FH-OÖ Hagenberg/HSD
Betriebssysteme 3, WS 2025
Übung 5

Name:	**Marco Söllinger**	Aufwand in h:	**2**
Mat.Nr:	**s2410306011**	Punkte:	
Übungsgruppe:	**Gruppe 1**	korrigiert:	

Schachtelungsanalyse für C-Programme (24 Punkte)

Bei der Erstellung eines C-Programms kann es vorkommen, dass eine öffnende oder schließende Klammer vergessen oder ein Kommentar nicht abgeschlossen wird. Dies kann zu schwer auffindbaren Syntaxfehlern führen, da der C-Compiler eine völlig andere Klammerungsstruktur annimmt und damit den Überblick verliert!

Erstellen Sie ein Programm `ccheck.c`, das C-Programme analysiert, indem es am Anfang jeder Zeile die einzelnen Schachtelungstiefen angibt, die nach dieser Zeile vorliegen. Die Zeichen {, }, (oder) bzw. /* und */ bewirken hierbei nur dann eine neue Schachtelung, wenn sie nicht in einem Kommentar oder in einer Zeichenkette angegeben sind.

Für nicht ausgeglichene Klammerung ist eine separate Meldung auszugeben!

Verwenden Sie die Standard E/A-Funktionen und lesen sich dazu das Dokument *Standard_EA_Funktionen.pdf* durch!

Der Aufruf des Programmes mit der Testdatei `tempnam.c` ergibt folgende Ausgabe:

```
 1: {0} (0) /*0*/ |#include  "eighdr.h"
 2: {0} (0) /*0*/ |
 3: {0} (0) /*0*/ |int
 4: {0} (0) /*0*/ |main(int argc, char *argv[])
 5: {1} (0) /*0*/ |{
 6: {1} (0) /*0*/ |   int    i;
 7: {1} (0) /*0*/ |   char   *tmpdir=NULL, *praefix=NULL;
 8: {1} (0) /*0*/ |
 9: {1} (1) /*0*/ |   for (i=1 ;
10: {1} (1) /*0*/ |         i<argc ;
11: {2} (0) /*0*/ |       i+=2) {
12: {2} (0) /*0*/ |       if (!strcmp(argv[i], "-t") && i+1 < argc)
13: {2} (0) /*0*/ |          tmpdir = argv[i+1];
14: {2} (0) /*0*/ |       else if (!strcmp(argv[i], "-p") && i+1 < argc)
```


1

```
15: {2} (0) /*0*/ |         praefix = argv[i+1];
16: {2} (0) /*0*/ |      else
17: {3} (0) /*0*/ |      {
18: {3} (0) /*1*/ |         /*fehler_meld ist eine Funktion
19: {3} (0) /*0*/ |         aus der Datei eighdr.h*/
20: {3} (0) /*0*/ |         fehler_meld(FATAL, "usage:  %s [-t tmpdir] [-p praefix]", argv[0]);
21: {2} (0) /*0*/ |   }
22: {2} (0) /*0*/ |
23: {2} (0) /*0*/ |   printf("%s\n", tempnam(tmpdir, praefix));
24: {2} (0) /*0*/ |
25: {2} (0) /*0*/ |   exit(0);
26: {1} (0) /*0*/ |}

-----------------------------
    - Geschweifte Klammerung { } nicht ausgeglichen
```

Versehen Sie Ihren Quelltext mit entsprechenden Kommentaren. Geben Sie weiters Ihre Ergebnisse und alle Dateien elektronisch ab!

Allgemeine Hinweise: Legen Sie bei der Erstellung Ihrer Übung großen Wert auf eine **saubere Strukturierung** und auf eine **sorgfältige Ausarbeitung!** Dokumentieren Sie alle Schnittstellen und versehen Sie Ihre Algorithmen an entscheidenden Stellen ausführlich mit Kommentaren! Testen Sie ihre Implementierungen ausführlich! Geben Sie den **Testoutput** mit ab!

Beispiel 1

Es wird keine Lösungsidee gefordert, deshalb wurde der Code entsprechend kommentiert.

Es ist nur darauf zu achten das speziel Fälle wie escaptes ” beachtet werden müssen.
Wenn das Program nicht eine ganze Zeile auf einmal auslesen kann, dann bricht es ab.
Das macht die line Analyse deutlich einfacher und es sollten noramlerweise nie in einem Code 512 chars in einer Zeile sein.

1.1 Code

ccheck.c

```c
#include <assert.h>
#include <memory.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define BUFFER_SIZE 128

// ----------------- Debug Macros ----------------
// #define DEBUG
// Enable debug output when DEBUG is defined
#ifdef DEBUG

// cyan
#define DBG_COLOR "\x1b[36m"
#define DBG_RESET "\x1b[0m"

#define DEBUG_LOG(fmt, ...)                                                    \
  do {                                                                         \
    fprintf(stderr, DBG_COLOR "[DEBUG] %s:%d:%s(): " fmt DBG_RESET "\n",       \
            __FILE__, __LINE__, __func__, ##__VA_ARGS__);                      \
  } while (0)
#else
#define DEBUG_LOG(...)                                                         \
  do {                                                                         \
  } while (0)
#endif
// ----------------- End Debug Macros ----------------

// ------------------ Error Messages ----------------
static char const *const cErrOpen = "error in fopen";
static char const *const cErrClose = "error in fclose";
static char const *const cErrRead = "error in fgets";
static char const *const cErrUsage =
    "error in commandline -> ./ccheck fileName\n";
static char const *const cErrLineBufferOverflow =
    "error: line buffer overflow, increase BUFFER_SIZE";
// ------------------ End Error Messages ----------------

// -------------------  typedef ----------------
typedef struct {
  int depth_curly; // {}
  int depth_round; // ()
  int depth_comment;
  unsigned int line_number;
  bool in_block_comment;
  bool in_line_comment;
  bool in_string;
  bool in_char;
  bool depth_curly_was_negative;
  bool depth_round_was_negative;
  bool depth_comment_was_negative;
  bool esc;
} State;

typedef enum {
  SKIP_LINE = -1,
  SKIP_NONE = 0,
  SKIP_1_CHAR = 1,
  SKIP_2_CHAR = 2,
```

```
} skipResult;

typedef enum {
  CHR_NONE = 0,
  CHR_BLOCK_COMMENT_START,
  CHR_BLOCK_COMMENT_END,
  CHR_LINE_COMMENT,
  CHR_STRING,
  CHR_CHAR,
  CHR_ESC,
  CHR_CURLEY_OPEN,
  CHR_CURLEY_CLOSE,
  CHR_ROUND_OPEN,
  CHR_ROUND_CLOSE,
} CharType;
// ------------------- End typedef ----------------

// ------------------- static Functions ----------------

// checks if the current state allows skipping characters
// and when in such a state, returns how many characters to skip
// Also checks for exiting such states
static skipResult checkIfCanSkip(State *state, char chr, char next_char) {
  assert(state != NULL);

  if (state->in_line_comment) {
    DEBUG_LOG("In line comment, skipping rest of line");
    return SKIP_LINE;
  }

  if (state->in_block_comment) {
    if (chr == '*' && next_char == '/') {
      DEBUG_LOG("Exiting block comment");
      state->in_block_comment = false;
      state->depth_comment--;
      return SKIP_2_CHAR;
    }
    return SKIP_1_CHAR;
  }

  if (state->in_string) {
    if (chr == '"' && !state->esc) {
      DEBUG_LOG("Exiting string literal");
      state->in_string = false;
      return SKIP_1_CHAR;
    }
    state->esc = (chr == '\\' && !state->esc);
    return SKIP_1_CHAR;
  }

  if (state->in_char) {
    if (chr == '\'' && !state->esc) {
      DEBUG_LOG("Exiting char literal");
      state->in_char = false;
      return SKIP_1_CHAR;
    }
    state->esc = (chr == '\\' && !state->esc);
    return SKIP_1_CHAR;
  }

  return SKIP_NONE;
}

// process a single line and update the state accordingly
static void process_line(char const *const line, State *state) {
  if (line == NULL || state == NULL) {
    return;
  }

  size_t i = 0;
  // while loop so it is clear that increamenting i is done manually
  while (line[i] != '\0') {
    char chr = line[i];
    char next_c = line[i + 1]; // never undefined, as \0 must exist after chr

    // check if we can skip characters due to current state
```

```
    // also handles exiting states
    skipResult skip = checkIfCanSkip(state, chr, next_c);
    if (skip == SKIP_LINE) {
      break;
    }
    if (skip == SKIP_2_CHAR) {
      i += 2;
      continue;
    }
    if (skip == SKIP_1_CHAR) {
      i += 1;
      continue;
    }

    // check for entering states or updating depths
    switch (chr) {
    case '/':
      if (next_c == '*') {
        DEBUG_LOG("Entering block comment at index %zu", i);
        state->in_block_comment = true;
        state->depth_comment++;
        i += 2;
        continue;
      }
      if (next_c == '/') {
        DEBUG_LOG("Entering line comment at index %zu", i);
        state->in_line_comment = true;
        // Dont need to increas i since line stops
      }
      break;
    case '*':
      // This state can only be reached if not in block comment and */ comes
      // This Implies an error
      if (next_c == '/') {
        DEBUG_LOG("Exiting block comment at index %zu", i);
        state->in_block_comment = false;
        state->depth_comment--;
        i += 2;
        continue;
      }
      break;
    case '"':
      DEBUG_LOG("Entering string literal at index %zu", i);
      state->in_string = true;
      state->esc = false;
      break;
    case '\'':
      DEBUG_LOG("Entering char literal at index %zu", i);
      state->in_char = true;
      break;
    case '{':
      DEBUG_LOG("Entering '{' at index %zu", i);
      state->depth_curly++;
      break;
    case '}':
      DEBUG_LOG("Exiting '}' at index %zu", i);
      state->depth_curly--;
      break;
    case '(':
      DEBUG_LOG("Entering '(' at index %zu", i);
      state->depth_round++;
      break;
    case ')':
      DEBUG_LOG("Found ')' at index %zu", i);
      state->depth_round--;
      break;
    default:
      break;
    }

    i += 1;

    // check here so negative depths are recorded even if multiple in one line
    if (state->depth_curly < 0) {
      state->depth_curly_was_negative = true;
    }
```

```
214     if (state->depth_round < 0) {
215       state->depth_round_was_negative = true;
216     }
217     if (state->depth_comment < 0) {
218       state->depth_comment_was_negative = true;
219     }
220   }
221 }
222 
223 // prints the end state of the parser
224 static void print_EndState(State const *const state) {
225   if (state == NULL) {
226     return;
227   }
228   printf("---------------------------------\n");
229   if (state->depth_curly != 0) {
230     printf("    - Geschweifte Klammern { } nicht ausgeglichen\n");
231   }
232   if (state->depth_round != 0) {
233     printf("    - Runde Klammern ( ) nicht ausgeglichen\n");
234   }
235   if (state->depth_comment != 0) {
236     printf("    - Blockkommentare /* */ nicht ausgeglichen\n");
237   }
238   if (state->depth_curly_was_negative) {
239     printf("    - Warning: Geschweifte Klammern waren einmal negative\n");
240   }
241   if (state->depth_round_was_negative) {
242     printf("    - Warning: Runde Klammern waren einmal negative\n");
243   }
244   if (state->depth_comment_was_negative) {
245     printf("    - Warning: Blockkommentare waren einmal negative\n");
246   }
247 
248   if (state->depth_curly == 0 && state->depth_round == 0 &&
249       state->depth_comment == 0 && !state->depth_curly_was_negative &&
250       !state->depth_round_was_negative && !state->depth_comment_was_negative) {
251     printf("    - Alles ausgeglichen!\n");
252   }
253 }
254 
255 // ------------------- End static Functions ----------------
256 
257 int main(int argc, char *argv[]) {
258   if (argc != 2) {
259     printf(cErrUsage);
260     DEBUG_LOG("Invalid number of arguments: %d", argc - 1);
261     return EXIT_FAILURE;
262   }
263   DEBUG_LOG("Given File %s", argv[1]);
264 
265   FILE *inputfile = fopen(argv[1], "r");
266   if (inputfile == NULL) {
267     perror(cErrOpen);
268     DEBUG_LOG("Could not open file %s", argv[1]);
269     return EXIT_FAILURE;
270   }
271 
272   char line[BUFFER_SIZE];
273   State state = {0};
274   while (fgets(line, sizeof(line), inputfile) != NULL) {
275     char *truncated = strchr(line, '\n');
276 
277     // when it was not possible to load the complete line into the buffer
278     // cancle program since general line length should not be over 512
279     // In a programm
280     // Correctly handling line feeds would require handling a lot of edge cases
281     // like /<bufferend overflow>/ or similar
282     if (truncated == NULL) {
283       perror(cErrLineBufferOverflow);
284       DEBUG_LOG("Line buffer overflow detected");
285       fclose(inputfile); // dont check in error state
286       return EXIT_FAILURE;
287     }
288 
289     // update state based on line content
```

```
290     process_line(line, &state);
291
292     state.in_line_comment = false; // reset line comment at EOL
293     // Remove newline character for cleaner output
294     *truncated = '\0';
295     state.line_number++;
296
297     printf("%5d: {%2d} (%2d) /*%2d*/ | %s\n", state.line_number,
298            state.depth_curly, state.depth_round, state.depth_comment, line);
299   }
300   // check if reading ended because of eof or error
301   if (!feof(inputfile) && ferror(inputfile)) {
302     perror(cErrRead);
303     DEBUG_LOG("Error occurred while reading file %s", argv[1]);
304     fclose(inputfile); // check for fclose error
305   }
306
307   // print final state
308   print_EndState(&state);
309
310   // close file and check for error
311   if (fclose(inputfile) != 0) {
312     perror(cErrClose);
313     DEBUG_LOG("Could not close file %s", argv[1]);
314     return EXIT_FAILURE;
315   }
316
317   return EXIT_SUCCESS;
318 }
```

1.2 Test

Zum Testen wurde ein bash script erstellt, welches die verschiedenen Faelle testet.

Die einzelen Terminal outputs wurden in ein File geschrieben.

test.sh

```
1  #!/usr/bin/env bash
2
3  gcc -o ccheck ccheck.c -Wall -Wextra
4
5  echo "Run with no arguments:"
6
7  ./ccheck
8
9  echo "########################################"
10 echo "Run with two argument:"
11
12 ./ccheck test1.txt test2.txt
13
14 echo "########################################"
15 echo "Run with no existing file:"
16
17 ./ccheck noexist.txt
18
19 echo "########################################"
20 echo "Run on test_00_error.txt:"
21
22 ./ccheck test_00_error.txt >test_00_output.txt
23
24 echo "########################################"
25 echo "Run on test_01_error.txt"
26
27 ./ccheck test_01_error.txt >test_01_output.txt
28
29 echo "########################################"
30 echo "Run on test_02_ok.txt"
31
32 ./ccheck test_02_ok.txt >test_02_output.txt
33
34 echo "########################################"
35 echo "Test buffer overflow"
```

```
i=0
: >one_line.txt
while [ "$i" -lt 1000 ]; do
  printf 'A' >>one_line.txt
  i=$((i + 1))
done
printf '\n' >>one_line.txt

./ccheck one_line.txt

echo "########################################"
echo "Run on ccheck.c"

./ccheck ccheck.c >test_04_output.txt
```

Terminal Output

```
flashfish@fedora ~/D/R/F/B/Uebung05 (main)> ./test.sh
Run with no arguments:
error in commandline -> ./ccheck fileName
########################################
Run with two argument:
error in commandline -> ./ccheck fileName
########################################
Run with no existing file:
error in fopen: No such file or directory
########################################
Run on test_00_error.txt:
########################################
Run on test_01_error.txt
########################################
Run on test_02_ok.txt
########################################
Test buffer overflow
error: line buffer overflow, increase BUFFER_SIZE: Success
########################################
Run on ccheck.c
```

test_00_error.txt

```
{
  {

  }
}

(
  (

  )
)

/*
  /*
  (
  {
  // (
  \{
  }
  )
  */
*/

/*
)
*/

}
  }

  {
{

)
  )

```

```
  (
(

/*/
( This is a string
*/

"("

"\" This is still string ) "

"\\" This is no string (
)

/* ( // { */ ( // }
)

This is allowed sind multi line strings exist
" This is a multi line string
   with a parenthesis )
   and a backslash \\ {
"

{{
}}

((
))

"\{"
")"

"//" (
)

\(
\)
```

test_00_output.txt

```
    1: { 1} ( 0) /* 0*/ | {
    2: { 2} ( 0) /* 0*/ |   {
    3: { 2} ( 0) /* 0*/ |
    4: { 1} ( 0) /* 0*/ |   }
    5: { 0} ( 0) /* 0*/ | }
    6: { 0} ( 0) /* 0*/ |
    7: { 0} ( 1) /* 0*/ | (
    8: { 0} ( 2) /* 0*/ |   (
    9: { 0} ( 2) /* 0*/ |
   10: { 0} ( 1) /* 0*/ |   )
   11: { 0} ( 0) /* 0*/ | )
   12: { 0} ( 0) /* 0*/ |
   13: { 0} ( 0) /* 1*/ | /*
   14: { 0} ( 0) /* 1*/ |   /*
   15: { 0} ( 0) /* 1*/ |   (
   16: { 0} ( 0) /* 1*/ |   {
   17: { 0} ( 0) /* 1*/ |   // (
   18: { 0} ( 0) /* 1*/ |   \{
   19: { 0} ( 0) /* 1*/ |   }
   20: { 0} ( 0) /* 1*/ |   )
   21: { 0} ( 0) /* 0*/ |   */
   22: { 0} ( 0) /*-1*/ | */
   23: { 0} ( 0) /*-1*/ |
   24: { 0} ( 0) /* 0*/ | /*
   25: { 0} ( 0) /* 0*/ | )
   26: { 0} ( 0) /*-1*/ | */
   27: { 0} ( 0) /*-1*/ |
   28: {-1} ( 0) /*-1*/ | }
   29: {-2} ( 0) /*-1*/ |   }
   30: {-2} ( 0) /*-1*/ |
   31: {-1} ( 0) /*-1*/ |   {
   32: { 0} ( 0) /*-1*/ | {
   33: { 0} ( 0) /*-1*/ |
   34: { 0} (-1) /*-1*/ | )
   35: { 0} (-2) /*-1*/ |   )
   36: { 0} (-2) /*-1*/ |
```

```
   37: { 0} (-1) /*-1*/ |   (
   38: { 0} ( 0) /*-1*/ | (
   39: { 0} ( 0) /*-1*/ |
   40: { 0} ( 0) /* 0*/ | /*/
   41: { 0} ( 0) /* 0*/ | ( This is a string
   42: { 0} ( 0) /*-1*/ | */
   43: { 0} ( 0) /*-1*/ |
   44: { 0} ( 0) /*-1*/ | "("
   45: { 0} ( 0) /*-1*/ |
   46: { 0} ( 0) /*-1*/ | "\" This is still string ) "
   47: { 0} ( 0) /*-1*/ |
   48: { 0} ( 1) /*-1*/ | "\\" This is no string (
   49: { 0} ( 0) /*-1*/ | )
   50: { 0} ( 0) /*-1*/ |
   51: { 0} ( 1) /*-1*/ | /* ( // { */ ( // }
   52: { 0} ( 0) /*-1*/ | )
   53: { 0} ( 0) /*-1*/ |
   54: { 0} ( 0) /*-1*/ | This is allowed sind multi line strings exist
   55: { 0} ( 0) /*-1*/ | " This is a multi line string
   56: { 0} ( 0) /*-1*/ |    with a parenthesis )
   57: { 0} ( 0) /*-1*/ |    and a backslash \\ {
   58: { 0} ( 0) /*-1*/ | "
   59: { 0} ( 0) /*-1*/ |
   60: { 2} ( 0) /*-1*/ | {{
   61: { 0} ( 0) /*-1*/ | }}
   62: { 0} ( 0) /*-1*/ |
   63: { 0} ( 2) /*-1*/ | ((
   64: { 0} ( 0) /*-1*/ | ))
   65: { 0} ( 0) /*-1*/ |
   66: { 0} ( 0) /*-1*/ | "\{"
   67: { 0} ( 0) /*-1*/ | ")"
   68: { 0} ( 0) /*-1*/ |
   69: { 0} ( 1) /*-1*/ | "//" (
   70: { 0} ( 0) /*-1*/ | )
   71: { 0} ( 0) /*-1*/ |
   72: { 0} ( 1) /*-1*/ | \(
   73: { 0} ( 0) /*-1*/ | \)
---------------------------------
    - Blockkommentare /* */ nicht ausgeglichen
    - Warning: Geschweifte Klammern waren einmal negative
    - Warning: Runde Klammern waren einmal negative
    - Warning: Blockkommentare waren einmal negative
```

test_01_error.txt

```c
#include  "eighdr.h"

int
main(int argc, char *argv[])
{
   int    i;
   char   *tmpdir=NULL, *praefix=NULL;

   for (i=1 ;
        i<argc ;
        i+=2) {
      if (!strcmp(argv[i], "-t") && i+1 < argc)
         tmpdir = argv[i+1];
      else if (!strcmp(argv[i], "-p") && i+1 < argc)
         praefix = argv[i+1];
      else
      {
         /*fehler_meld ist eine Funktion
         aus der Datei eighdr.h*/
         fehler_meld(FATAL, "usage:  %s [-t tmpdir] [-p praefix]", argv[0]);
   }

   printf("%s\n", tempnam(tmpdir, praefix));

   exit(0);
}
```

test_01_output.txt

```
    1: { 0} ( 0) /* 0*/ | #include  "eighdr.h"
    2: { 0} ( 0) /* 0*/ |
```

```
    3: { 0} ( 0) /* 0*/ | int
    4: { 0} ( 0) /* 0*/ | main(int argc, char *argv[])
    5: { 1} ( 0) /* 0*/ | {
    6: { 1} ( 0) /* 0*/ |    int    i;
    7: { 1} ( 0) /* 0*/ |    char   *tmpdir=NULL, *praefix=NULL;
    8: { 1} ( 0) /* 0*/ |
    9: { 1} ( 1) /* 0*/ |    for (i=1 ;
   10: { 1} ( 1) /* 0*/ |         i<argc ;
   11: { 2} ( 0) /* 0*/ |        i+=2) {
   12: { 2} ( 0) /* 0*/ |       if (!strcmp(argv[i], "-t") && i+1 < argc)
   13: { 2} ( 0) /* 0*/ |          tmpdir = argv[i+1];
   14: { 2} ( 0) /* 0*/ |       else if (!strcmp(argv[i], "-p") && i+1 < argc)
   15: { 2} ( 0) /* 0*/ |          praefix = argv[i+1];
   16: { 2} ( 0) /* 0*/ |       else
   17: { 3} ( 0) /* 0*/ |       {
   18: { 3} ( 0) /* 1*/ |          /*fehler_meld ist eine Funktion
   19: { 3} ( 0) /* 0*/ |          aus der Datei eighdr.h*/
   20: { 3} ( 0) /* 0*/ |          fehler_meld(FATAL, "usage:  %s [-t tmpdir] [-p praefix]",
       argv[0]);
   21: { 2} ( 0) /* 0*/ |    }
   22: { 2} ( 0) /* 0*/ |
   23: { 2} ( 0) /* 0*/ |    printf("%s\n", tempnam(tmpdir, praefix));
   24: { 2} ( 0) /* 0*/ |
   25: { 2} ( 0) /* 0*/ |    exit(0);
   26: { 1} ( 0) /* 0*/ | }
---------------------------------
    - Geschweifte Klammern { } nicht ausgeglichen
```

test_02_ok.txt

```
int main(int argc, char *argv[]){

return 0;
}
```

test_02_output.txt

```
    1: { 1} ( 0) /* 0*/ | int main(int argc, char *argv[]){
    2: { 1} ( 0) /* 0*/ |
    3: { 1} ( 0) /* 0*/ | return 0;
    4: { 0} ( 0) /* 0*/ | }
---------------------------------
    - Alles ausgeglichen!
```

test_04_output.txt

```
    1: { 0} ( 0) /* 0*/ | #include <assert.h>
    2: { 0} ( 0) /* 0*/ | #include <memory.h>
    3: { 0} ( 0) /* 0*/ | #include <stdbool.h>
    4: { 0} ( 0) /* 0*/ | #include <stdio.h>
    5: { 0} ( 0) /* 0*/ | #include <stdlib.h>
    6: { 0} ( 0) /* 0*/ | #include <string.h>
    7: { 0} ( 0) /* 0*/ |
    8: { 0} ( 0) /* 0*/ | #define BUFFER_SIZE 128
    9: { 0} ( 0) /* 0*/ |
   10: { 0} ( 0) /* 0*/ | // ----------------- Debug Macros ----------------
   11: { 0} ( 0) /* 0*/ | // #define DEBUG
   12: { 0} ( 0) /* 0*/ | // Enable debug output when DEBUG is defined
   13: { 0} ( 0) /* 0*/ | #ifdef DEBUG
   14: { 0} ( 0) /* 0*/ |
   15: { 0} ( 0) /* 0*/ | // cyan
   16: { 0} ( 0) /* 0*/ | #define DBG_COLOR "\x1b[36m"
   17: { 0} ( 0) /* 0*/ | #define DBG_RESET "\x1b[0m"
   18: { 0} ( 0) /* 0*/ |
   19: { 0} ( 0) /* 0*/ | #define DEBUG_LOG(fmt, ...)
                                                      \
   20: { 1} ( 0) /* 0*/ |   do {
                                                                  \
   21: { 1} ( 1) /* 0*/ |     fprintf(stderr, DBG_COLOR "[DEBUG] %s:%d:%s(): " fmt DBG_RESET "
       \n",       \
   22: { 1} ( 0) /* 0*/ |             __FILE__, __LINE__, __func__, ##__VA_ARGS__);
                        \
   23: { 0} ( 0) /* 0*/ |   } while (0)
   24: { 0} ( 0) /* 0*/ | #else
   25: { 0} ( 0) /* 0*/ | #define DEBUG_LOG(...)
                                                          \
```

```
26: { 1} ( 0) /* 0*/ |   do {                                                        \
27: { 0} ( 0) /* 0*/ |   } while (0)
28: { 0} ( 0) /* 0*/ | #endif
29: { 0} ( 0) /* 0*/ | // ----------------- End Debug Macros ----------------
30: { 0} ( 0) /* 0*/ |
31: { 0} ( 0) /* 0*/ | // ------------------ Error Messages ----------------
32: { 0} ( 0) /* 0*/ | static char const *const cErrOpen = "error in fopen";
33: { 0} ( 0) /* 0*/ | static char const *const cErrClose = "error in fclose";
34: { 0} ( 0) /* 0*/ | static char const *const cErrRead = "error in fgets";
35: { 0} ( 0) /* 0*/ | static char const *const cErrUsage =
36: { 0} ( 0) /* 0*/ |     "error in commandline -> ./ccheck fileName\n";
37: { 0} ( 0) /* 0*/ | static char const *const cErrLineBufferOverflow =
38: { 0} ( 0) /* 0*/ |     "error: line buffer overflow, increase BUFFER_SIZE";
39: { 0} ( 0) /* 0*/ | // ------------------ End Error Messages ----------------
40: { 0} ( 0) /* 0*/ |
41: { 0} ( 0) /* 0*/ | // -------------------  typedef ----------------
42: { 1} ( 0) /* 0*/ | typedef struct {
43: { 1} ( 0) /* 0*/ |   int depth_curly; // {}
44: { 1} ( 0) /* 0*/ |   int depth_round; // ()
45: { 1} ( 0) /* 0*/ |   int depth_comment;
46: { 1} ( 0) /* 0*/ |   unsigned int line_number;
47: { 1} ( 0) /* 0*/ |   bool in_block_comment;
48: { 1} ( 0) /* 0*/ |   bool in_line_comment;
49: { 1} ( 0) /* 0*/ |   bool in_string;
50: { 1} ( 0) /* 0*/ |   bool in_char;
51: { 1} ( 0) /* 0*/ |   bool depth_curly_was_negative;
52: { 1} ( 0) /* 0*/ |   bool depth_round_was_negative;
53: { 1} ( 0) /* 0*/ |   bool depth_comment_was_negative;
54: { 1} ( 0) /* 0*/ |   bool esc;
55: { 0} ( 0) /* 0*/ | } State;
56: { 0} ( 0) /* 0*/ |
57: { 1} ( 0) /* 0*/ | typedef enum {
58: { 1} ( 0) /* 0*/ |   SKIP_LINE = -1,
59: { 1} ( 0) /* 0*/ |   SKIP_NONE = 0,
60: { 1} ( 0) /* 0*/ |   SKIP_1_CHAR = 1,
61: { 1} ( 0) /* 0*/ |   SKIP_2_CHAR = 2,
62: { 0} ( 0) /* 0*/ | } skipResult;
63: { 0} ( 0) /* 0*/ |
64: { 1} ( 0) /* 0*/ | typedef enum {
65: { 1} ( 0) /* 0*/ |   CHR_NONE = 0,
66: { 1} ( 0) /* 0*/ |   CHR_BLOCK_COMMENT_START,
67: { 1} ( 0) /* 0*/ |   CHR_BLOCK_COMMENT_END,
68: { 1} ( 0) /* 0*/ |   CHR_LINE_COMMENT,
69: { 1} ( 0) /* 0*/ |   CHR_STRING,
70: { 1} ( 0) /* 0*/ |   CHR_CHAR,
71: { 1} ( 0) /* 0*/ |   CHR_ESC,
72: { 1} ( 0) /* 0*/ |   CHR_CURLEY_OPEN,
73: { 1} ( 0) /* 0*/ |   CHR_CURLEY_CLOSE,
74: { 1} ( 0) /* 0*/ |   CHR_ROUND_OPEN,
75: { 1} ( 0) /* 0*/ |   CHR_ROUND_CLOSE,
76: { 0} ( 0) /* 0*/ | } CharType;
77: { 0} ( 0) /* 0*/ | // ------------------- End typedef ----------------
78: { 0} ( 0) /* 0*/ |
79: { 0} ( 0) /* 0*/ | // ------------------- static Functions ----------------
80: { 0} ( 0) /* 0*/ |
81: { 0} ( 0) /* 0*/ | // checks if the current state allows skipping characters
82: { 0} ( 0) /* 0*/ | // and when in such a state, returns how many characters to skip
83: { 0} ( 0) /* 0*/ | // Also checks for exiting such states
84: { 1} ( 0) /* 0*/ | static skipResult checkIfCanSkip(State *state, char chr, char
    next_char) {
85: { 1} ( 0) /* 0*/ |   assert(state != NULL);
86: { 1} ( 0) /* 0*/ |
87: { 2} ( 0) /* 0*/ |   if (state->in_line_comment) {
88: { 2} ( 0) /* 0*/ |     DEBUG_LOG("In line comment, skipping rest of line");
89: { 2} ( 0) /* 0*/ |     return SKIP_LINE;
90: { 1} ( 0) /* 0*/ |   }
91: { 1} ( 0) /* 0*/ |
92: { 2} ( 0) /* 0*/ |   if (state->in_block_comment) {
93: { 3} ( 0) /* 0*/ |     if (chr == '*' && next_char == '/') {
94: { 3} ( 0) /* 0*/ |       DEBUG_LOG("Exiting block comment");
95: { 3} ( 0) /* 0*/ |       state->in_block_comment = false;
96: { 3} ( 0) /* 0*/ |       state->depth_comment--;
97: { 3} ( 0) /* 0*/ |       return SKIP_2_CHAR;
98: { 2} ( 0) /* 0*/ |     }
99: { 2} ( 0) /* 0*/ |     return SKIP_1_CHAR;
```

```
100 |   100: { 1} ( 0) /* 0*/ |    }
101 |   101: { 1} ( 0) /* 0*/ |
102 |   102: { 2} ( 0) /* 0*/ |    if (state->in_string) {
103 |   103: { 3} ( 0) /* 0*/ |      if (chr == '"' && !state->esc) {
104 |   104: { 3} ( 0) /* 0*/ |        DEBUG_LOG("Exiting string literal");
105 |   105: { 3} ( 0) /* 0*/ |        state->in_string = false;
106 |   106: { 3} ( 0) /* 0*/ |        return SKIP_1_CHAR;
107 |   107: { 2} ( 0) /* 0*/ |      }
108 |   108: { 2} ( 0) /* 0*/ |      state->esc = (chr == '\\' && !state->esc);
109 |   109: { 2} ( 0) /* 0*/ |      return SKIP_1_CHAR;
110 |   110: { 1} ( 0) /* 0*/ |    }
111 |   111: { 1} ( 0) /* 0*/ |
112 |   112: { 2} ( 0) /* 0*/ |    if (state->in_char) {
113 |   113: { 3} ( 0) /* 0*/ |      if (chr == '\'' && !state->esc) {
114 |   114: { 3} ( 0) /* 0*/ |        DEBUG_LOG("Exiting char literal");
115 |   115: { 3} ( 0) /* 0*/ |        state->in_char = false;
116 |   116: { 3} ( 0) /* 0*/ |        return SKIP_1_CHAR;
117 |   117: { 2} ( 0) /* 0*/ |      }
118 |   118: { 2} ( 0) /* 0*/ |      state->esc = (chr == '\\' && !state->esc);
119 |   119: { 2} ( 0) /* 0*/ |      return SKIP_1_CHAR;
120 |   120: { 1} ( 0) /* 0*/ |    }
121 |   121: { 1} ( 0) /* 0*/ |
122 |   122: { 1} ( 0) /* 0*/ |    return SKIP_NONE;
123 |   123: { 0} ( 0) /* 0*/ |  }
124 |   124: { 0} ( 0) /* 0*/ |
125 |   125: { 0} ( 0) /* 0*/ |  // process a single line and update the state accordingly
126 |   126: { 1} ( 0) /* 0*/ |  static void process_line(char const *const line, State *state) {
127 |   127: { 2} ( 0) /* 0*/ |    if (line == NULL || state == NULL) {
128 |   128: { 2} ( 0) /* 0*/ |      return;
129 |   129: { 1} ( 0) /* 0*/ |    }
130 |   130: { 1} ( 0) /* 0*/ |
131 |   131: { 1} ( 0) /* 0*/ |    size_t i = 0;
132 |   132: { 1} ( 0) /* 0*/ |    // while loop so it is clear that increamenting i is done manually
133 |   133: { 2} ( 0) /* 0*/ |    while (line[i] != '\0') {
134 |   134: { 2} ( 0) /* 0*/ |      char chr = line[i];
135 |   135: { 2} ( 0) /* 0*/ |      char next_c = line[i + 1]; // never undefined, as \0 must exist
      after chr
136 |   136: { 2} ( 0) /* 0*/ |
137 |   137: { 2} ( 0) /* 0*/ |      // check if we can skip characters due to current state
138 |   138: { 2} ( 0) /* 0*/ |      // also handles exiting states
139 |   139: { 2} ( 0) /* 0*/ |      skipResult skip = checkIfCanSkip(state, chr, next_c);
140 |   140: { 3} ( 0) /* 0*/ |      if (skip == SKIP_LINE) {
141 |   141: { 3} ( 0) /* 0*/ |        break;
142 |   142: { 2} ( 0) /* 0*/ |      }
143 |   143: { 3} ( 0) /* 0*/ |      if (skip == SKIP_2_CHAR) {
144 |   144: { 3} ( 0) /* 0*/ |        i += 2;
145 |   145: { 3} ( 0) /* 0*/ |        continue;
146 |   146: { 2} ( 0) /* 0*/ |      }
147 |   147: { 3} ( 0) /* 0*/ |      if (skip == SKIP_1_CHAR) {
148 |   148: { 3} ( 0) /* 0*/ |        i += 1;
149 |   149: { 3} ( 0) /* 0*/ |        continue;
150 |   150: { 2} ( 0) /* 0*/ |      }
151 |   151: { 2} ( 0) /* 0*/ |
152 |   152: { 2} ( 0) /* 0*/ |      // check for entering states or updating depths
153 |   153: { 3} ( 0) /* 0*/ |      switch (chr) {
154 |   154: { 3} ( 0) /* 0*/ |      case '/':
155 |   155: { 4} ( 0) /* 0*/ |        if (next_c == '*') {
156 |   156: { 4} ( 0) /* 0*/ |          DEBUG_LOG("Entering block comment at index %zu", i);
157 |   157: { 4} ( 0) /* 0*/ |          state->in_block_comment = true;
158 |   158: { 4} ( 0) /* 0*/ |          state->depth_comment++;
159 |   159: { 4} ( 0) /* 0*/ |          i += 2;
160 |   160: { 4} ( 0) /* 0*/ |          continue;
161 |   161: { 3} ( 0) /* 0*/ |        }
162 |   162: { 4} ( 0) /* 0*/ |        if (next_c == '/') {
163 |   163: { 4} ( 0) /* 0*/ |          DEBUG_LOG("Entering line comment at index %zu", i);
164 |   164: { 4} ( 0) /* 0*/ |          state->in_line_comment = true;
165 |   165: { 4} ( 0) /* 0*/ |          // Dont need to increas i since line stops
166 |   166: { 3} ( 0) /* 0*/ |        }
167 |   167: { 3} ( 0) /* 0*/ |        break;
168 |   168: { 3} ( 0) /* 0*/ |      case '*':
169 |   169: { 3} ( 0) /* 0*/ |        // This state can only be reached if not in block comment and
      */ comes
170 |   170: { 3} ( 0) /* 0*/ |        // This Implies an error
171 |   171: { 4} ( 0) /* 0*/ |        if (next_c == '/') {
172 |   172: { 4} ( 0) /* 0*/ |          DEBUG_LOG("Exiting block comment at index %zu", i);
173 |   173: { 4} ( 0) /* 0*/ |          state->in_block_comment = false;
```

```
174: { 4} ( 0) /* 0*/ |          state->depth_comment--;
175: { 4} ( 0) /* 0*/ |          i += 2;
176: { 4} ( 0) /* 0*/ |          continue;
177: { 3} ( 0) /* 0*/ |        }
178: { 3} ( 0) /* 0*/ |        break;
179: { 3} ( 0) /* 0*/ |      case '"':
180: { 3} ( 0) /* 0*/ |        DEBUG_LOG("Entering string literal at index %zu", i);
181: { 3} ( 0) /* 0*/ |        state->in_string = true;
182: { 3} ( 0) /* 0*/ |        state->esc = false;
183: { 3} ( 0) /* 0*/ |        break;
184: { 3} ( 0) /* 0*/ |      case '\'':
185: { 3} ( 0) /* 0*/ |        DEBUG_LOG("Entering char literal at index %zu", i);
186: { 3} ( 0) /* 0*/ |        state->in_char = true;
187: { 3} ( 0) /* 0*/ |        break;
188: { 3} ( 0) /* 0*/ |      case '{':
189: { 3} ( 0) /* 0*/ |        DEBUG_LOG("Entering '{' at index %zu", i);
190: { 3} ( 0) /* 0*/ |        state->depth_curly++;
191: { 3} ( 0) /* 0*/ |        break;
192: { 3} ( 0) /* 0*/ |      case '}':
193: { 3} ( 0) /* 0*/ |        DEBUG_LOG("Exiting '}' at index %zu", i);
194: { 3} ( 0) /* 0*/ |        state->depth_curly--;
195: { 3} ( 0) /* 0*/ |        break;
196: { 3} ( 0) /* 0*/ |      case '(':
197: { 3} ( 0) /* 0*/ |        DEBUG_LOG("Entering '(' at index %zu", i);
198: { 3} ( 0) /* 0*/ |        state->depth_round++;
199: { 3} ( 0) /* 0*/ |        break;
200: { 3} ( 0) /* 0*/ |      case ')':
201: { 3} ( 0) /* 0*/ |        DEBUG_LOG("Found ')' at index %zu", i);
202: { 3} ( 0) /* 0*/ |        state->depth_round--;
203: { 3} ( 0) /* 0*/ |        break;
204: { 3} ( 0) /* 0*/ |      default:
205: { 3} ( 0) /* 0*/ |        break;
206: { 2} ( 0) /* 0*/ |      }
207: { 2} ( 0) /* 0*/ |
208: { 2} ( 0) /* 0*/ |      i += 1;
209: { 2} ( 0) /* 0*/ |
210: { 2} ( 0) /* 0*/ |      // check here so negative depths are recorded even if multiple
     in one line
211: { 3} ( 0) /* 0*/ |      if (state->depth_curly < 0) {
212: { 3} ( 0) /* 0*/ |        state->depth_curly_was_negative = true;
213: { 2} ( 0) /* 0*/ |      }
214: { 3} ( 0) /* 0*/ |      if (state->depth_round < 0) {
215: { 3} ( 0) /* 0*/ |        state->depth_round_was_negative = true;
216: { 2} ( 0) /* 0*/ |      }
217: { 3} ( 0) /* 0*/ |      if (state->depth_comment < 0) {
218: { 3} ( 0) /* 0*/ |        state->depth_comment_was_negative = true;
219: { 2} ( 0) /* 0*/ |      }
220: { 1} ( 0) /* 0*/ |    }
221: { 0} ( 0) /* 0*/ |  }
222: { 0} ( 0) /* 0*/ |
223: { 0} ( 0) /* 0*/ |  // prints the end state of the parser
224: { 1} ( 0) /* 0*/ |  static void print_EndState(State const *const state) {
225: { 2} ( 0) /* 0*/ |    if (state == NULL) {
226: { 2} ( 0) /* 0*/ |      return;
227: { 1} ( 0) /* 0*/ |    }
228: { 1} ( 0) /* 0*/ |    printf("---------------------------------\n");
229: { 2} ( 0) /* 0*/ |    if (state->depth_curly != 0) {
230: { 2} ( 0) /* 0*/ |      printf("    - Geschweifte Klammern { } nicht ausgeglichen\n");
231: { 1} ( 0) /* 0*/ |    }
232: { 2} ( 0) /* 0*/ |    if (state->depth_round != 0) {
233: { 2} ( 0) /* 0*/ |      printf("    - Runde Klammern ( ) nicht ausgeglichen\n");
234: { 1} ( 0) /* 0*/ |    }
235: { 2} ( 0) /* 0*/ |    if (state->depth_comment != 0) {
236: { 2} ( 0) /* 0*/ |      printf("    - Blockkommentare /* */ nicht ausgeglichen\n");
237: { 1} ( 0) /* 0*/ |    }
238: { 2} ( 0) /* 0*/ |    if (state->depth_curly_was_negative) {
239: { 2} ( 0) /* 0*/ |      printf("    - Warning: Geschweifte Klammern waren einmal
     negative\n");
240: { 1} ( 0) /* 0*/ |    }
241: { 2} ( 0) /* 0*/ |    if (state->depth_round_was_negative) {
242: { 2} ( 0) /* 0*/ |      printf("    - Warning: Runde Klammern waren einmal negative\n");
243: { 1} ( 0) /* 0*/ |    }
244: { 2} ( 0) /* 0*/ |    if (state->depth_comment_was_negative) {
245: { 2} ( 0) /* 0*/ |      printf("    - Warning: Blockkommentare waren einmal negative\n")
     ;
246: { 1} ( 0) /* 0*/ |    }
```

```
247:  247: { 1} ( 0) /* 0*/ |
248:  248: { 1} ( 1) /* 0*/ |   if (state->depth_curly == 0 && state->depth_round == 0 &&
249:  249: { 1} ( 1) /* 0*/ |       state->depth_comment == 0 && !state->depth_curly_was_negative
      &&
250:  250: { 2} ( 0) /* 0*/ |       !state->depth_round_was_negative && !state->
      depth_comment_was_negative) {
251:  251: { 2} ( 0) /* 0*/ |     printf("    - Alles ausgeglichen!\n");
252:  252: { 1} ( 0) /* 0*/ |   }
253:  253: { 0} ( 0) /* 0*/ | }
254:  254: { 0} ( 0) /* 0*/ |
255:  255: { 0} ( 0) /* 0*/ | // -------------------- End static Functions ----------------
256:  256: { 0} ( 0) /* 0*/ |
257:  257: { 1} ( 0) /* 0*/ | int main(int argc, char *argv[]) {
258:  258: { 2} ( 0) /* 0*/ |   if (argc != 2) {
259:  259: { 2} ( 0) /* 0*/ |     printf(cErrUsage);
260:  260: { 2} ( 0) /* 0*/ |     DEBUG_LOG("Invalid number of arguments: %d", argc - 1);
261:  261: { 2} ( 0) /* 0*/ |     return EXIT_FAILURE;
262:  262: { 1} ( 0) /* 0*/ |   }
263:  263: { 1} ( 0) /* 0*/ |   DEBUG_LOG("Given File %s", argv[1]);
264:  264: { 1} ( 0) /* 0*/ |
265:  265: { 1} ( 0) /* 0*/ |   FILE *inputfile = fopen(argv[1], "r");
266:  266: { 2} ( 0) /* 0*/ |   if (inputfile == NULL) {
267:  267: { 2} ( 0) /* 0*/ |     perror(cErrOpen);
268:  268: { 2} ( 0) /* 0*/ |     DEBUG_LOG("Could not open file %s", argv[1]);
269:  269: { 2} ( 0) /* 0*/ |     return EXIT_FAILURE;
270:  270: { 1} ( 0) /* 0*/ |   }
271:  271: { 1} ( 0) /* 0*/ |
272:  272: { 1} ( 0) /* 0*/ |   char line[BUFFER_SIZE];
273:  273: { 1} ( 0) /* 0*/ |   State state = {0};
274:  274: { 2} ( 0) /* 0*/ |   while (fgets(line, sizeof(line), inputfile) != NULL) {
275:  275: { 2} ( 0) /* 0*/ |     char *truncated = strchr(line, '\n');
276:  276: { 2} ( 0) /* 0*/ |
277:  277: { 2} ( 0) /* 0*/ |     // when it was not possible to load the complete line into the
      buffer
278:  278: { 2} ( 0) /* 0*/ |     // cancle program since general line length should not be over
      512
279:  279: { 2} ( 0) /* 0*/ |     // In a programm
280:  280: { 2} ( 0) /* 0*/ |     // Correctly handling line feeds would require handling a lot of
       edge cases
281:  281: { 2} ( 0) /* 0*/ |     // like /<bufferend overflow>/ or similar
282:  282: { 3} ( 0) /* 0*/ |     if (truncated == NULL) {
283:  283: { 3} ( 0) /* 0*/ |       perror(cErrLineBufferOverflow);
284:  284: { 3} ( 0) /* 0*/ |       DEBUG_LOG("Line buffer overflow detected");
285:  285: { 3} ( 0) /* 0*/ |       fclose(inputfile); // dont check in error state
286:  286: { 3} ( 0) /* 0*/ |       return EXIT_FAILURE;
287:  287: { 2} ( 0) /* 0*/ |     }
288:  288: { 2} ( 0) /* 0*/ |
289:  289: { 2} ( 0) /* 0*/ |     // update state based on line content
290:  290: { 2} ( 0) /* 0*/ |     process_line(line, &state);
291:  291: { 2} ( 0) /* 0*/ |
292:  292: { 2} ( 0) /* 0*/ |     state.in_line_comment = false; // reset line comment at EOL
293:  293: { 2} ( 0) /* 0*/ |     // Remove newline character for cleaner output
294:  294: { 2} ( 0) /* 0*/ |     *truncated = '\0';
295:  295: { 2} ( 0) /* 0*/ |     state.line_number++;
296:  296: { 2} ( 0) /* 0*/ |
297:  297: { 2} ( 1) /* 0*/ |     printf("%5d: {%2d} (%2d) /*%2d*/ | %s\n", state.line_number,
298:  298: { 2} ( 0) /* 0*/ |            state.depth_curly, state.depth_round, state.depth_comment
      , line);
299:  299: { 1} ( 0) /* 0*/ |   }
300:  300: { 1} ( 0) /* 0*/ |   // check if reading ended because of eof or error
301:  301: { 2} ( 0) /* 0*/ |   if (!feof(inputfile) && ferror(inputfile)) {
302:  302: { 2} ( 0) /* 0*/ |     perror(cErrRead);
303:  303: { 2} ( 0) /* 0*/ |     DEBUG_LOG("Error occurred while reading file %s", argv[1]);
304:  304: { 2} ( 0) /* 0*/ |     fclose(inputfile); // check for fclose error
305:  305: { 1} ( 0) /* 0*/ |   }
306:  306: { 1} ( 0) /* 0*/ |
307:  307: { 1} ( 0) /* 0*/ |   // print final state
308:  308: { 1} ( 0) /* 0*/ |   print_EndState(&state);
309:  309: { 1} ( 0) /* 0*/ |
310:  310: { 1} ( 0) /* 0*/ |   // close file and check for error
311:  311: { 2} ( 0) /* 0*/ |   if (fclose(inputfile) != 0) {
312:  312: { 2} ( 0) /* 0*/ |     perror(cErrClose);
313:  313: { 2} ( 0) /* 0*/ |     DEBUG_LOG("Could not close file %s", argv[1]);
314:  314: { 2} ( 0) /* 0*/ |     return EXIT_FAILURE;
315:  315: { 1} ( 0) /* 0*/ |   }
316:  316: { 1} ( 0) /* 0*/ |
```

```
317   317: { 1} ( 0) /* 0*/ |   return EXIT_SUCCESS;
318   318: { 0} ( 0) /* 0*/ | }
319 ---------------------------------
320     - Alles ausgeglichen!
```