C-Helper: C Latent-error Static/Heuristic Checker for Novice Programmers

Kota Uchida[1] and Katsuhiko Gondow[2]
[1]*Cyboze, Inc., Tokyo, Japan*
[2]*Department of Computing Science, Tokyo Institute of Technology, Tokyo, Japan*

Keywords: Programming Education, C Static Checker, Compiler Warning Messages, Latent Errors, Heuristics, Novice Programmer.

Abstract: For better programming language education, it is crucial to make compiler warning messages more understandable for novice programmers. Unfortunately, however, Kojima's research showed warning messages in commercial-level compilers like GCC are still difficult to understand, and the commercial-level compilers tend not to emit how to modify programs to correct the problems. Furthermore, we found that they also tend not to handle latent errors. To solve this problem, by using a heuristic approach, we propose a novel C static checker called C-Helper, that aims to emit more direct error messages understandable for novices to correct wrong programs, and also aims to handle latent errors. Our preliminary evaluation shows that C-Helper was positively evaluated, although our heuristic approach increased false-positives.

1 INTRODUCTION

As Kojima et al. indicated (Yoshitaka Kojima, 2015), warning messages in commercial-level compilers like GCC are difficult for novice programmers; the compilers and static checkers, except Splint (Inexpensive Program Analysis Group, 2015), emit little guidance to correct wrong programs; technical terms used by Splint are difficult for novices. Thus this also indicates it is very challenging to provide novices with error messages understandable for them in plain terms.

For a motivating example, the line 5 in Listing 1 is a typical mistake by novices where a string is illegally assigned to a variable of type `char`. The reason why this mistake occurs is because the assignment statement at line 5 is very similar to the array initialization at line 3 in Listing 1, which is perfectly legal. Even worse, for Listing 1, GCC-4.7.2 emits the message "test.c:5:13: warning: assignment makes integer from pointer without a cast", which is very confusing for novices. Instead, we would like to provide a more direct message "String cannot be stored in an element of `char` array variable. Consider to use `strcpy`". This paper proposes a novel static checker called C-Helper, which does emit this message.

Another important issue is to cope with latent errors. Here we use the term "latent error" as a program for which commercial-compilers emit no error messages, but the intentions of (novice) programmers are contrary to the program. For another motivating example, at line 6 in Listing 2, `sizeof` operator is applied to a pointer variable `a` that points a dynamically allocated array. The intention of novice programmers is often likely to obtain the size of the dynamically allocated array (i.e., 20), not the size of pointer (i.e., 4 in ILP32 platforms), while the actual value of `sizeof(a)` is 4. The reason why this mistake occurs is because `sizeof` operator applied to statically allocated memory like the line 5 in Listing 3 produces the same result (i.e., 20) as the programmer's intention.

Listing 1: Example of assigning string to `char` array variable.

```
1 int main (void)
2 {
3     char arr1[20]="This is legal";
4     char arr2[20];
5     arr2[20]="This is illegal";
6 }
```

For Listing 2, our C-Helper emits "The result of sizeof(a) is 4, not 20. Is it really intended?", Of course, there are some cases the intention is to obtain 4, but we use a *heuristic* or assumption that in most cases the intention of novices is to obtain 20.

Programs with a latent error are categorized into the following three groups:

- Bad coding styles: e.g., unbalanced indentation

Listing 2: Example of applying `sizeof` to dynamically allocated arrays.

```
#include <stdlib.h>
int main (void)
{
  int i;
  char *a = malloc (20);
  for (i=0; i < sizeof(a)/sizeof(a[0]); i++) {
      a[i]='\0';
  }
}
```

Listing 3: Example of applying `sizeof` to statically allocated arrays.

```
int main (void)
{
  int i;
  char a[20];
  for (i=0; i < sizeof(a)/sizeof(a[0]); i++) {
      a[i]='\0';
  }
}
```

(E9 in Appendix).

- Dynamic errors: e.g., memory leak and buffer overflow (E10 and E11 in Appendix).
- Pitfalls in programming languages: e.g., the above `sizeof` misuse (E12 through E15 in Appendix).

In this paper, we design and implement a novel C static checker called C-Helper that aims to emit more direct error messages understandable for novices to correct wrong programs, and also aims to handle 15 latent errors including all the above three groups, using some heuristics and assumptions.

The main contributions of this research are as follows:

- We have provided the first prototype implementation as a C static checker using heuristic rules that can also cope with latent errors (Sec. 3.2). The full source code of C-Helper is publicly available at the C-Helper homepage (uchan-nos, 2015).
- Our preliminary evaluation shows that the 8 students positively evaluated C-Helper, although our heuristic approach increased false-positives (Sec. 4).
- We obtained several important findings: e.g., to more efficiently implement static analyzers, we need a more powerful and abstract language; we experienced that Java and Eclipse/CDT is not enough for our purpose (Sec. 3.3).

2 RELATED WORK

There are several papers on compiler warning messages understandable for novices, summarized in this section. To our knowledge, however, none of them tried to provide novices with understandable messages for latent-errors in C.

- Splint (Inexpensive Program Analysis Group, 2015) is a powerful C static checker, especially for professional programmers, since Splint emits more informative messages to correct problems than other commercial-level compilers (Yoshitaka Kojima, 2015). Although Splint emits messages for some latent-errors (e.g., E10, E11, E15, but not E9, E12[1], E13, E14 in Appendix), Splint's messages are unfortunately less understandable for novices (Yoshitaka Kojima, 2015),
- Thetis (Freund and Roberts, 1996) is an integrated development environment for C with C interpreter, understandable error reporting, run-time error detection, debugging/visualization tools. A special feature of Thetis is *strengthened syntactic restrictions*, where common code fragments often mistakenly used by novices are regarded as errors by Thetis, even though they are perfectly legal in the C standard. The most common example is `if(i=0)`···, where the assignment operator = is often misused instead of the relational operator == by novices. The concept of this Thetis's feature is very similar to C-Helper's latent-error detection, but the details are unknown since Thetis implementation is not available now.
- C-Tutor (Song et al., 1997) is a C program analysis tool, which provides novices with understandable messages to correct their wrong programs. C-Tutor combines static and dynamic analysis techniques, and extracts novice's intentions using sample programs. Thus, C-Tutor needs sample programs given by teachers, while C-Helper doesn't.
- CX-checker (Osuka et al., 2012) is a C coding style checker. While the current C-Helper only checks unbalanced indentation as a common coding style, CX-checker copes with various styles, which are highly customizable using XPath, DOM and wrapper API. CX-checker does not cover syntax/semantic/latent errors.
- Kummerfeld's method (Kummerfeld and Kay, 2003) catalogs some common C/C++ compiler error messages, typical code examples for them, and

[1]Splint emits unrelated messages, but nothing for `sizeof(array)`.

their possible corrections as a Web-based reference guide. This method might be effective, but its maintenance cost is very high since the catalog must be updated whenever new compilers are released. This method covers syntax errors, bot doesn't cover latent-errors.

- There are programming environments proposed to support novices in other languages like DrRacket (Marceau et al., 2011b; Marceau et al., 2011a) for Scheme, and BlueJ (Kölling et al., 2003) Expresso (Hristova et al., 2003) Gauntlet (Flowers et al., 2004) for Java. Although the concept of more understandable messages is almost the same as C-Helper, the language differs. Due to the language differences, in C-Helper, we needed to implement alias analysis, for example, to detect memory leak and buffer overflow, which is not necessary in Scheme and Java.

3 DESIGN AND IMPLEMENTATION

3.1 Design

3.1.1 Target Errors and Heuristics

Although there are various programs that novices often mistake, we decided to focus on 15 typical errors (E1 through E15 in Appendix) in the first implementation. The 15 errors were collected from a C programming forum (RemicalSoft, 2015), including 2 syntax errors, 6 semantic errors and 7 latent errors.

To analyze the 15 errors, we use heuristics in the following sense.

- We assume some novice's programming tendencies. For example, as mentioned in Sec. 1, we assume that novices tend to use `sizeof(a)` in Listing 2 to obtain the size of dynamically allocated array.
- We also assume that novices tend not to use complex expressions like `if((p=malloc(512))!=NULL)` This assumption supports the statement-level granularity in C-Helper (Sec. 3.1.2).

3.1.2 The Granularity is a Statement, Not Expression

We decided to use a statement as the analysis granularity in C-Helper, not an expression, since

- The order of evaluation of the subexpressions within an expression is unspecified in the C standard (i.e., unspecified behavior in C), which means different compilers evaluate the expressions in different orders. Thus, it is impossible to analyze C expressions in a portable way.
- The granularity of expression makes it much harder to implement C-Helper, since the syntax of C expressions is far more complex than simple WHILE language in the textbook of static analysis (Nielson et al., 2004).

3.1.3 Static Analyzers to Implement

As discussed in Sec. 3.2, we utilize the abstract syntax trees (AST) generated by Eclipse/CDT, but we need to implement our own static analyzers, since Eclipse/CDT does not generate static analysis information enough to our need. In C-Helper, we decided to implement the following:

- Control flow graph generator.
- Reaching definition analyzer.
- Alias analyzer.

Reaching definition analysis is required to detect `sizeof(a)` in Listing 2 for example, while alias analysis is required to detect memory leak like Listing 13.

We implement them by originally extending, to fit the C language, the iterative algorithms in the textbook of static analysis (Nielson et al., 2004), in a flow-sensitive and context-insensitive way.

Also we implement them to analyze only simple variable definitions and assignment statements[2] as nodes in control flow graphs, since we assume that novices tend not to use complex expressions, as discussed in Sec. 3.1.1 and Sec. 3.1.2 to simplify the implementation of C-Helper.

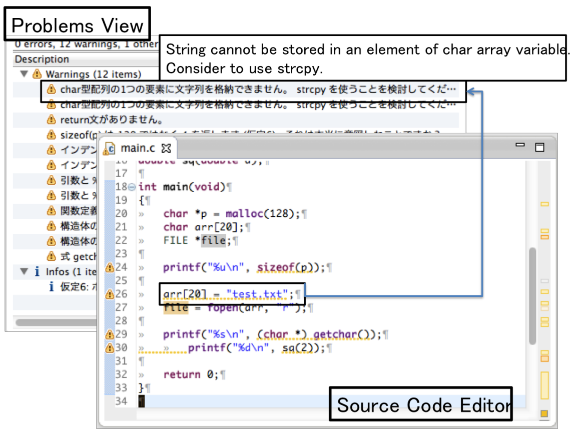
3.2 Implementation

3.2.1 Overview of C-Helper

We implemented C-Helper in Java as a plugin of Eclipse/CDT[3] (Fig. 2). The code size of C-Helper is about 10,000 lines in Java. Fig. 1 shows a screenshot of C-Helper; the error messages emitted by C-

[2]To be more precise, expression statement whose outermost operator is =.

[3]CDT (abbr. of C/C++ Development Tooling) is an Eclipse plugin including C/C++ parser, type analysis, syntax highlighting, etc.


Problems View
String cannot be stored in an element of char array variable. Consider to use strcpy.
Source Code Editor
main.c

0 errors, 12 warnings, 1 other
Description
Warnings (12 items)
char型配列の1つの要素に文字列を格納できません。 strcpy を使うことを検討してくだ…
char型配列の1つの要素に文字列を格納できません。 strcpy を使うことを検討してくだ…
returnが文がありません。
sizeof(p
インデン
インデン
引数と
引数と
関数定義
構造体の
構造体の
式 getch
Infos (1 ite
仮定6:

```
int main(void)
{
    char *p = malloc(128);
    char arr[20];
    FILE *file;

    printf("%u\n", sizeof(p));

    arr[20] = "test.txt";
    file = fopen(arr, "r");

    printf("%s\n", (char *) getchar());
        printf("%d\n", sq(2));

    return 0;
}
```

Figure 1: Screenshot of C-Helper.

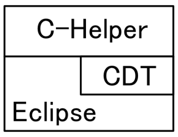


Figure 2: System overview of C-Helper.

Helper appear on Problems View[4][5], and the positions that caused the errors are underlined in Source Code Editor. The users can come and go between the error message and its error position by clicking them.

3.2.2 Processing Flow of C-Helper

The processing flow of C-Helper is as follows (Fig. 3).

1. From Eclipse, C-Helper obtains the source code that the user is editing.
2. C-Helper sends the source code and the standard headers[6], included in the source code to CDT. Note that the standard headers used here are not native system headers. Instead, C-Helper provides the minimal plain headers conformed to the C89 standard, to avoid the problems of compiler-specific extensions.
3. C-Helper receives from CDT, an abstract syntax tree (AST) including the type information and symbol table, and sends the AST to various analyzers in C-Helper.
4. C-Helper receives the result of analysis.

[4]C-Helper is developed for the Japanese novice students, so the error messages are written in Japanese, because they almost always skip over and ignore messages in English.

[5]In current implementation of C-Helper, this needs explicit start by the user; i.e., the messages appear when the user clicks the analysis icon.

[6]The current C-Helper supports `stdlib.h` and `stdio.h`. There is no difficulty to add other headers.

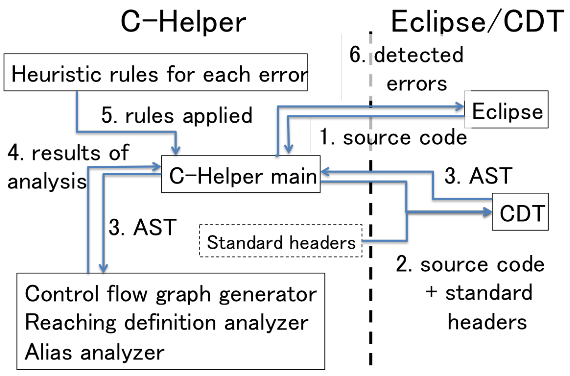


Figure 3: Processing flow of C-Helper.

5. C-Helper applies heuristic rules to the analysis result.
6. C-Helper asks Eclipse to display the detected errors.

3.3 Discussion

Even though we restricted the analysis granularity to a statement, not an expression (Sec. 3.1.2), we found that the implementation of C-Helper is still difficult. One of the reasons is that the syntax of the C language is complex to implement static analyzers. For example, to analyze loops, we have to handle all possible combinations of syntax elements like the statements of `while`, `do-while`, `for`, `if` with `goto`, that behave as loops. Just to implement an AST matching code for the loops, we experienced that the code becomes very large and complex. One idea to alleviate this problem is to use an intermediate form like GIMPLE/RTL in GCC, or to develop yet another more abstract form to simplify static analyzers. This is one of our future work.

Another issue is a problem of redundant downcast code. Listing 4 shows an example of this problem, which is a Java code fragment in C-Helper. Listing 4 traverses an AST to check if the value assigned to a variable is a pointer returned by `malloc`. Even though `instanceof` is used before downcast (e.g., at the line 3 in Listing 4) to check if the downcast is safe or not, the corresponding downcast is also required (e.g., `(IASTCastExpression)rhs` at the line 4 in Listing 4). This code pattern appears so often (e.g., the line 6/7 and 8/9 in Listing 4), so the code becomes very redundant.

3.4 Limitations

In the current C-Helper, there are several limitations, which cause false-positives and false-negatives (see also Sec. 4). For example, the current C-Helper only

Listing 4: Java code fragment in C-Helper to check if the value assigned to a variable is a pointer returned by `malloc`.

```
for (AssignExpression assign : assigns) {
 IASTNode rhs = assign.getRHS();
 while (rhs instanceof IASTCastExpression) {
   rhs = ((IASTCastExpression)rhs).getOperand();
 }
   if (rhs instanceof IASTFunctionCallExpression) {
    IASTFunctionCallExpression fce = (IASTFunctionCallExpression)rhs;
      if (fce.getFunctionNameExpression() instanceof IASTIdExpression) {
      String funcname=((IASTIdExpression)fce.getFunctionNameExpression()).getName().toString();
          if (funcname.equals("malloc")) {
              beginnerExpectingValues.add(fce.getArguments()[0].getRawSignature());
}}}}
```

Table 1: Evaluation by 8 novice students.

C-Helper feature	# good	# unnecessary	# bad
Detecting unbalanced indentation	2	1	0
Suggesting headers of the standard library not included	1	0	0
Detecting the lack of `return` statement in non-`void` function	3	0	0
Messages in Japanese	3	0	0

Table 2: Result of applying C-Helper to Kojima's benchmark.

error category	# programs	# false-positives	# true-positives
pointer/array	31	16	0
conditional	16	0	0
function	16	15	5
variable	14	3	0
expression/statement	13	5	1
total	90	39	6

handles simple `malloc` and `realloc` calls[7] (as discussed in Sec. 3.1.1), so it cannot handle calls like `if((p=malloc(512))!=NULL)`.

Another limitation comes from the difficulty of handling C pointers. As is the case with other static analyzers and garbage collectors for C, the alias analysis in C-Helper is incomplete due to this problem. It is difficult to cope with C pointers, since pointer arithmetic and casting in C makes it possible for a variable to point to <u>any</u> location in memory.

4 PRELIMINARY EVALUATION

4.1 Questionnaire in C Language Tutorial

The first author conducted a tutorial on the programming language C for 18 students[8] who learn the programming language for the first time. The tutorial consisted of 5 lectures covering "hello, world", variables, control statements and simple function definition. The first author asked them to use C-Helper in the tutorial, and also asked them to tell us which features in C-Helper are good/unnecessary/bad in free format. Out of 18, 8 students replied their answers (Table. 1). The result indicates that the students positively evaluated C-Helper, since there is only one "unnecessary" and no "bad" answers, while there are the total 9 "good" answers. It is not surprising that they answered only 3 errors out of 15 that C-Helper can detect, since the tutorial did not cover a little bit advanced topics like structures and `malloc`. It is interesting that 3 students answered "good" for messages in Japanese; as we expected, Japanese students tend to prefer messages in Japanese to ones in English.

4.2 Applying C-Helper to Kojima's Benchmark

Before we implemented C-Helper, we expected our heuristic approach increases false-positives (that is, the case where some error message is emitted, but it is wrong). To measure this objectively, we applied C-Helper to Kojima's benchmark (Yoshitaka Kojima,

[7] `realloc` is not supported in the current C-Helper.

[8] Japanese undergraduates in Tokyo Institute of Technology, whose majors are not limited to computer science.

2015), which is a set of 90 small programs in C language that novice programmers often mistake.

The result is shown in Table. 2; out of 90, there are 39 false-positives and 6 true-positives. This result shows that as we expected, our heuristic approach increases false-positives. The main reasons are because the current C-Helper supports only 15 errors (see Appendix), and also because the benchmark includes many complex expressions that the current C-Helper cannot handle like `if((p=malloc(512))!=NULL)`. We have a plan to extend and refine our heuristic rules to decrease false-positives while preserving the merits of C-Helper.

5 CONCLUSIONS

In this paper, we proposed a novel C static checker called C-Helper, that aims to emit more direct error messages understandable for novices to correct wrong programs, and also aims to handle latent errors. Our preliminary evaluation shows that C-Helper was positively evaluated, although our heuristic approach increased false-positives.

As future work, we have a plan to extend and refine our heuristic rules to decrease false-positives while preserving the merits of C-Helper.

REFERENCES

Flowers, T., Carver, C., and Jackson, J. (2004). Empowering students and building confidence in novice programmers through gauntlet. In *Frontiers in Education, 2004. FIE 2004. 34th Annual*, pages T3H/10–T3H/13 Vol. 1.

Freund, S. N. and Roberts, E. S. (1996). Thetis: An ansi c programming environment designed for introductory use. In *Proceedings of the Twenty-seventh SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '96, pages 300–304, New York, NY, USA. ACM.

Hristova, M., Misra, A., Rutter, M., and Mercuri, R. (2003). Identifying and correcting java programming errors for introductory computer science students. *SIGCSE Bull.*, 35(1):153–156.

Inexpensive Program Analysis Group, University of Virginia, D. o. C. S. (2015). Splint annotation-assisted lightweight static checking. http://www.splint.org/, [Online; accessed 14-Oct-2015].

Kölling, M., Quig, B., Patterson, A., and Rosenberg, J. (2003). The bluej system and its pedagogy. *Computer Science Education*, 13(4):249–268.

Kummerfeld, S. K. and Kay, J. (2003). The neglected battle fields of syntax errors. In *Proceedings of the Fifth Australasian Conference on Computing Education - Volume 20*, ACE '03, pages 105–111, Darlinghurst, Australia, Australia. Australian Computer Society, Inc.

Marceau, G., Fisler, K., and Krishnamurthi, S. (2011a). Measuring the effectiveness of error messages designed for novice programmers. In *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education*, SIGCSE '11, pages 499–504, New York, NY, USA. ACM.

Marceau, G., Fisler, K., and Krishnamurthi, S. (2011b). Mind your language: On novices' interactions with error messages. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2011, pages 3–18, New York, NY, USA. ACM.

Nielson, F., Nielson, H. R., and Hankin, C. (2004). *Principles of Program Analysis*. Springer; Corrected edition.

Osuka, T., Kobayashi, T., Atsumi, N., Mase, J., Yamamoto, S., Suzumura, N., and Agusa, K. (2012). CX-checker: A flexibly customizable coding checker for C. *Journal of Information Processing Society of Japan*, 53(2):590–600.

RemicalSoft (2015). Forum of any questions in the programming language C. http://dixq.net/forum/viewforum.php?f=3, [Online; accessed 14-Oct-2015].

Song, J. S., Hahn, S. H., Tak, K. Y., and Kim, J. H. (1997). An intelligent tutoring system for introductory c language course. *Comput. Educ.*, 28(2):93–102.

uchan-nos (2015). C-helper: A programming environment for beginners of C programming language. https://github.com/uchan-nos/c-helper, [Online; accessed 14-Oct-2015].

Yoshitaka Kojima, Yoshitaka Arahori, K. G. (2015). Investigating the difficulty of commercial-level compiler warning messages for novice programmers. In *Proceedings of the 8th International Conference on Computer Supported Education*, CSEDU! 2015, pages 483–490.

APPENDIX

The following is the list of 15 errors checked by the current C-Helper, categorized into three groups: (1) 2 syntax errors, (2) 6 semantic errors, (3) 7 latent errors.

Listing 5: Example of extra semicolon in function definition.

```c
#include <stdio.h>
int main(void);
{
}
```

Syntax Errors

E1: Extra semicolon in function definition

For Listing 5, C-Helper emits "test.c:2:16: Don't place semicolon ';' for function definition"[9], while GCC-4.7.2 emits "test.c:3:1: error: expected identifier or '(' before { token".

E2: The lack of semicolon in structure definition

For Listing 6, C-Helper emits "test.c:5:1: Place semicolon for structure definition", while GCC-4.7.2 emits "test.c:6:1: error: expected ';', identifier or '(' before 'int'"[10].

Semantic Errors

E3: Assigning string to `char` array variable

For Listing 1, C-Helper emits "String cannot be stored in an element of `char` array variable. Consider to use `strcpy`", while GCC-4.7.2 emits "test.c:5:13: warning: assignment makes integer from pointer without a cast".

E4: Parameter mismatch in `printf`

For Listing 7, C-Helper emits " argument is of type 'signed int', but format '%s' expects argument of type `char` pointer (format '%d' can be used for 'signed int'). "[11], while GCC-4.7.2 emits "test.c:5:5: warning: format '%s' expects argument of type char *, but argument 2 has type int".

E5: The lack of `return` statement in non-`void` function

For Listing 8, C-Helper emits "No return statement", while GCC-4.7.2 emits "foo.c:5:1: warning: control reaches end of non-void function",

E6: Comparing `char` and string

For Listing 9, C-Helper emits "character and string cannot be compared (enclose with ', not " for a character)", while GCC-4.7.2 emits "foo.c:7:17: warning: comparison between pointer and integer".

E7: Calling `scanf` with non-pointer parameters

For Listing 10, C-Helper emits "Specify pointer (Obtain the pointer to variable by adding & (&data))", while GCC-4.7.2 emits, with `-Wall` option, "foo.c:5:5: warning: format %d expects argument of type int *, but argument 2 has type int".

E8: Headers of the standard library not included

[9]In the current C-Helper, all messages are written in Japanese, which are translated in English here.

[10]Note that the position `test.c:5:1` that C-Helper emits is the end of the structure, while the position `test.c:6:1` that GCC emits is the beginning of `int main(void)`. Thus, C-Helper is better than GCC also in this respect.

[11]The integer promotion rule in C implicitly converts 'signed char' to 'signed int' in an expression.

Listing 6: Example of the lack of semicolon in structure definition.

```
#include <stdio.h>
struct mystruct
{
    char name[64];
}
int main (void)
{
}
```

Listing 7: Example of parameter mismatch in `printf`.

```
#include <stdio.h>
int main (void)
{
    char c = 'A';
    printf ("%s\n", c);
}
```

Listing 8: Example of the lack of `return` statement in non-`void` function.

```
#include <stdio.h>
int func (FILE *fp, int i)
{
    fprintf (fp, "%d\n", i);
}
```

Listing 9: Example of comparing `char` and string.

```
#include <stdio.h>
int main ()
{
    int num = 0, word;
    FILE *fp = fopen ("foo", "r");
    while ((word = fgetc(fp)) != EOF) {
        if ("+" == word) num++;
    }
}
```

Listing 10: Example of calling `scanf` with non-pointer parameters.

```
#include <stdio.h>
int main (void)
{
    int data;
    scanf ("%d\n", data);
    data = (int)(data * 1.05);
    printf ("%d\n", data);
}
```

The C90 standard is still the default mode in many C compilers. In C90, when an undeclared function `foo` is called, the compiler regards its type as `int foo();`. Therefore, when a header of the standard library is not included, all library functions in the header are regarded as of type `int foo();`, which often confuses novices.

For Listing 11, C-Helper emits "strlen is an undeclared function (Specify #include <string.h>)", while GCC-4.7.2 emits "test.c:4:26: warning: incompatible implicit declaration of built-in function strlen". Clang (LLVM-7.0.0) emits "test.c:4:20: note: include the header <string.h> or explicitly provide a declaration for 'strlen'".

Latent Errors

E9: Unbalanced indentation
For Listing 12, C-Helper emits "Unbalanced indentation. Place indentation equivalent to 4 space characters." for the 4th line, while GCC-4.7.2 emits nothing.

E10: Memory leak
For Listing 13, C-Helper emits "Possible memory leak", while GCC-4.7.2 emits nothing. In current implementation of C-Helper, there are several limitations to detect memory leak and buffer overflow, which results in false-positives/negatives (See Sec. 3.4).

E11: Buffer overrun in `fread`
For Listing 14, C-Helper emits "Possible buffer overflow (Specify sizeof(unsigned char) as second argument, and 100 as third argument)", while GCC-4.7.2 emits nothing in compile-time.

E12: Using `sizeof` to dynamically allocated arrays
Novices sometimes mistakenly apply `sizeof` to a pointer variable (`array` in Listing 2) yielding the size of the pointer, where the intention is to obtain the size of dynamically allocated array by `malloc`.
For Listing 2, C-Helper emits, assuming the above novice's intention, "The result of sizeof(array) is 4, not sizeof(int)*128. Is it really intended?", while GCC-4.7.2 emits nothing.

E13: Cast suppressing compiler warnings
Novices sometimes mistakenly use cast to suppress type mismatch warning (`(char *)` in Listing 15, where the format string `%c` should be used).
For Listing 15, C-Helper emits, assuming the above novice's mistake, "You can use %c if the expression getchar() represents a character", while GCC-4.7.2 emits nothing under 32-bit integer environments.

E14: Variable definition in header files
For Listing 16, C-Helper emits "It is better not to define a variable in header files (Instead, use extern declaration (extern int g_x;))", while GCC-4.7.2 emits nothing.

Listing 11: Example of using undeclared function.

```
#include <stdio.h>
int main (void)
{
    printf ("%d\n", (int)strlen("hoge"));
}
```

Listing 12: Example of Unbalanced indentation.

```
void func(char (*name)[M], int n)
{
  int i = 0, j;
      for (j = 0; name[i][j] != '\0'; ++j) {
  // the reset omitted
```

Listing 13: Example of memory leak.

```
#include <stdlib.h>
void func (int cond)
{
    int *p = malloc (128);
    if (cond) {
        free (p);
    }
}
```

Listing 14: Example of buffer overrun in `fread`.

```
#include <stdio.h>
int main (void)
{
  FILE *fpread, *fpwrite;
  unsigned char buf [100];
  int size;
  fpread  = fopen ("foo", "rb");
  fpwrite = fopen ("bar", "wb");
  size = fread (buf, sizeof (unsigned char),
                10000, fpread);
  fwrite (buf, sizeof(unsigned char),
          size, fpwrite);
  fclose (fpread); fclose (fpwrite);
}
```

Listing 15: Example of cast suppressing compiler warnings.

```
#include <stdio.h>
int main (void)
{
    printf ("%s\n", (char *)getchar());
}
```

Listing 16: Example of variable definition in header files.

```
// test.h
int g_x;
```

```
// test.c
#include "test.h"
void func (void)
{
}
```

Listing 17: Example of shadowed identifiers.

```
#include <stdio.h>
int val;
void func (void)
{
    int val = 41;
}
int main (void)
{
    printf ("%d\n", val);
}
```

E15: Shadowed identifiers

For Listing 17, C-Helper emits “Identifier val is shadowed”, while GCC-4.7.2 emits, with the option `-Wall`, nothing; GCC-4.7.2 emits, with the option `-Wshadow`, “test.c:5:9: warning: declaration of val shadows a global declaration.. test.c:2:5: warning: shadowed declaration is here.”.