```
#include <ctype.h>     // for tolower, toupper
#include <stdio.h>     // for size_t, fprintf, snprintf, fclose, FILE, NULL
#include <stdlib.h>    // for free, malloc
#include <string.h>    // for strlen, strrchr, strcat, strcmp, strstr, strchr,
memcpy
#include <time.h>      // for localtime, strftime, time, time_t

#ifdef _WIN32
#include <windows.h>   // for FindFirstFileA, FindNextFileA, FindClose,
WIN32_FIND_DATAA, INVALID_HANDLE_VALUE, FILE_ATTRIBUTE_DIRECTORY, DWORD, HANDLE
#else
#include <dirent.h>    // for dirent, DIR, closedir, opendir, readdir
#include <sys/stat.h>  // for stat, S_ISREG, S_ISDIR
#endif

#define MAX_HDR_BUF 65536
#define MAX_PATH_BUF 4096

static int hex_val(char c) {
    if (c >= '0' && c <= '9') return c - '0';
    if (c >= 'A' && c <= 'F') return c - 'A' + 10;
    if (c >= 'a' && c <= 'f') return c - 'a' + 10;
    return -1;
}

static int b64_val(char c) {
    if (c >= 'A' && c <= 'Z') return c - 'A';
    if (c >= 'a' && c <= 'z') return c - 'a' + 26;
    if (c >= '0' && c <= '9') return c - '0' + 52;
    if (c == '+') return 62;
    if (c == '/') return 63;
    return -1;
}

static size_t decode_qp(const char *src, size_t len, unsigned char *dst) {
    size_t out = 0;
    for (size_t i = 0; i < len; i++) {
        if (src[i] == '_') {
            dst[out++] = ' ';
        } else if (src[i] == '=' && i + 2 < len) {
            int h1 = hex_val(src[i + 1]);
            int h2 = hex_val(src[i + 2]);
            if (h1 >= 0 && h2 >= 0) {
                dst[out++] = (unsigned char)((h1 << 4) | h2);
                i += 2;
            } else {
                dst[out++] = (unsigned char)src[i];
            }
        } else {
            dst[out++] = (unsigned char)src[i];
        }
    }
    return out;
}

static size_t decode_b64(const char *src, size_t len, unsigned char *dst) {
    size_t out = 0;
    int val = 0, bits = -8;
    for (size_t i = 0; i < len; i++) {
```

```
        if (src[i] == '=') break;
        int d = b64_val(src[i]);
        if (d < 0) continue;
        val = (val << 6) | d;
        bits += 6;
        if (bits >= 0) {
            dst[out++] = (unsigned char)((val >> bits) & 0xFF);
            bits -= 8;
        }
    }
    return out;
}

static int starts_with_ci(const char *s, const char *prefix) {
    while (*prefix) {
        if (tolower((unsigned char)*s) != tolower((unsigned char)*prefix)) return
0;
        s++;
        prefix++;
    }
    return 1;
}

/* Returns length of matched header name (including ':'), or 0 if not matched.
   When all_headers != 0, matches any valid RFC 5322 header name up to ':'.
   Sets *is_addr_hdr = 1 for From/To/Cc/Reply-To, 0 otherwise. */
static size_t match_target_header(const char *line, int all_headers, int
*is_addr_hdr) {
    if (starts_with_ci(line, "Subject:")) {
        if (is_addr_hdr) *is_addr_hdr = 0;
        return 8;
    }
    if (starts_with_ci(line, "From:")) {
        if (is_addr_hdr) *is_addr_hdr = 1;
        return 5;
    }
    if (starts_with_ci(line, "To:")) {
        if (is_addr_hdr) *is_addr_hdr = 1;
        return 3;
    }
    if (starts_with_ci(line, "Cc:")) {
        if (is_addr_hdr) *is_addr_hdr = 1;
        return 3;
    }
    if (starts_with_ci(line, "Reply-To:")) {
        if (is_addr_hdr) *is_addr_hdr = 1;
        return 9;
    }

    if (all_headers) {
        if (is_addr_hdr) *is_addr_hdr = 0;
        size_t i = 0;
        /* RFC 5322 field-name characters: printable ASCII 33..126 except ':' */
        while (line[i] >= 33 && line[i] <= 126 && line[i] != ':') {
            i++;
        }
        if (i > 0 && line[i] == ':') {
            return i + 1;
        }
```

```
    }
    return 0;
}

static int ends_with_cnm_ci(const char *name) {
    size_t len = strlen(name);
    if (len < 4) return 0;
    const char *ext = name + len - 4;
    return (ext[0] == '.' &&
            tolower((unsigned char)ext[1]) == 'c' &&
            tolower((unsigned char)ext[2]) == 'n' &&
            tolower((unsigned char)ext[3]) == 'm');
}

static void append_str_safe(char *dst, size_t *oi, size_t max_len, const char *str)
{
    while (*str && *oi + 1 < max_len) {
        dst[(*oi)++] = *str++;
    }
}

/* Map UTF-8 0xC3 xx Latin-1 Supplement letters (including multi-char ligatures
like ss) */
static const char *map_latin1_c3(unsigned char b2) {
    if (b2 >= 0x80 && b2 <= 0x85) return "A";
    if (b2 == 0x86) return "AE";
    if (b2 == 0x87) return "C";
    if (b2 >= 0x88 && b2 <= 0x8B) return "E";
    if (b2 >= 0x8C && b2 <= 0x8F) return "I";
    if (b2 == 0x90) return "D";
    if (b2 == 0x91) return "N";
    if ((b2 >= 0x92 && b2 <= 0x96) || b2 == 0x98) return "O";
    if (b2 >= 0x99 && b2 <= 0x9C) return "U";
    if (b2 == 0x9D) return "Y";
    if (b2 == 0x9E) return "TH";
    if (b2 == 0x9F) return "ss";

    if (b2 >= 0xA0 && b2 <= 0xA5) return "a";
    if (b2 == 0xA6) return "ae";
    if (b2 == 0xA7) return "c";
    if (b2 >= 0xA8 && b2 <= 0xAB) return "e";
    if (b2 >= 0xAC && b2 <= 0xAF) return "i";
    if (b2 == 0xB0) return "d";
    if (b2 == 0xB1) return "n";
    if ((b2 >= 0xB2 && b2 <= 0xB6) || b2 == 0xB8) return "o";
    if (b2 >= 0xB9 && b2 <= 0xBC) return "u";
    if (b2 == 0xBD || b2 == 0xBF) return "y";
    if (b2 == 0xBE) return "th";

    return NULL;
}

/* Map common UTF-8 0xC4 / 0xC5 Latin Extended-A characters */
static const char *map_latin_ext_a(unsigned char b1, unsigned char b2) {
    if (b1 == 0xC4) {
        if (b2 == 0x80 || b2 == 0x82 || b2 == 0x84) return "A";
        if (b2 == 0x81 || b2 == 0x83 || b2 == 0x85) return "a";
        if (b2 == 0x86 || b2 == 0x88 || b2 == 0x8A || b2 == 0x8C) return "C";
        if (b2 == 0x87 || b2 == 0x89 || b2 == 0x8B || b2 == 0x8D) return "c";
```

```
        if (b2 == 0x8E || b2 == 0x90) return "D";
        if (b2 == 0x8F || b2 == 0x91) return "d";
        if (b2 == 0x92 || b2 == 0x94 || b2 == 0x96 || b2 == 0x98 || b2 == 0x9A)
return "E";
        if (b2 == 0x93 || b2 == 0x95 || b2 == 0x97 || b2 == 0x99 || b2 == 0x9B)
return "e";
        if (b2 == 0x9C || b2 == 0x9E || b2 == 0xA0 || b2 == 0xA2) return "G";
        if (b2 == 0x9D || b2 == 0x9F || b2 == 0xA1 || b2 == 0xA3) return "g";
        if (b2 == 0xA4 || b2 == 0xA6) return "H";
        if (b2 == 0xA5 || b2 == 0xA7) return "h";
        if (b2 == 0xA8 || b2 == 0xAA || b2 == 0xAC || b2 == 0xAE || b2 == 0xB0)
return "I";
        if (b2 == 0xA9 || b2 == 0xAB || b2 == 0xAD || b2 == 0xAF || b2 == 0xB1)
return "i";
        if (b2 == 0xB4) return "J";
        if (b2 == 0xB5) return "j";
        if (b2 == 0xB6) return "K";
        if (b2 == 0xB7 || b2 == 0xB8) return "k";
        if (b2 == 0xB9 || b2 == 0xBB || b2 == 0xBD || b2 == 0xBF) return "L";
        if (b2 == 0xBA || b2 == 0xBC || b2 == 0xBE) return "l";
    } else if (b1 == 0xC5) {
        if (b2 == 0x80) return "l";
        if (b2 == 0x81) return "L";
        if (b2 == 0x82) return "l";
        if (b2 == 0x83 || b2 == 0x85 || b2 == 0x87 || b2 == 0x8A) return "N";
        if (b2 == 0x84 || b2 == 0x86 || b2 == 0x88 || b2 == 0x89 || b2 == 0x8B)
return "n";
        if (b2 == 0x8C || b2 == 0x8E || b2 == 0x90) return "O";
        if (b2 == 0x8D || b2 == 0x8F || b2 == 0x91) return "o";
        if (b2 == 0x92) return "OE";
        if (b2 == 0x93) return "oe";
        if (b2 == 0x94 || b2 == 0x96 || b2 == 0x98) return "R";
        if (b2 == 0x95 || b2 == 0x97 || b2 == 0x99) return "r";
        if (b2 == 0x9A || b2 == 0x9C || b2 == 0x9E || b2 == 0xA0) return "S";
        if (b2 == 0x9B || b2 == 0x9D || b2 == 0x9F || b2 == 0xA1) return "s";
        if (b2 == 0xA2 || b2 == 0xA4 || b2 == 0xA6) return "T";
        if (b2 == 0xA3 || b2 == 0xA5 || b2 == 0xA7) return "t";
        if (b2 == 0xA8 || b2 == 0xAA || b2 == 0xAC || b2 == 0xAE || b2 == 0xB0 ||
b2 == 0xB2) return "U";
        if (b2 == 0xA9 || b2 == 0xAB || b2 == 0xAD || b2 == 0xAF || b2 == 0xB1 ||
b2 == 0xB3) return "u";
        if (b2 == 0xB4) return "W";
        if (b2 == 0xB5) return "w";
        if (b2 == 0xB6 || b2 == 0xB8) return "Y";
        if (b2 == 0xB7) return "y";
        if (b2 == 0xB9 || b2 == 0xBB || b2 == 0xBD) return "Z";
        if (b2 == 0xBA || b2 == 0xBC || b2 == 0xBE) return "z";
    }
    return NULL;
}

/* Append decoded UTF-8 bytes into clean ASCII out AND raw decoded UTF-8 out.
   Sets *had_stripped = 1 if any character was dropped instead of transliterated.
*/
static void process_utf8_chunk(
    const unsigned char *src, size_t slen,
    char *out_clean, size_t *oi_clean, size_t clean_max,
    char *out_utf8, size_t *oi_utf8, size_t utf8_max,
    int *had_stripped
```

```
) {
    size_t k = 0;
    while (k < slen && *oi_clean + 1 < clean_max) {
        unsigned char b1 = src[k];
        if (b1 < 0x80) {
            if (b1 >= 0x20 && b1 <= 0x7E) {
                out_clean[(*oi_clean)++] = static_cast<char>(b1);
                if (*oi_utf8 + 1 < utf8_max) out_utf8[(*oi_utf8)++] =
static_cast<char>(b1);
            }
            k++;
        } else if ((b1 & 0xE0) == 0xC0 && k + 1 < slen) {
            if (*oi_utf8 + 2 < utf8_max) {
                out_utf8[(*oi_utf8)++] = static_cast<char>(src[k]);
                out_utf8[(*oi_utf8)++] = static_cast<char>(src[k + 1]);
            }

            if (b1 == 0xC3) {
                const char *mapped = map_latin1_c3(src[k + 1]);
                if (mapped) {
                    append_str_safe(out_clean, oi_clean, clean_max, mapped);
                } else {
                    *had_stripped = 1;
                }
            } else if (b1 == 0xC4 || b1 == 0xC5) {
                const char *mapped = map_latin_ext_a(b1, src[k + 1]);
                if (mapped) {
                    append_str_safe(out_clean, oi_clean, clean_max, mapped);
                } else {
                    *had_stripped = 1;
                }
            } else if (b1 == 0xC2) {
                unsigned char b2 = src[k + 1];
                if (b2 == 0xA0) {
                    /* Non-breaking space -> regular space */
                    out_clean[(*oi_clean)++] = ' ';
                } else if (b2 == 0xA9) {
                    append_str_safe(out_clean, oi_clean, clean_max, "(C)");
                } else if (b2 == 0xAE) {
                    append_str_safe(out_clean, oi_clean, clean_max, "(R)");
                } else if (b2 == 0xB0) {
                    append_str_safe(out_clean, oi_clean, clean_max, " deg");
                } else if (b2 == 0xB7) {
                    out_clean[(*oi_clean)++] = '-';
                } else {
                    *had_stripped = 1;
                }
            } else {
                *had_stripped = 1;
            }
            k += 2;
        } else if ((b1 & 0xF0) == 0xE0 && k + 2 < slen) {
            if (*oi_utf8 + 3 < utf8_max) {
                out_utf8[(*oi_utf8)++] = static_cast<char>(src[k]);
                out_utf8[(*oi_utf8)++] = static_cast<char>(src[k + 1]);
                out_utf8[(*oi_utf8)++] = static_cast<char>(src[k + 2]);
            }

            if (b1 == 0xE2 && src[k + 1] == 0x80) {
```

```
                unsigned char b3 = src[k + 2];
                if (b3 == 0x93 || b3 == 0x94) {
                    out_clean[(*oi_clean)++] = '-';
                } else if (b3 == 0x98 || b3 == 0x99) {
                    out_clean[(*oi_clean)++] = '\'';
                } else if (b3 == 0x9C || b3 == 0x9D) {
                    out_clean[(*oi_clean)++] = '"';
                } else if (b3 == 0xA2) {
                    out_clean[(*oi_clean)++] = '*';
                } else if (b3 == 0xA6) {
                    append_str_safe(out_clean, oi_clean, clean_max, "...");
                } else {
                    *had_stripped = 1;
                }
            } else if (b1 == 0xE2 && src[k + 1] == 0x84 && src[k + 2] == 0xA2) {
                append_str_safe(out_clean, oi_clean, clean_max, "(TM)");
            } else {
                *had_stripped = 1;
            }
            k += 3;
        } else if ((b1 & 0xF8) == 0xF0 && k + 3 < slen) {
            if (*oi_utf8 + 4 < utf8_max) {
                out_utf8[(*oi_utf8)++] = static_cast<char>(src[k]);
                out_utf8[(*oi_utf8)++] = static_cast<char>(src[k + 1]);
                out_utf8[(*oi_utf8)++] = static_cast<char>(src[k + 2]);
                out_utf8[(*oi_utf8)++] = static_cast<char>(src[k + 3]);
            }
            *had_stripped = 1;
            k += 4;
        } else {
            k++;
        }
    }
}

/* Collapse consecutive spaces/tabs into single spaces and trim leading/trailing
whitespace */
static void normalize_whitespace(char *s) {
    size_t r = 0, w = 0;
    while (s[r] == ' ' || s[r] == '\t') r++;

    int prev_space = 0;
    while (s[r] != '\0') {
        if (s[r] == ' ' || s[r] == '\t') {
            if (!prev_space) {
                s[w++] = ' ';
                prev_space = 1;
            }
        } else {
            s[w++] = s[r];
            prev_space = 0;
        }
        r++;
    }
    if (w > 0 && s[w - 1] == ' ') w--;
    s[w] = '\0';
}

static int clean_header_value(
```

```
    const char *in,
    char *out_clean, size_t clean_max,
    char *out_utf8, size_t utf8_max
) {
    size_t oi_clean = 0;
    size_t oi_utf8 = 0;
    const char *p = in;
    int prev_was_encoded = 0;
    int had_stripped = 0;

    while (*p && oi_clean + 1 < clean_max) {
        if (p[0] == '=' && p[1] == '?') {
            const char *q1 = strchr(p + 2, '?');
            if (q1 && (q1[1] == 'Q' || q1[1] == 'q' || q1[1] == 'B' || q1[1] ==
'b') && q1[2] == '?') {
                char enc = static_cast<char>(toupper(static_cast<unsigned
char>(q1[1])));
                const char *payload = q1 + 3;
                const char *end = strstr(payload, "?=");
                if (end) {
                    if (prev_was_encoded) {
                        while (oi_clean > 0 && (out_clean[oi_clean - 1] == ' ' ||
out_clean[oi_clean - 1] == '\t')) {
                            oi_clean--;
                        }
                        while (oi_utf8 > 0 && (out_utf8[oi_utf8 - 1] == ' ' ||
out_utf8[oi_utf8 - 1] == '\t')) {
                            oi_utf8--;
                        }
                    }

                    size_t plen = static_cast<size_t>(end - payload);
                    unsigned char *tmp = static_cast<unsigned char *>(malloc(plen +
4));
                    if (tmp) {
                        size_t dlen = (enc == 'Q')
                            ? decode_qp(payload, plen, tmp)
                            : decode_b64(payload, plen, tmp);

                        process_utf8_chunk(
                            tmp, dlen,
                            out_clean, &oi_clean, clean_max,
                            out_utf8, &oi_utf8, utf8_max,
                            &had_stripped
                        );
                        free(tmp);
                    }
                    p = end + 2;
                    prev_was_encoded = 1;
                    continue;
                }
            }
        }

        unsigned char ch = static_cast<unsigned char>(*p);
        if (ch != ' ' && ch != '\t') {
            prev_was_encoded = 0;
        }
```

```
        if (ch < 0x80) {
            if (ch >= 0x20 && ch <= 0x7E) {
                out_clean[oi_clean++] = static_cast<char>(ch);
                if (oi_utf8 + 1 < utf8_max) out_utf8[oi_utf8++] =
static_cast<char>(ch);
            }
            p++;
        } else {
            size_t avail = strlen(p);
            size_t seq_len = 1;
            if ((ch & 0xE0) == 0xC0 && avail >= 2) seq_len = 2;
            else if ((ch & 0xF0) == 0xE0 && avail >= 3) seq_len = 3;
            else if ((ch & 0xF8) == 0xF0 && avail >= 4) seq_len = 4;

            process_utf8_chunk(
                reinterpret_cast<const unsigned char *>(p), seq_len,
                out_clean, &oi_clean, clean_max,
                out_utf8, &oi_utf8, utf8_max,
                &had_stripped
            );
            p += seq_len;
        }
    }

    out_clean[oi_clean] = '\0';
    out_utf8[oi_utf8] = '\0';

    normalize_whitespace(out_clean);
    normalize_whitespace(out_utf8);

    return had_stripped;
}

static void log_stripped_header(
    const char *msg_path,
    const char *custom_log,
    const char *hdr_name,
    const char *raw_val,
    const char *clean_val
) {
    char log_path[MAX_PATH_BUF];
    if (custom_log && custom_log[0] != '\0') {
        snprintf(log_path, sizeof(log_path), "%s", custom_log);
    } else {
        snprintf(log_path, sizeof(log_path), "%s", msg_path);
        char *slash1 = strrchr(log_path, '/');
        char *slash2 = strrchr(log_path, '\\');
        char *last_slash = (slash1 > slash2) ? slash1 : slash2;
        if (last_slash) {
            *(last_slash + 1) = '\0';
            strncat(log_path, "stripped_utf8.log", sizeof(log_path) -
strlen(log_path) - 1);
        } else {
            snprintf(log_path, sizeof(log_path), "stripped_utf8.log");
        }
    }

    FILE *flog = fopen(log_path, "ab");
    if (!flog) return;
```

```
    time_t now = time(NULL);
    const struct tm *tm_info = localtime(&now);
    char ts[32] = "0000-00-00 00:00:00";
    if (tm_info) {
        strftime(ts, sizeof(ts), "%Y-%m-%d %H:%M:%S", tm_info);
    }

    const char *fname = msg_path;
    const char *s1 = strrchr(msg_path, '/');
    const char *s2 = strrchr(msg_path, '\\');
    if (s1 || s2) {
        fname = ((s1 > s2) ? s1 : s2) + 1;
    }

    const char *label = (hdr_name && hdr_name[0] != '\0') ? hdr_name : "Other:";
    fprintf(flog, "[%s] %s (%s)\n  Orig  : %s\n  Clean : %s\n\n", ts, fname, label,
raw_val, clean_val);
    fclose(flog);
}


/* Quick scan of headers to skip files that do not need updating */
static int file_needs_cleaning(const char *path, int all_headers) {
    FILE *f = fopen(path, "rb");
    if (!f) return 0;

    char line[4096];
    int in_target = 0;
    int needs_work = 0;

    while (fgets(line, sizeof(line), f)) {
        size_t len = strlen(line);
        while (len > 0 && (line[len - 1] == '\r' || line[len - 1] == '\n')) {
            line[--len] = '\0';
        }
        if (len == 0) break;

        if (in_target && (line[0] == ' ' || line[0] == '\t')) {
            /* Continuation of active target header */
        } else {
            in_target = (match_target_header(line, all_headers, NULL) > 0) ? 1 : 0;
        }

        if (in_target) {
            if (strstr(line, "=?") != NULL) {
                needs_work = 1;
                break;
            }
            for (size_t i = 0; i < len; i++) {
                if (static_cast<unsigned char>(line[i]) >= 0x80) {
                    needs_work = 1;
                    break;
                }
            }
            if (needs_work) break;
        }
    }
```

```
    fclose(f);
    return needs_work;
}

static void emit_cleaned_header(
    FILE *fout,
    const char *path,
    const char *custom_log,
    const char *hdr_label,
    const char *hdr_raw,
    int detected_crlf
) {
    char hdr_clean[MAX_HDR_BUF];
    char hdr_utf8[MAX_HDR_BUF];

    int had_stripped = clean_header_value(
        hdr_raw,
        hdr_clean, sizeof(hdr_clean),
        hdr_utf8, sizeof(hdr_utf8)
    );

    if (had_stripped || strcmp(hdr_raw, hdr_clean) != 0) {
        log_stripped_header(path, custom_log, hdr_label, hdr_raw, hdr_clean);
    }
    fprintf(fout, "%s %s%s", hdr_label, hdr_clean, detected_crlf ? "\r\n" : "\n");
}

static int process_cnm_file(const char *path, const char *custom_log, int
all_headers) {
    if (!file_needs_cleaning(path, all_headers)) {
        return 0;
    }

    FILE *fin = fopen(path, "rb");
    if (!fin) return -1;

    char tmp_path[MAX_PATH_BUF];
    snprintf(tmp_path, sizeof(tmp_path), "%s.tmp", path);
    FILE *fout = fopen(tmp_path, "wb");
    if (!fout) {
        fclose(fin);
        return -1;
    }

    char line[4096];
    char hdr_label[128];
    char hdr_raw[MAX_HDR_BUF];
    int in_target = 0;
    int headers_done = 0;
    int detected_crlf = 1;

    while (fgets(line, sizeof(line), fin)) {
        if (!headers_done) {
            size_t len = strlen(line);
            int has_crlf = (len >= 2 && line[len - 2] == '\r' && line[len - 1] ==
'\n');
            if (len >= 1) detected_crlf = has_crlf;

            while (len > 0 && (line[len - 1] == '\r' || line[len - 1] == '\n')) {
```

```
                line[--len] = '\0';
            }

            if (in_target) {
                if (len > 0 && (line[0] == ' ' || line[0] == '\t')) {
                    const char *cont = line;
                    while (*cont == ' ' || *cont == '\t') cont++;
                    size_t cur = strlen(hdr_raw);
                    if (cur + strlen(cont) + 2 < sizeof(hdr_raw)) {
                        if (cur > 0) strcat(hdr_raw, " ");
                        strcat(hdr_raw, cont);
                    }
                    continue;
                } else {
                    emit_cleaned_header(fout, path, custom_log, hdr_label, hdr_raw,
detected_crlf);
                    in_target = 0;
                }
            }

            if (len == 0) {
                headers_done = 1;
                fprintf(fout, "%s", detected_crlf ? "\r\n" : "\n");
                continue;
            }

            size_t hlen = match_target_header(line, all_headers, NULL);
            if (hlen > 0) {
                if (hlen < sizeof(hdr_label)) {
                    memcpy(hdr_label, line, hlen);
                    hdr_label[hlen] = '\0';
                } else {
                    snprintf(hdr_label, sizeof(hdr_label), "Other:");
                }

                const char *val = line + hlen;
                while (*val == ' ' || *val == '\t') val++;
                snprintf(hdr_raw, sizeof(hdr_raw), "%s", val);
                in_target = 1;
                continue;
            }

            fprintf(fout, "%s%s", line, has_crlf ? "\r\n" : "\n");
        } else {
            fwrite(line, 1, strlen(line), fout);
        }
    }

    if (in_target) {
        emit_cleaned_header(fout, path, custom_log, hdr_label, hdr_raw,
detected_crlf);
    }

    fclose(fin);
    fclose(fout);

    remove(path);
    if (rename(tmp_path, path) != 0) {
        return -1;
```

```
    }

    return 1;
}

int main(int argc, const char *argv[]) {
    int all_headers = 0;
    const char *arg_path = NULL;
    const char *custom_log = NULL;

    for (int i = 1; i < argc; i++) {
        if (strcmp(argv[i], "-a") == 0 || strcmp(argv[i], "--all-headers") == 0) {
            all_headers = 1;
        } else if (!arg_path) {
            arg_path = argv[i];
        } else if (!custom_log) {
            custom_log = argv[i];
        } else {
            fprintf(stderr, "Usage: %s [-a] <mail_dir_or_file.cnm> [logfile]\n",
argv[0]);
            return 1;
        }
    }

    if (!arg_path) {
        fprintf(stderr, "Usage: %s [-a] <mail_dir_or_file.cnm> [logfile]\n",
argv[0]);
        return 1;
    }

    char target[MAX_PATH_BUF];
    snprintf(target, sizeof(target), "%s", arg_path);
    size_t tlen = strlen(target);
    while (tlen > 1 && (target[tlen - 1] == '/' || target[tlen - 1] == '\\')) {
        target[--tlen] = '\0';
    }

#ifdef _WIN32
    DWORD attr = GetFileAttributesA(target);
    if (attr == INVALID_FILE_ATTRIBUTES) {
        fprintf(stderr, "Error: Path not found: %s\n", target);
        return 1;
    }
    if (attr & FILE_ATTRIBUTE_DIRECTORY) {
        char search_pattern[MAX_PATH_BUF];
        snprintf(search_pattern, sizeof(search_pattern), "%s\\*", target);
        WIN32_FIND_DATAA ffd;
        HANDLE hFind = FindFirstFileA(search_pattern, &ffd);
        if (hFind == INVALID_HANDLE_VALUE) return 1;

        int cleaned_count = 0;
        do {
            if (ffd.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY) continue;
            if (ends_with_cnm_ci(ffd.cFileName)) {
                char full_path[MAX_PATH_BUF];
                snprintf(full_path, sizeof(full_path), "%s\\%s", target,
ffd.cFileName);
                if (process_cnm_file(full_path, custom_log, all_headers) == 1) {
                    cleaned_count++;
```

```
                }
            }
        } while (FindNextFileA(hFind, &ffd) != 0);
        FindClose(hFind);
        printf("Processed directory: %d file(s) updated.\n", cleaned_count);
    } else {
        process_cnm_file(target, custom_log, all_headers);
    }
#else
    struct stat st;
    if (stat(target, &st) != 0) {
        fprintf(stderr, "Error: Path not found: %s\n", target);
        return 1;
    }
    if (S_ISDIR(st.st_mode)) {
        DIR *dir = opendir(target);
        if (!dir) {
            fprintf(stderr, "Error: Cannot open directory: %s\n", target);
            return 1;
        }
        const struct dirent *ent;
        int cleaned_count = 0;
        while ((ent = readdir(dir)) != NULL) {
            if (strcmp(ent->d_name, ".") == 0 || strcmp(ent->d_name, "..") == 0)
continue;
            if (ends_with_cnm_ci(ent->d_name)) {
                char full_path[MAX_PATH_BUF];
                snprintf(full_path, sizeof(full_path), "%s/%s", target, ent-
>d_name);
                struct stat fst;
                if (stat(full_path, &fst) == 0 && S_ISREG(fst.st_mode)) {
                    if (process_cnm_file(full_path, custom_log, all_headers) == 1)
{
                        cleaned_count++;
                    }
                }
            }
        }
        closedir(dir);
        printf("Processed directory: %d file(s) updated.\n", cleaned_count);
    } else if (S_ISREG(st.st_mode)) {
        process_cnm_file(target, custom_log, all_headers);
    }
#endif

    return 0;
}
```