* number of keywords */

#define PTR         'p' /* pointer id */
#define PTRARRAY    'z' /* pointer array id */
#define ARRAY       'a' /* array id */
#define SINGLE      's' /* single variable id */

#define TRUE   1
#define YES    1
#define ON     1
#define FALSE  0
#define NO     0
#define OFF    0

#include <stdio.h>

/*
 * EXTERNAL DECLARATIONS
 */

struct keys {               /* keyword lookup table */
```

```
   char keyword[10];
   int  keynum;
} keytable[NKEYS] = {
   "asm", ASM,
   "auto", AUTO,
   "break", BREAK,
   "case", CASE,
   "char", CHAR,
   "continue", CONTINUE,
   "default", DEFAULT,
   "do", DO,
   "double", DOUBLE,
   "else", ELSE,
   "entry", ENTRY,
   "enum", ENUM,
   "extern", EXTERN,
   "float", FLOAT,
   "for", FOR,
   "fortran", FORTRAN,
   "fscanf", FSCANF,
   "goto", GOTO,
   "if", IF,
   "int", INT,
   "long", LONG,
   "main", MAIN,
   "register", REGISTER,
   "return", RETURN,
   "scanf", SCANF,
   "short", SHORT,
   "sizeof", SIZEOF,
   "sscanf", SSCANF,
   "static", STATIC,
   "struct", STRUCT,
   "switch", SWITCH,
   "typedef", TYPEDEF,
   "union", UNION,
   "unsigned", UNSIGNED,
   "void", VOID,
   "while", WHILE
};

char Buf[BUFSIZE];        /* input buffer */
char Type;                /* type of token */
char Charbuff[BUFSIZE];   /* character input buffer */

int Bufp;                 /* Buf array index */
int Charpos;              /* Charbuff array index */
int Token;                /* input token */
int Linenum = 1;          /* current line number */
int Block;                /* current code block */
int Dostack[BUFSIZE];     /* trackes do statements */
int Doptr;                /* Dostack array index */
int Lastok;               /* previous input token */
int Lastype;              /* type of previous token */
int Lastcasetok;          /* previous token within case */
int Lastcasetype;         /* type of previous token within case */
int Switchflag = OFF;     /* in switch = ON, otherwise OFF */


int Help = FALSE;         /* help option flag */
```

```
int Assignments = TRUE;    /* assignment option flag */
int Break_stmnts = TRUE;   /* breaks in switch/case option flag */
int Null_stmnts = TRUE;    /* unintentional null option flag */
int Pointers = TRUE;       /* uninitialized pointers option flag */
int Scan_args = TRUE;      /* improper scanf() args option flag */

char Command[50];          /* "system" command buffer */
char Workfile[14];         /* filename.c.p */

/*
 * SYMBOL TABLE
 */

struct tab {
  char id[100];
  char dcltype;
  char vartype;
  int newtoken;
  int block;
  int init;
} Symtable[HASHSIZE], Nulltable;

FILE *inptr;               /* file input pointer */

main (argc, argv)
int    argc;               /* argument counter */
char   *argv[];            /* pointers to arguments */
{

/*
 * INTEGER DECLARATIONS
 */

int x;                     /* holds return from lex() */
int i;                     /* general index */
int token;                 /* value of token */
int init;                  /* pointer initialized flag */


/*
 * CHARACTER DECLARATIONS
 */

char   *s;                 /* scratchpad pointer */
char vartype;              /* variable type */
char name[100];            /* variable name */

/*
 * Get desired options - set option flags
 */

while (argc > 1 && (*++argv)[0] == '-') {
      for (s = argv[0]+1; *s != '\0'; s++) {
            switch (*s) {
            case 'h':
                  Help = TRUE;
                  break;
            case 'a':
                  Assignments = FALSE;
                  break;
```

```
                case 'b':
                        Break_stmnts = FALSE;
                        break;
                case 'n':
                        Null_stmnts = FALSE;
                        break;
                case 'p':
                        Pointers = FALSE;
                        break;
                case 's':
                        Scan_args = FALSE;
                        break;
                default:
                        fprintf(stdout,"Invalid Option %c\n", *s);
                        fprintf(stdout,"For help use: dust -h\n");
                        exit(0);
                }
        }
        argc--;
}

/*
 * Does user want help?  If so, print help info.
 */
        if (Help == TRUE) {
        fprintf(stdout,"\nThis command searches C Language programs for a variety of errors.\n");
        fprintf(stdout,"\nIt assumes your program has compiled successfully, but is not running properly.\n");
        fprintf(stdout,"By default, most error checks are activated.  To selectively\n");
        fprintf(stdout,"suppress or activate checks, use the appropriate command line option(s).\n\n");
        fprintf(stdout,"OPTIONS:\n\n");
        fprintf(stdout,"\t-h  HELP!\n");
        fprintf(stdout,"\t-a  Suppress check for inappropriate assignments.\n");
        fprintf(stdout,"\t-b  Suppress check for breaks in switch/case statements.\n");
        fprintf(stdout,"\t-n  Suppress check for unintentional null statements.\n");
        fprintf(stdout,"\t-p  Suppress check for uninitialized pointers.\n");
        fprintf(stdout,"\t-s  Suppress check for improper scanf() function arguments.\n\n");
        fprintf(stdout,"EXAMPLES:\n\n");
        fprintf(stdout,"\t$ dust program.c\n\n");
        fprintf(stdout,"\t$ dust -an program.c\n\n");
        exit(1);
        }
/* if no source file, print error */
if (argc != 2) {
        fprintf(stdout,"You must specify a source file to be checked!\n");
        fprintf(stdout,"For help use: dust -h\n");
        exit(0);
}
else {
        for (s = argv[0]+1; *s != '\0'; s++)
          ;
/* if no .c suffix, print error */
        if ( (*(s-1) != 'c') | (*(s-2) != '.') ) {
          printf("Source file name must end with \".c\"\n");
          exit(1);
        }
 * if can't open source file, print error */
        if((inptr = fopen(argv[0],"r")) == NULL) {
                fprintf(stderr,"Can't open \"%s\" for reading.\n",argv[0]);
                exit(1);
         }
```

```
        fclose(inptr);
    }
/* gather #includes in a file */
sprintf(Command,"grep '^#include' %s > %s.x", argv[0], argv[0]);
system(Command);
sprintf(Workfile,"%s.x", argv[0]);
if((inptr = fopen(Workfile,"r")) == NULL) {
    fprintf(stderr,"Can't open \"%s\" for reading \n",Workfile);
    exit(1);
}
/* offset Linenum by number of #includes */
while ( (x = getch()) != EOF )
  if (x == '\n') Linenum++;
fclose(inptr);
/* get rid of includes */
sprintf(Command,"grep -v '^#include' %s > %s.x", argv[0], argv[0]);
system(Command);
/* run C preprocessor */
sprintf(Command,"cc -E %s.x > %s.p", argv[0], argv[0]);
system(Command);
/* get rid of superfluous output from cc -E */
sprintf(Command,"grep -v '^#' %s.p > %s.x", argv[0], argv[0]);
system(Command);
/* move file.c.x to file c.p */
sprintf(Command,"mv %s.x %s.p", argv[0], argv[0]);
system(Command);
sprintf(Workfile,"%s.p", argv[0]);
if((inptr = fopen(Workfile,"r")) == NULL) {
    fprintf(stderr,"Can't open %s for reading \n",Workfile);
    exit(1);
}
while ( (x=lex()) != EOF) {     /* input tokens until EOF */
/* strip out comments */
  if ( Token == OPENCOMM && Type == OP ) {
    while( Token != CLSCOMM || Type != OP ) {
      lex();
    }
  }
/* strip out quoted strings */
  if ( (Token == DBLQT && Type == OP) &&
      (Lastok != BKSLASH || Lastype != OP) ) {
    do {
      lex();
    }
    while( (Token != DBLQT || Type != OP) ||
          (Lastok == BKSLASH && Lastype == OP) );
    lex();
  }

/*
 * Determine what action to take for this input token.
 */

  if (Type == OP) {
    switch(Token) {
/* if pending do, push brace onto Dostack */
      case OPENBRACE:
                if (Dostack[0] != NULL) Dostack[Doptr++] = OPENBRACE;
              break;
/* clear out local variables after end of block */
```

```
        case CLSBRACE:
                for ( i = 0; i < HASHSIZE; i++ ) {
                  if (Symtable[i].block == Block)
                    Symtable[i] = Nulltable;
                }
/* if pending do, pop brace off Dostack */
                if (Dostack[0] != NULL) Dostack[--Doptr] = NULL;
                break;
      }
    }
    else if ( Type == KEYWD) {
      switch (Token) {
        case INT:
        case LONG:
        case SHORT:
        case UNSIGNED:
        case CHAR:
        case FLOAT:
        case DOUBLE:
        case STATIC:
        case REGISTER:
                symhand();
                /*dumptable();*/
                break;
        case IF:
              if_while_hand(IF);
                break;
        case WHILE:
              if_while_hand(WHILE);
                break;
        case DO:
              if_while_hand(DO);
                break;

        case FOR:
             for_hand(FOR);
                break;


        case SWITCH:
              if ( Break_stmnts == TRUE )
                Switchflag = ON;
                break;

        case CASE:
        case DEFAULT:
              if ( Break_stmnts == TRUE )
                case_hand();
              break;
        case SCANF:
        case SSCANF:
        case FSCANF:
              if ( Scan_args == TRUE )
                scanf_hand();
                break;

        default:
             continue;
      }
    }
```

```
 else if (Type == ID && Pointers == TRUE) {
      strcpy(name, Charbuff);
      point_hand(T ken, name);
 }
}
} /* end of main */


/* DUMP SYMBOL TABLE - USED DURING DEVELOPMENT */

dumptable()
{
int i;
printf("\nSymtable\n\n");
 for ( i = 0; i < HASHSIZE; i++ ) {
   if (Symtable[i].newtoken != 0)
   printf("%d\tid = %s\tnewtoken = %d\tvartype = %c\tblock = %d\n",
i ,Symtable[i].id, Symtable[i].newtoken, Symtable[i].vartype, Symtable[i].block);
 }
}


/* LEXICAL ANALYZER - GET INPUT TDKENS */

lex()
{
Lastok = Token;
Lastype = Type;
if ( Token != NL && Token != SEMI ) {
  Lastcasetok = Token;
  Lastcasetype = Type;
}

for ( Charpos = 0; Charpos < 100; Charpos++ )
      Charbuff[Charpos] = 0;

Charpos = 0;          /* reset Buffer index */
Type = 'x';          /* bogus initalizer */

while ( ( Charbuff[Charpos] = getch() ) == ' '   (Charbuff[Charpos] == '\t') )       /* skip white space */
  ; /* null statement */
  if ( Charbuff[Charpos] == EOF ) {
    fclose(inptr);
    sprintf(Command,"rm %s", Workfile);
    system(Command);
    exit(0);
  }
  if ( what_type(Charbuff[Charpos]) == LETTER )
    key_id();
  else {
     switch(Charbuff[Charpos]) {
       case '0':
       case '1':
       case '2':
       case '3':
       case '4':
       case '5':
       case '6':
       case '7':
       case '8':
```

```
        case '9':
            numproc();
            break;

        case '!':
            exclamproc();
            break;
        case '%':
            percentproc();
            break;
        case '&':
            amperproc();
            break;
        case '(':
            Type = OP;
            Token = OPENPAR;
            break;
        case ')':
            Type = OP;
            Token = CLSPAR;
            break;
        case '*':
            starproc();
            break;
        case '+':
            plusproc();
            break;
        case ',':
            Type = OP;
            Token = COMMA;
            break;
        case '-':
            minusproc();
            break;
        case '.':
            Type = OP;
            Token = DOT;
            break;
        case '/':
            slashproc();
            break;
        case '<':
            lessproc();
            break;
        case '=':
            equalproc();
            break;
        case '>':
            greatproc();
            break;
        case '?':
            Type = OP;
            Token = QUEST;
            break;
        case '[':
            Type = OP;
            Token = OPENBRAK;
            break;
        case ']':
            Type = OP;
```

```
            Token = CLSBRAK,
            break;
        case '^':
            xorproc();
            break;
        case '|':
            pipeproc();
            break;
        case '~':
            Type = OP;
            Token = TWOSCOMP;
            break;
        case '{':
            Type = OP;
            Token = OPENBRACE;
            break;
        case '}':
            Type = OP;
            Token = CLSBRACE;
            break,
        case ';':
            Type = OP;
            Token = SEMI;
            break,
        case ':':
            Type = OP,
            Token = COLON,
            break,
        case '\"':
            Type = OP;
            Token = DBLQT;
            break;
        case '\'':
            Type = OP;
            Token = SNGLQT;
            break;
        case '#':
            Type = OP;
            Token = POUND;
            break;
        case '\\':
            Type = OP;
            Token = BKSLASH,
            break;
          case '@':
            Type = OP;
            Token = ATSIGN;
            break;
        case '`':
            Type = OP;
            Token = GRAVE;
            break;
        case '$':
            Type = OP;
            Token = DOLLAR;
            break;
        case '\n':
            Type = OP;
            Token = NL;
            Linenum++;
```

```
                break;
            default:
                printf("Illegal character - I quit!\n");
                exit(-1);
        }
    }
} /* end of lex */


/*
 * Determine whether token is a keyword or identifier.
 */

key_id()
{
int c, n;
while ( what_type(c = Charbuff[++Charpos] = getch()) == LETTER ||
        what_type(c) == DIGIT)
  ;
ungetch(c);
Charbuff[Charpos] = '\0';
Charpos = 0;
if ( (n = findkey(Charbuff, keytable, NKEYS) ) >= 0) {
  Type = KEYWD;
  Token = keytable[n].keynum;
}
 else {
  Type = ID;
  Token = pchash(Charbuff);
}
} /* end of key_id */

/* bring in an input digit */

numproc()
{
int c;
while ( what_type(c = Charbuff[++Charpos] = getch()) == DIGIT)
  ;
ungetch(c);
Charbuff[Charpos] = '\0';
Charpos = 0;
Type = 'c';
Token = atoi(Charbuff);
} /* end of numproc */


/*
 * Use binary search to find keyword from table
 */

findkey(word, tab, n)
char *word;
struct keys tab[NKEYS];
int n;
{
int low, high, mid, cond;
low = 0;
high = n - 1;
while (low <= high) {
```

```
    mid = (low + high) / 2;
    if ( (cond = strcmp(word, tab[mid].keyword)) < 0 )
      high = mid - 1;
    else if ( cond > 0 )
      low = mid + 1;
    else
      return(mid);
  }
  return(-1);
} /* end of findkey */


/*
 * get a character from input stream or from
 * Buf[Bufp] if availalbe.
 */

getch()
{
  if ( Bufp > 0 )
    return (Buf[--Bufp]);
  else
    return(getc(inptr));
} /* end of getch */

/*
 * put a character back "on the shelf" in Buf[Bufp]
 */

ungetch(c)
int c;
{
  if (Bufp > BUFSIZE)
    printf("ungetch: too many characters\n");
  else
    Buf[Bufp++] = c;
} /* end of ungetch */

/*
 * Determine what type the token is ( Letter or digit )
 * and return appropriate indication.
 */

what_type(c)
int c;
{
if ( (c >= 'a' && c <= 'z') ||
     (c >= 'A' && c <= 'Z') ||
     (c == '_') )
  return(LETTER);
else if  (c >= '0' && c <= '9')
  return(DIGIT);
else
  return(c);
} /* end of what_type */


/*
 * Process exclamation mark tokens.
 */
```

```
exclamproc()
{
int c;
  if ( (c = Charbuff[++Charpos] = getch()) == '=') {
    Type = OP;
    Token = NE;
    return;
  }
  else {
    Type = OP;
    Token = NOT;
    ungetch(c);
    Charbuff[Charpos] = '\0';
    return;
  }
} /* end of exclamproc */


/*
 * Process percent sign tokens.
 */

percentproc()
{
int c;
  if ( (c = Charbuff[++Charpos] = getch()) == '=') {
    Type = OP;
    Token = REMEQ;
    return;
  }
  else {
    Type = OP;
    Token = REM;
    ungetch(c);
    Charbuff[Charpos] = '\0';
    return;
  }
} /* end of percentproc */


/*
 * Process ampersand tokens
 */

amperproc()
{
int c;
  if ( (c = Charbuff[++Charpos] = getch()) == '=') {
    Type = OP;
    Token = ANDEQ;
    return;
  }
  else if (c == '&') {
    Type = OP;
    Token = LOGAND;
  }
  else {
    Type = OP;
    Token = AND;
```

```
    ungetch(c);
    Charbuff[Charpos] = '\0';
    return;
  }
} /* end of amperproc */


/*
 * Process asterisk tokens.
 */

starproc()
{
int c;
  if ( (c = Charbuff[++Charpos] = getch()) == '=') {
    Type = OP;
    Token = STAREQ;
    return;
  }
  else if (c == '/') {
    Type = OP;
    Token = CLSCOMM;
  }
  else {
    Type = OP;
    Token = STAR;
    ungetch(c);
    Charbuff[Charpos] = '\0';
    return;
  }
} /* end of starproc */


/*
 * Process plus sign tokens.
 */

plusproc()
{
int c;
  if ( (c = Charbuff[++Charpos] = getch()) == '=') {
    Type = OP;
    Token = PLUSEQ;
    return;
  }
  else if (c == '+') {
    Type = OP;
    Token = PPLUS;
  }
  else {
    Type = OP;
    Token = PLUS;
    ungetch(c);
    Charbuff[Charpos] = '\0';
    return;
  }
} /* end of plusproc */


/*
```

```
 * Process minus sign tokens.
 */

minusproc()
{
int c;
 if ( (c = Charbuff[++Charpos] = getch()) == '=') {
   Type = OP;
   Token = MINUSEQ;
   return;
 }
 else if (c == '-') {
   Type = OP;
   Token = MMINUS;
 }
 else if (c == '>') {
   Type = OP;
   Token = ARROW;
 }
 else {
   Type = OP;
   Token = MINUS;
   ungetch(c);
   Charbuff[Charpos] = '\0';
   return;
 }
} /* end of minusproc */


/*
 * Process slash tokens.
 */

slashproc()
{
int c;
 if ( (c = Charbuff[++Charpos] = getch()) == '=') {
   Type = OP;
   Token = SLASHEQ;
   return;
 }
 else if (c == '*') {
   Type = OP;
   Token = OPENCOMM;
 }
 else {
   Type = OP;
   Token = SLASH;
   ungetch(c);
   Charbuff[Charpos] = '\0';
   return;
 }
} /* end of slashproc */


/*
 * Process less than tokens.
 */

lessproc()
```

```
{
int c;
 if ( (c = Charbuff[++Charpos] = getch()) == '=') {
   Type = OP;
   Token = LE;
   return;
 }
 else if (c == '<') {
   if ( (c = Charbuff[++Charpos] = getch()) == '=') {
     Type = OP;
     Token = SHFTLEQ;
     return;
   }
   else {
     Type = OP;
     Token = SHFTL;
     ungetch(c);
     Charbuff[Charpos] = '\0';
     return;
   }
 }
 else {
   Type = OP;
   Token = LT;
   ungetch(c);
   Charbuff[Charpos] = '\0';
   return;
 }
} /* end of lessproc */


/*
 * Process equal sign tokens.
 */

equalproc()
{
int c;
 if ( (c = Charbuff[++Charpos] = getch()) == '=') {
   Type = OP;
   Token = EQEQ;
   return;
 }
 else {
   Type = OP;
   Token = EQ;
   ungetch(c);
   Charbuff[Charpos] = '\0';
   return;
 }
} /* end of equalproc */


/*
 * Process greater than tokens
 */

greatproc()
{
int c;
```

```
    if ( (c = Charbuff[++Charpos] = getch()) == '=') {
      Type = OP;
      Token = LE;
      return;
    }
    else if (c == '>') {
      if ( (c = Charbuff[++Charpos] = getch()) == '=') {
        Type = OP;
        Token = SHFTREQ;
        return;
      }
      else {
        Type = OP;
        Token = SHFTR;
        ungetch(c);
        Charbuff[Charpos] = '\0';
        return;
      }
    }
    else {
      Type = OP;
      Token = GT;
      ungetch(c);
      Charbuff[Charpos] = '\0';
      return;
    }
  } /* end of greatproc */


  /*
   * Process xor tokens
   */

  xorproc()
  {
  int c;
    if ( (c = Charbuff[++Charpos] = getch()) == '=') {
      Type = OP;
      Token = XOREQ;
      return;
    }
    else {
      Type = OP;
      Token = XOR;
      ungetch(c);
      Charbuff[Charpos] = '\0';
      return;
    }
  } /* end of xorproc */


  /*
   * Process pipe symbol tokens.
   */

  pipeproc()
  {
  int c;
    if ( (c = Charbuff[++Charpos] = getch()) == '=') {
      Type = OP;
```

```
      Token = OREQ;
      return;
    }
    else if (c == '|') {
      Type = OP;
      Token = LOGOR;
    }
    else {
      Type = OP;
      Token = OR;
      ungetch(c);
      Charbuff[Charpos] = '\0';
      return;
    }
} /* end of pipeproc */


/*
 * Hashing function
 */

pchash(s)
char *s;
{
  int hashval;
  for ( hashval = 0; *s != '\0'; )
    hashval += *s++;
  return (hashval % HASHSIZE);
}

/*
 * Handles symbol table entries
 */

symhand()
{
char vartype;
int token;
char name[BUFSIZE];
int init = NO;
lex();
while ( Token != SEMI ) {
   if ( Token == STAR ) {
     lex();
     token = Token;
     strcpy(name, Charbuff);
     if ( Token == STAR ) {
       vartype = PTRARRAY;
       lex();
       token = Token;
       strcpy(name, Charbuff);
       make_entry(vartype, token, name, init);
     }
     else {
       lex();
       if ( Token == OPENBRAK ) {
         vartype = PTRARRAY;
         make_entry(vartype, token, name, init);
       }
       else {
```

```
          vartype = PTR;
          if (Token == EQ && Type == OP ) init = YES;
          make_entry(vartype, token, name, init);
        }
      }
    }
    else if ( Type == ID ) {
      token = Token;
      strcpy(name, Charbuff);
      lex();
      if ( Token == OPENBRAK ) {
        vartype = ARRAY;
        make_entry(vartype, token, name, init);
      }
      else {
        vartype = SINGLE;
        make_entry(vartype, token, name, init);
      }
    }
    if ( Token == SEMI )
      break;
    lex();
  }
} /* end of symhand */


/*
 * Makes symbol table entries
 */

make_entry(vartype, token, name, init)
char vartype;
int token;
char *name;
int init;
{
int search;
int count = 0;
search = token;
while (Symtable[search++ % HASHSIZE].newtoken != 0 && count <= HASHSIZE)
  ;
if (count > HASHSIZE) {
  printf("Symbol table overflow - I quit!\n");
  exit(1);
}
search--;
Symtable[search].newtoken = search;
strcpy(Symtable[search].id, name);
Symtable[search].block = Block;
Symtable[search].vartype = vartype;
Symtable[search].init = init;
} /* end of make_entry */


/*
 * Finds symbol table entries
 */

find_entry(vartype, token, name, init)
char *vartype;
```

```
int token;
char *name;
int *init;
{
int count;
int search = token;
int searchblock;
if ( (token == Symtable[token].newtoken) &&
    (strcmp(name, Symtable[token].id) == 0) &&
    (Block == Symtable[search].block) )
  {
   *vartype = Symtable[search].vartype;
   *init = Symtable[search].init;
   return(Symtable[search].newtoken);
  }
else {
  for (searchblock = Block; searchblock >= 0; searchblock--) {
    count = 0;
    search = token + 1;
    while ( ((strcmp(name, Symtable[search % HASHSIZE].id) != 0)
          (Block != Symtable[search % HASHSIZE].block)) &&
          ( search <= HASHSIZE) ) {
      count++;
      search++,
    }
  }
}
if (strcmp(name, Symtable[search%HASHSIZE].id) == 0) {
  *vartype = Symtable[search].vartype;
  *init = Symtable[search].init;
}
else {
  *init = 1;
}
return(Symtable[search].newtoken);
} /* end of find_entry */


/*
 * Process if or while statement
 */

if_while_hand(condtype)
int condtype;
{
int tokbuf[BUFSIZE];
char typebuf[BUFSIZE];
int i;
int parencnt = 0;
for ( i = 0; i < 100; i++) {
  tokbuf[i] = 0;
  typebuf[i] = '\0';
}
i = 0;
if (condtype == DO) {
  Dostack[Doptr++] = DO;
  return;
}
lex();
if ( (typebuf[i] = Type, tokbuf[i] = Token) != OPENPAR || Type != OP)
```

```
    printf("line %3d: - No \"(\" after if/while!\n",Linenum);
  else parencnt++;

  i++;
  while ( parencnt > 0 ) {
    lex();
    typebuf[i] = Type;
    tokbuf[i++] = Token;
    if ( Type == OP )
      switch (Token) {
        case OPENPAR:
                parencnt++;
                break;
        case CLSPAR:
                parencnt--;
                break;
      }
  }
  if ( Assignments == TRUE)
    cond_hand(tokbuf, typebuf, condtype);
  if ( Null_stmnts == TRUE)
    null_hand(condtype);
} /* end of if_while_hand */



/*
 * Process for statement
 */

for_hand(condtype)
int condtype;
{
int tokbuf[BUFSIZE];
char typebuf[BUFSIZE];
int i;
for ( i = 0; i < 100; i++) {
  tokbuf[i] = 0;
  typebuf[i] = '\0';
}
i = 0;
 while ( Token != SEMI || Type != OP) {
  lex();
}
  typebuf[i] = OP;
  tokbuf[i++] = OPENPAR;
  Token = NULL;
while ( Token != SEMI || Type != OP) {
  lex();
  typebuf[i] = Type;
  tokbuf[i++] = Token;
}
typebuf[i] = OP;
tokbuf[i++] = CLSPAR;
if ( Assignments == TRUE)
  cond_hand(tokbuf, typebuf, condtype);
if ( Null_stmnts == TRUE)
  null_hand(condtype);
} /* end of for_hand */
```

```
/*
 * Process conditional part of if, while, for, do-while
 */

cond_hand(tokbuf, typebuf, condtype)
int *tokbuf;
char *typebuf;
int condtype;
{
int i;
int assigncnt = 0;
int relopcnt = 0;
int logopcnt = 0;
int parencnt = 0;
int optype[BUFSIZE];

for ( i = 0; i <= BUFSIZE; i++ ) optype[i] = 0;

for ( i = 0; (tokbuf[i] != 0)  (typebuf[i] == CONST); i++ ) {
  if (typebuf[i] == OP) {
   switch(tokbuf[i]) {
    case EQ:
        if ( tokbuf[i-1] != SNGLQT ) {
          assigncnt++;
          optype[i] = ASSIGN;
        }
        break;

    case NE:
    case LT:
    case LE:
    case EQEQ:
    case GT:
    case GE:
        relopcnt++;
        optype[i] = RELOP;
        break;

    case LOGOR:
    case LOGAND:
        logopcnt++;
        optype[i] = LOGOP;
        break;
   }
  }
}

 i = 0;
 if ( relopcnt == 0 && assigncnt == 0 )
   printf("line %3d: - No relational operators in \"%s\".\n",Linenum, keytable[condtype].keyword);
 else if ( relopcnt == 0 && assigncnt > 0 ) {
   if (strcmp(keytable[condtype].keyword, 'for') == 0)
   printf("line %3d: - Misuse of assignment operator ( = ) in \"%s\" ( try <= ).\n',
Linenum, keytable[condtype].keyword);
   else
   printf( 'line %3d: - Misuse of assignment operator ( = ) in \"%s\" ( try == ). n',
Linenum, keytable[condtype].keyword);
 }
 else if ( relopcnt == logopcnt && assigncnt > 0 ) {
```

```
    if (strcmp(keytable[condtype] keyword,"for") == 0)
    printf("line %3d: - Misuse of assignment operator ( = ) in \"%s\" ( try <= ).\n",
Linenum, keytable[condtype].keyword);
    else
    printf("line %3d: - Misuse of assignment operator ( = ) in \"%s\" ( try == ).\n",
Linenum, keytable[condtype].keyword);
  }
  else if ( relopcnt > 0 && assigncnt > 0 ) {
  i = 0;
    while ( assigncnt > 0 && relopcnt > 0 ) {
      if (typebuf[i] == OP ) {
        switch(optype[i]) {
          case ASSIGN:
                i++;
                if (relopcnt > 0) {
                  while ( optype[i] != RELOP ) {
                    if ( tokbuf[i] == OPENPAR && typebuf[i] == OP ) parencnt--;
                    if ( tokbuf[i] == CLSPAR && typebuf[i] == OP ) parencnt++;
                    if ( tokbuf[i] == EQ && typebuf[i] == OP ) assigncnt--;
                    i++;
                  }
                if ( parencnt <= 0 )
                  printf( 'line %3d: - Operator precedence error involving assignment in \"%s\" \n',
              Linenum, keytable[condtype].keyword);
                }
                parencnt = 0;
                assigncnt--;
                relopcnt--;
                break;

          case RELOP:
                i++;
                if ( assigncnt > 0 ) {
                  while ( optype[i] != ASSIGN ) {
                    if ( tokbuf[i] == OPENPAR && typebuf[i] == OP ) parencnt++;
                    if ( tokbuf[i] == CLSPAR && typebuf[i] == OP ) parencnt--;
                    i++;
                  }
                 if ( parencnt <= 0 )
                    printf("line %3d: - Operator precedence error involving assignment in \"%s\".\n",
                Linenum, keytable[condtype] keyword);
                }
                parencnt = 0;
                assigncnt--;
                relopcnt--;
                break;
          default:
                i++;
        }
      }
      else if (typebuf[i] == ID )
        i++; /* do nothing (call pointer checker later) */
      else if (typebuf[i] == KEYWD )
        i++; /* do nothing (call scanf checker later) */

    }
  }
  }

/*
```

```
 * Look for null statement in wrong place after if, while, for.
 */

null_hand(condtype)
int condtype;
{
int loopcnt = 0;
if ( condtype == FOR ) {
  while ( Token != CLSPAR || Type != OP)
    lex();
}
lex();
while ( (Token != NL) && ( Token != SEMI) || Type != OP) {
  loopcnt++;
  lex();
}
if ( Token == NL && Type == OP) loopcnt++;
switch(condtype) {

  case WHILE:
        if (loopcnt == 0 && Dostack[Doptr - 1] != DO)
        printf("line %3d: - Null statement (;) after \"%s\".\n", Linenum, keytable[condtype].keyword);
        else if (loopcnt == 0 && Dostack[Doptr - 1] == DO)
        Dostack[--Doptr] = NULL;
        break;

  case IF:
  case FOR:
        if (loopcnt == 0)
        printf("line %3d: - Null statement (;) after \"%s\".\n", Linenum, keytable[condtype].keyword);
        break;
}
} /* end of null_hand */

/*
 * Look for missing break statements in case constructs.
 */

case_hand()
{
if (Lastcasetok == COLON || Lastcasetok == BREAK || Switchflag == ON)
  Switchflag = OFF;
else
  printf("line %3d: - No break at end of \"case\".\n", Linenum - 1);
} /* end of case_hand */


/*
 * Process scanf() statement
 */

scanf_hand()
{
int tokbuf[BUFSIZE];
int qtcnt = 0;
char typebuf[BUFSIZE];
char vartype;
char name[100];
int init;
int i;
```

```
for ( i = 0; i < 100; i++ ) {
  tokbuf[i] = 0;
  typebuf[i] = '\0';
}
i = 0;
lex();
while (qtcnt < 2) {
  if (Token == DBLQT && Type == OP && Lastok != BKSLASH)
    qtcnt++;
  lex();
}
if ( Token != COMMA) printf("Error in scanf format\n");
lex();
if ( Type == ID ) strcpy(name, Charbuff);
while ( !(Token == CLSPAR && Type == OP) ) {
  while ( !( (Token == COMMA && Type == OP) | (Token == CLSPAR && Type == OP ) ) ) {
    tokbuf[i] = Token;
    typebuf[i++] = Type;
    lex();
    if ( Type == ID ) strcpy(name, Charbuff);
  }
  i = 0;
  while ( typebuf[i] != ID )
    i++;
  find_entry(&vartype,tokbuf[i],name, &init);
  if (vartype == SINGLE && (tokbuf[0] != AND | typebuf[0] != OP) )
    printf("line %3d:  Incorrect address specification for \"%s\" in scanf\(\),\n\t\ttry \"&%s\".\n",
 Linenum, name, name);
  else if ( vartype == ARRAY && !( (i == 0 && tokbuf[1] != OPENBRAK)
                    | (tokbuf[0] == AND && tokbuf[2] == OPENBRAK) ) )
    printf("line %3d:  Incorrect address specification for \"%s\" in scanf\(\),\n\t\ttry \"%s\" or \"&%s[n]\".\n",
 Linenum, name, name, name);
  else if ( vartype == PTR ) {
        if ( i != 0   typebuf[0] != ID )
          printf("line %3d:  Incorrect address specification for \"%s\" in scanf\(\),\n\t\ttry \"%s\".\n",
 Linenum, name, name);
          else
            if ( Pointers == TRUE ) point_hand(tokbuf[0], name);
    }
  else if ( vartype == PTRARRAY && !( (i == 0 && typebuf[0] == ID )
                          (i == 0 && tokbuf[1] == OPENBRAK)
                          (tokbuf[0] == STAR && typebuf[1] == ID
                        && tokbuf[2] != OPENBRAK) ) )
    printf("line %3d:  Incorrect address specification for \"%s\" in scanf\(\),\n\t\ttry \"*%s\" or \"%s[n]\".\n",
 Linenum, name, name, name);
 for ( i = 0; i < 100; i++ ) {
  tokbuf[i] = 0;
  typebuf[i] = '\0';
 }
 i=0;
 if ( Token == CLSPAR && Type == OP ) break;
 lex();
 if ( Type == ID ) strcpy(name, Charbuff);
 }
}
/* end of scanf_hand */

/*
 * Look for uninitialized pointers
 */
```

```
point_hand(token,name)
int token;
char *name;
{
char vartype;
int init;
  find_entry(&vartype, token, name, &init);
  if ( vartype == PTR ) {
   if ( init == NO ) {
       lex();
         if ( Token == EQ ) {
          set_init(token, name);
         }
         else
         printf("line %3d:  Possible uninitialized pointer - \"%s\" \n",Linenum, name);
   }
  }
  else {
  return;
  }
} /* end of point_hand */


/*
 * set initialized flag to YES
 */

set_init(token,name)
int token;
char *name;
{
int count;
int search = token;
int searchblock;
if ( (token == Symtable[token].newtoken) &&
    (strcmp(name, Symtable[token].id) == 0) &&
    (Block == Symtable[search].block) )
  {
   Symtable[search].init = YES;
   return(Symtable[search].newtoken);
  }
else {
  for (searchblock = Block; searchblock >= 0; searchblock--) {
    count = 0;
    search = token + 1;
    while ( ((strcmp(name, Symtable[search % HASHSIZE].id) != 0) ||
          (Block != Symtable[search % HASHSIZE].block)) &&
          ( count <= HASHSIZE) ) {
      count++;
      search++;
    }
  }
}
Symtable[search].init = YES;
return(Symtable[search].newtoken);
} /* end of set_init */

/* end of program */
```

ANALYZING "C" PROGRAMS FOR COMMON ERRORS

by

Dennis M. Frederick

B.S., University of Missouri, 1970

AN ABSTRACT OF A REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

COMPUTER SCIENCE

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1988

ANALYZING "C" PROGRAMS FOR COMMON ERRORS

ABSTRACT

Several programming errors commonly made by users of the C Programming Language escape detection by the C compiler. These errors in usage result in statements which are syntactically and semantically correct, but which are usually not what the programmer intended and cause incorrect program execution. Several debugging tools are available for C Language, but none of them detect these commonly made errors. The focus of this investigation is the development of a tool to analyze C programs to detect and report these errors. Specifically, it detects inappropriate uses of the assignment operator and operator precedence errors in control constructs, unintended null statements, omission of break statements from switch/case constructs, improper address specifications in scanf() function calls, and uninitialized pointers.