Федеральное агентство связи  
Федеральное государственное бюджетное  
образовательное учреждение высшего образования  
 «Сибирский государственный университет  
телекоммуникаций и информатики»

Кафедра вычислительных систем

Курсовая работа

по дисциплине «Технологии разработки программного обеспечения»

на тему «Конвертер файлов из Markdown в HTML»

Выполнил:

ст. гр. ИВ-122

Исмагилов М.И

Проверил:

ст. преподаватель Токмашева Е. И.

Оглавление

[Введение и постановка задачи 3](#_Toc116627533)

[Техническое задание 4](#_Toc116627534)

[Описание 4](#_Toc116627535)

[Пример итогового файла: 4](#_Toc116627536)

[Интерфейс взаимодействия с пользователем 5](#_Toc116627537)

[Используемые технологии: 5](#_Toc116627538)

[Входные данные 5](#_Toc116627539)

[Описание выполненного проекта 6](#_Toc116627540)

[Приложение. Текст программы 7](#_Toc116627541)

# Введение и постановка задачи

Создать программу, которая реализует форматирование маркдаун текстов в HTML тексты.

Markdown— облегчённый язык разметки, созданный с целью обозначения форматирования в простом тексте, с максимальным сохранением его читаемости человеком, и пригодный для машинного преобразования в языки для продвинутых публикаций.

HTML — стандартизированный язык гипертекстовой разметки документов для просмотра веб-страниц в браузере.

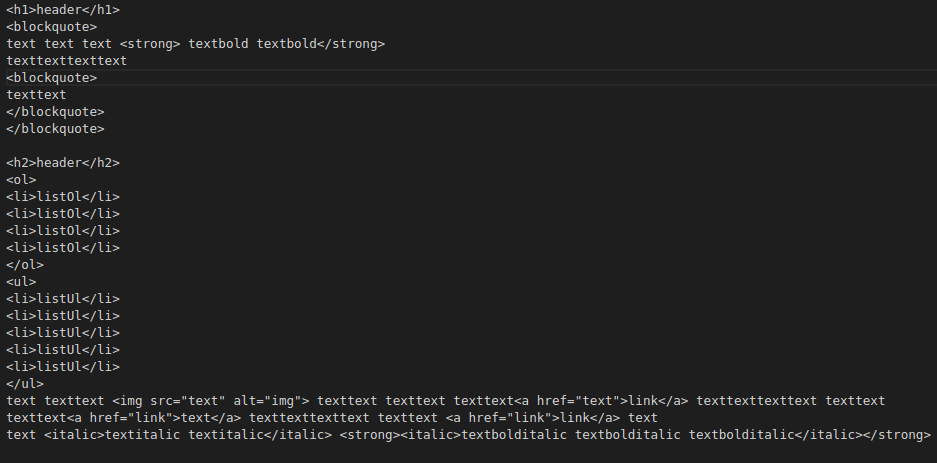
# Техническое задание

## Описание

Отформатировать текст во входном файле из markdown в HTML.

**Пример стартового файла**:

### **Пример итогового** **файла:**



### **Интерфейс взаимодействия с пользователем**

CLI - command line interface (Терминал).

## Используемые технологии:

* Платформа: GNU Linux.
* Язык: C99.
* Компилятор: gcc.
* Система сборки: Make.
* Библиотека для тестирования: ctest.

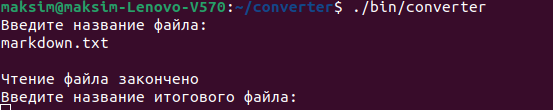
## Входные данные

Текстовый файл с текстом на основе markdown.

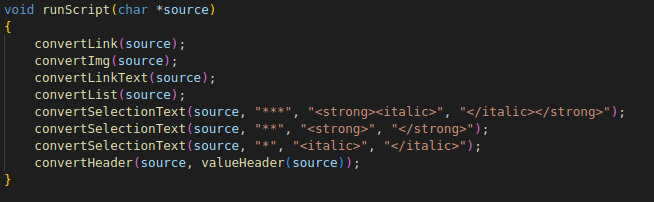
**Аргументы командной строки**

Не предполагается.

# Описание выполненного проекта

1. После запуска программы в терминал нужно написать название входного файла:
2. 
3. После форматирования в терминал нужно записать название выходного файла:
4. 
5. Алгоритм конвертирования:

Скрипт проходит каждую строку в поиске markdown тегов. Строки проверяются сначала на наличие строчных тегов, потом на наличие блочных тегов.



Если в строке находится блочный элемент,закрывающий тег которого может находиться в следующих строках, то скрипт начинает поиск закрывающего тега. Если закрывающий элемент не удаётся найти, или встречается начало другого блочного элемента, то скрипт оставляет этот элемент без изменений. После завершения запускаем скрипт во второй раз, для поиска тега цитат и записываем готовый HTML текст в итоговый файл.

# Приложение. Текст программы

ctest.h

ifndef CTEST\_H

#define CTEST\_H

#ifdef \_\_cplusplus

extern "C" {

#endif

#ifdef \_\_GNUC\_\_

#define CTEST\_IMPL\_FORMAT\_PRINTF(a, b) \_\_attribute\_\_ ((format(printf, a, b)))

#else

#define CTEST\_IMPL\_FORMAT\_PRINTF(a, b)

#endif

#include <inttypes.h> /\* intmax\_t, uintmax\_t, PRI\* \*/

#include <stddef.h> /\* size\_t \*/

typedef void (\*ctest\_nullary\_run\_func)(void);

typedef void (\*ctest\_unary\_run\_func)(void\*);

typedef void (\*ctest\_setup\_func)(void\*);

typedef void (\*ctest\_teardown\_func)(void\*);

union ctest\_run\_func\_union {

    ctest\_nullary\_run\_func nullary;

    ctest\_unary\_run\_func unary;

};

#define CTEST\_IMPL\_PRAGMA(x) \_Pragma (#x)

#if defined(\_\_GNUC\_\_)

#if defined(\_\_clang\_\_) || \_\_GNUC\_\_ > 4 || (\_\_GNUC\_\_ == 4 && \_\_GNUC\_MINOR\_\_ >= 6)

/\* the GCC argument will work for both gcc and clang  \*/

#define CTEST\_IMPL\_DIAG\_PUSH\_IGNORED(w) \

    CTEST\_IMPL\_PRAGMA(GCC diagnostic push) \

    CTEST\_IMPL\_PRAGMA(GCC diagnostic ignored "-W" #w)

#define CTEST\_IMPL\_DIAG\_POP() \

    CTEST\_IMPL\_PRAGMA(GCC diagnostic pop)

#else

/\* the push/pop functionality wasn't in gcc until 4.6, fallback to "ignored"  \*/

#define CTEST\_IMPL\_DIAG\_PUSH\_IGNORED(w) \

    CTEST\_IMPL\_PRAGMA(GCC diagnostic ignored "-W" #w)

#define CTEST\_IMPL\_DIAG\_POP()

#endif

#else

/\* leave them out entirely for non-GNUC compilers  \*/

#define CTEST\_IMPL\_DIAG\_PUSH\_IGNORED(w)

#define CTEST\_IMPL\_DIAG\_POP()

#endif

struct ctest {

    const char\* ssname;  // suite name

    const char\* ttname;  // test name

    union ctest\_run\_func\_union run;

    void\* data;

    ctest\_setup\_func\* setup;

    ctest\_teardown\_func\* teardown;

    int skip;

    unsigned int magic;

};

#define CTEST\_IMPL\_NAME(name) ctest\_##name

#define CTEST\_IMPL\_FNAME(sname, tname) CTEST\_IMPL\_NAME(sname##\_##tname##\_run)

#define CTEST\_IMPL\_TNAME(sname, tname) CTEST\_IMPL\_NAME(sname##\_##tname)

#define CTEST\_IMPL\_DATA\_SNAME(sname) CTEST\_IMPL\_NAME(sname##\_data)

#define CTEST\_IMPL\_DATA\_TNAME(sname, tname) CTEST\_IMPL\_NAME(sname##\_##tname##\_data)

#define CTEST\_IMPL\_SETUP\_FNAME(sname) CTEST\_IMPL\_NAME(sname##\_setup)

#define CTEST\_IMPL\_SETUP\_FPNAME(sname) CTEST\_IMPL\_NAME(sname##\_setup\_ptr)

#define CTEST\_IMPL\_SETUP\_TPNAME(sname, tname) CTEST\_IMPL\_NAME(sname##\_##tname##\_setup\_ptr)

#define CTEST\_IMPL\_TEARDOWN\_FNAME(sname) CTEST\_IMPL\_NAME(sname##\_teardown)

#define CTEST\_IMPL\_TEARDOWN\_FPNAME(sname) CTEST\_IMPL\_NAME(sname##\_teardown\_ptr)

#define CTEST\_IMPL\_TEARDOWN\_TPNAME(sname, tname) CTEST\_IMPL\_NAME(sname##\_##tname##\_teardown\_ptr)

#define CTEST\_IMPL\_MAGIC (0xdeadbeef)

#ifdef \_\_APPLE\_\_

#define CTEST\_IMPL\_SECTION \_\_attribute\_\_ ((used, section ("\_\_DATA, .ctest"), aligned(1)))

#else

#define CTEST\_IMPL\_SECTION \_\_attribute\_\_ ((used, section (".ctest"), aligned(1)))

#endif

#define CTEST\_IMPL\_STRUCT(sname, tname, tskip, tdata, tsetup, tteardown) \

    static struct ctest CTEST\_IMPL\_TNAME(sname, tname) CTEST\_IMPL\_SECTION = { \

        #sname, \

        #tname, \

        { (ctest\_nullary\_run\_func) CTEST\_IMPL\_FNAME(sname, tname) }, \

        tdata, \

        (ctest\_setup\_func\*) tsetup, \

        (ctest\_teardown\_func\*) tteardown, \

        tskip, \

        CTEST\_IMPL\_MAGIC, \

    }

#ifdef \_\_cplusplus

#define CTEST\_SETUP(sname) \

    template <> void CTEST\_IMPL\_SETUP\_FNAME(sname)(struct CTEST\_IMPL\_DATA\_SNAME(sname)\* data)

#define CTEST\_TEARDOWN(sname) \

    template <> void CTEST\_IMPL\_TEARDOWN\_FNAME(sname)(struct CTEST\_IMPL\_DATA\_SNAME(sname)\* data)

#define CTEST\_DATA(sname) \

    template <typename T> void CTEST\_IMPL\_SETUP\_FNAME(sname)(T\* data) { } \

    template <typename T> void CTEST\_IMPL\_TEARDOWN\_FNAME(sname)(T\* data) { } \

    struct CTEST\_IMPL\_DATA\_SNAME(sname)

#define CTEST\_IMPL\_CTEST(sname, tname, tskip) \

    static void CTEST\_IMPL\_FNAME(sname, tname)(void); \

    CTEST\_IMPL\_STRUCT(sname, tname, tskip, NULL, NULL, NULL); \

    static void CTEST\_IMPL\_FNAME(sname, tname)(void)

#define CTEST\_IMPL\_CTEST2(sname, tname, tskip) \

    static struct CTEST\_IMPL\_DATA\_SNAME(sname) CTEST\_IMPL\_DATA\_TNAME(sname, tname); \

    static void CTEST\_IMPL\_FNAME(sname, tname)(struct CTEST\_IMPL\_DATA\_SNAME(sname)\* data); \

    static void (\*CTEST\_IMPL\_SETUP\_TPNAME(sname, tname))(struct CTEST\_IMPL\_DATA\_SNAME(sname)\*) = &CTEST\_IMPL\_SETUP\_FNAME(sname)<struct CTEST\_IMPL\_DATA\_SNAME(sname)>; \

    static void (\*CTEST\_IMPL\_TEARDOWN\_TPNAME(sname, tname))(struct CTEST\_IMPL\_DATA\_SNAME(sname)\*) = &CTEST\_IMPL\_TEARDOWN\_FNAME(sname)<struct CTEST\_IMPL\_DATA\_SNAME(sname)>; \

    CTEST\_IMPL\_STRUCT(sname, tname, tskip, &CTEST\_IMPL\_DATA\_TNAME(sname, tname), &CTEST\_IMPL\_SETUP\_TPNAME(sname, tname), &CTEST\_IMPL\_TEARDOWN\_TPNAME(sname, tname)); \

    static void CTEST\_IMPL\_FNAME(sname, tname)(struct CTEST\_IMPL\_DATA\_SNAME(sname)\* data)

#else

#define CTEST\_SETUP(sname) \

    static void CTEST\_IMPL\_SETUP\_FNAME(sname)(struct CTEST\_IMPL\_DATA\_SNAME(sname)\* data); \

    static void (\*CTEST\_IMPL\_SETUP\_FPNAME(sname))(struct CTEST\_IMPL\_DATA\_SNAME(sname)\*) = &CTEST\_IMPL\_SETUP\_FNAME(sname); \

    static void CTEST\_IMPL\_SETUP\_FNAME(sname)(struct CTEST\_IMPL\_DATA\_SNAME(sname)\* data)

#define CTEST\_TEARDOWN(sname) \

    static void CTEST\_IMPL\_TEARDOWN\_FNAME(sname)(struct CTEST\_IMPL\_DATA\_SNAME(sname)\* data); \

    static void (\*CTEST\_IMPL\_TEARDOWN\_FPNAME(sname))(struct CTEST\_IMPL\_DATA\_SNAME(sname)\*) = &CTEST\_IMPL\_TEARDOWN\_FNAME(sname); \

    static void CTEST\_IMPL\_TEARDOWN\_FNAME(sname)(struct CTEST\_IMPL\_DATA\_SNAME(sname)\* data)

#define CTEST\_DATA(sname) \

    struct CTEST\_IMPL\_DATA\_SNAME(sname); \

    static void (\*CTEST\_IMPL\_SETUP\_FPNAME(sname))(struct CTEST\_IMPL\_DATA\_SNAME(sname)\*); \

    static void (\*CTEST\_IMPL\_TEARDOWN\_FPNAME(sname))(struct CTEST\_IMPL\_DATA\_SNAME(sname)\*); \

    struct CTEST\_IMPL\_DATA\_SNAME(sname)

#define CTEST\_IMPL\_CTEST(sname, tname, tskip) \

    static void CTEST\_IMPL\_FNAME(sname, tname)(void); \

    CTEST\_IMPL\_STRUCT(sname, tname, tskip, NULL, NULL, NULL); \

    static void CTEST\_IMPL\_FNAME(sname, tname)(void)

#define CTEST\_IMPL\_CTEST2(sname, tname, tskip) \

    static struct CTEST\_IMPL\_DATA\_SNAME(sname) CTEST\_IMPL\_DATA\_TNAME(sname, tname); \

    static void CTEST\_IMPL\_FNAME(sname, tname)(struct CTEST\_IMPL\_DATA\_SNAME(sname)\* data); \

    CTEST\_IMPL\_STRUCT(sname, tname, tskip, &CTEST\_IMPL\_DATA\_TNAME(sname, tname), &CTEST\_IMPL\_SETUP\_FPNAME(sname), &CTEST\_IMPL\_TEARDOWN\_FPNAME(sname)); \

    static void CTEST\_IMPL\_FNAME(sname, tname)(struct CTEST\_IMPL\_DATA\_SNAME(sname)\* data)

#endif

void CTEST\_LOG(const char\* fmt, ...) CTEST\_IMPL\_FORMAT\_PRINTF(1, 2);

void CTEST\_ERR(const char\* fmt, ...) CTEST\_IMPL\_FORMAT\_PRINTF(1, 2);  // doesn't return

#define CTEST(sname, tname) CTEST\_IMPL\_CTEST(sname, tname, 0)

#define CTEST\_SKIP(sname, tname) CTEST\_IMPL\_CTEST(sname, tname, 1)

#define CTEST2(sname, tname) CTEST\_IMPL\_CTEST2(sname, tname, 0)

#define CTEST2\_SKIP(sname, tname) CTEST\_IMPL\_CTEST2(sname, tname, 1)

void assert\_str(const char\* exp, const char\* real, const char\* caller, int line);

#define ASSERT\_STR(exp, real) assert\_str(exp, real, \_\_FILE\_\_, \_\_LINE\_\_)

void assert\_wstr(const wchar\_t \*exp, const wchar\_t \*real, const char\* caller, int line);

#define ASSERT\_WSTR(exp, real) assert\_wstr(exp, real, \_\_FILE\_\_, \_\_LINE\_\_)

void assert\_data(const unsigned char\* exp, size\_t expsize,

                 const unsigned char\* real, size\_t realsize,

                 const char\* caller, int line);

#define ASSERT\_DATA(exp, expsize, real, realsize) \

    assert\_data(exp, expsize, real, realsize, \_\_FILE\_\_, \_\_LINE\_\_)

void assert\_equal(intmax\_t exp, intmax\_t real, const char\* caller, int line);

#define ASSERT\_EQUAL(exp, real) assert\_equal(exp, real, \_\_FILE\_\_, \_\_LINE\_\_)

void assert\_equal\_u(uintmax\_t exp, uintmax\_t real, const char\* caller, int line);

#define ASSERT\_EQUAL\_U(exp, real) assert\_equal\_u(exp, real, \_\_FILE\_\_, \_\_LINE\_\_)

void assert\_not\_equal(intmax\_t exp, intmax\_t real, const char\* caller, int line);

#define ASSERT\_NOT\_EQUAL(exp, real) assert\_not\_equal(exp, real, \_\_FILE\_\_, \_\_LINE\_\_)

void assert\_not\_equal\_u(uintmax\_t exp, uintmax\_t real, const char\* caller, int line);

#define ASSERT\_NOT\_EQUAL\_U(exp, real) assert\_not\_equal\_u(exp, real, \_\_FILE\_\_, \_\_LINE\_\_)

void assert\_interval(intmax\_t exp1, intmax\_t exp2, intmax\_t real, const char\* caller, int line);

#define ASSERT\_INTERVAL(exp1, exp2, real) assert\_interval(exp1, exp2, real, \_\_FILE\_\_, \_\_LINE\_\_)

void assert\_null(void\* real, const char\* caller, int line);

#define ASSERT\_NULL(real) assert\_null((void\*)real, \_\_FILE\_\_, \_\_LINE\_\_)

void assert\_not\_null(const void\* real, const char\* caller, int line);

#define ASSERT\_NOT\_NULL(real) assert\_not\_null(real, \_\_FILE\_\_, \_\_LINE\_\_)

void assert\_true(int real, const char\* caller, int line);

#define ASSERT\_TRUE(real) assert\_true(real, \_\_FILE\_\_, \_\_LINE\_\_)

void assert\_false(int real, const char\* caller, int line);

#define ASSERT\_FALSE(real) assert\_false(real, \_\_FILE\_\_, \_\_LINE\_\_)

void assert\_fail(const char\* caller, int line);

#define ASSERT\_FAIL() assert\_fail(\_\_FILE\_\_, \_\_LINE\_\_)

void assert\_dbl\_near(double exp, double real, double tol, const char\* caller, int line);

#define ASSERT\_DBL\_NEAR(exp, real) assert\_dbl\_near(exp, real, 1e-4, \_\_FILE\_\_, \_\_LINE\_\_)

#define ASSERT\_DBL\_NEAR\_TOL(exp, real, tol) assert\_dbl\_near(exp, real, tol, \_\_FILE\_\_, \_\_LINE\_\_)

void assert\_dbl\_far(double exp, double real, double tol, const char\* caller, int line);

#define ASSERT\_DBL\_FAR(exp, real) assert\_dbl\_far(exp, real, 1e-4, \_\_FILE\_\_, \_\_LINE\_\_)

#define ASSERT\_DBL\_FAR\_TOL(exp, real, tol) assert\_dbl\_far(exp, real, tol, \_\_FILE\_\_, \_\_LINE\_\_)

#ifdef CTEST\_MAIN

#include <setjmp.h>

#include <stdarg.h>

#include <stdio.h>

#include <string.h>

#include <sys/time.h>

#include <unistd.h>

#include <stdint.h>

#include <stdlib.h>

#include <wchar.h>

static size\_t ctest\_errorsize;

static char\* ctest\_errormsg;

#define MSG\_SIZE 4096

static char ctest\_errorbuffer[MSG\_SIZE];

static jmp\_buf ctest\_err;

static int color\_output = 1;

static const char\* suite\_name;

typedef int (\*ctest\_filter\_func)(struct ctest\*);

#define ANSI\_BLACK    "\033[0;30m"

#define ANSI\_RED      "\033[0;31m"

#define ANSI\_GREEN    "\033[0;32m"

#define ANSI\_YELLOW   "\033[0;33m"

#define ANSI\_BLUE     "\033[0;34m"

#define ANSI\_MAGENTA  "\033[0;35m"

#define ANSI\_CYAN     "\033[0;36m"

#define ANSI\_GREY     "\033[0;37m"

#define ANSI\_DARKGREY "\033[01;30m"

#define ANSI\_BRED     "\033[01;31m"

#define ANSI\_BGREEN   "\033[01;32m"

#define ANSI\_BYELLOW  "\033[01;33m"

#define ANSI\_BBLUE    "\033[01;34m"

#define ANSI\_BMAGENTA "\033[01;35m"

#define ANSI\_BCYAN    "\033[01;36m"

#define ANSI\_WHITE    "\033[01;37m"

#define ANSI\_NORMAL   "\033[0m"

CTEST(suite, test) { }

static void vprint\_errormsg(const char\* const fmt, va\_list ap) CTEST\_IMPL\_FORMAT\_PRINTF(1, 0);

static void print\_errormsg(const char\* const fmt, ...) CTEST\_IMPL\_FORMAT\_PRINTF(1, 2);

static void vprint\_errormsg(const char\* const fmt, va\_list ap) {

    // (v)snprintf returns the number that would have been written

    const int ret = vsnprintf(ctest\_errormsg, ctest\_errorsize, fmt, ap);

    if (ret < 0) {

        ctest\_errormsg[0] = 0x00;

    } else {

        const size\_t size = (size\_t) ret;

        const size\_t s = (ctest\_errorsize <= size ? size -ctest\_errorsize : size);

        // ctest\_errorsize may overflow at this point

        ctest\_errorsize -= s;

        ctest\_errormsg += s;

    }

}

static void print\_errormsg(const char\* const fmt, ...) {

    va\_list argp;

    va\_start(argp, fmt);

    vprint\_errormsg(fmt, argp);

    va\_end(argp);

}

static void msg\_start(const char\* color, const char\* title) {

    if (color\_output) {

        print\_errormsg("%s", color);

    }

    print\_errormsg("  %s: ", title);

}

static void msg\_end(void) {

    if (color\_output) {

        print\_errormsg(ANSI\_NORMAL);

    }

    print\_errormsg("\n");

}

void CTEST\_LOG(const char\* fmt, ...)

{

    va\_list argp;

    msg\_start(ANSI\_BLUE, "LOG");

    va\_start(argp, fmt);

    vprint\_errormsg(fmt, argp);

    va\_end(argp);

    msg\_end();

}

CTEST\_IMPL\_DIAG\_PUSH\_IGNORED(missing-noreturn)

void CTEST\_ERR(const char\* fmt, ...)

{

    va\_list argp;

    msg\_start(ANSI\_YELLOW, "ERR");

    va\_start(argp, fmt);

    vprint\_errormsg(fmt, argp);

    va\_end(argp);

    msg\_end();

    longjmp(ctest\_err, 1);

}

CTEST\_IMPL\_DIAG\_POP()

void assert\_str(const char\* exp, const char\*  real, const char\* caller, int line) {

    if ((exp == NULL && real != NULL) ||

        (exp != NULL && real == NULL) ||

        (exp && real && strcmp(exp, real) != 0)) {

        CTEST\_ERR("%s:%d  expected '%s', got '%s'", caller, line, exp, real);

    }

}

void assert\_wstr(const wchar\_t \*exp, const wchar\_t \*real, const char\* caller, int line) {

    if ((exp == NULL && real != NULL) ||

        (exp != NULL && real == NULL) ||

        (exp && real && wcscmp(exp, real) != 0)) {

        CTEST\_ERR("%s:%d  expected '%ls', got '%ls'", caller, line, exp, real);

    }

}

void assert\_data(const unsigned char\* exp, size\_t expsize,

                 const unsigned char\* real, size\_t realsize,

                 const char\* caller, int line) {

    size\_t i;

    if (expsize != realsize) {

        CTEST\_ERR("%s:%d  expected %" PRIuMAX " bytes, got %" PRIuMAX, caller, line, (uintmax\_t) expsize, (uintmax\_t) realsize);

    }

    for (i=0; i<expsize; i++) {

        if (exp[i] != real[i]) {

            CTEST\_ERR("%s:%d expected 0x%02x at offset %" PRIuMAX " got 0x%02x",

                caller, line, exp[i], (uintmax\_t) i, real[i]);

        }

    }

}

void assert\_equal(intmax\_t exp, intmax\_t real, const char\* caller, int line) {

    if (exp != real) {

        CTEST\_ERR("%s:%d  expected %" PRIdMAX ", got %" PRIdMAX, caller, line, exp, real);

    }

}

void assert\_equal\_u(uintmax\_t exp, uintmax\_t real, const char\* caller, int line) {

    if (exp != real) {

        CTEST\_ERR("%s:%d  expected %" PRIuMAX ", got %" PRIuMAX, caller, line, exp, real);

    }

}

void assert\_not\_equal(intmax\_t exp, intmax\_t real, const char\* caller, int line) {

    if ((exp) == (real)) {

        CTEST\_ERR("%s:%d  should not be %" PRIdMAX, caller, line, real);

    }

}

void assert\_not\_equal\_u(uintmax\_t exp, uintmax\_t real, const char\* caller, int line) {

    if ((exp) == (real)) {

        CTEST\_ERR("%s:%d  should not be %" PRIuMAX, caller, line, real);

    }

}

void assert\_interval(intmax\_t exp1, intmax\_t exp2, intmax\_t real, const char\* caller, int line) {

    if (real < exp1 || real > exp2) {

        CTEST\_ERR("%s:%d  expected %" PRIdMAX "-%" PRIdMAX ", got %" PRIdMAX, caller, line, exp1, exp2, real);

    }

}

void assert\_dbl\_near(double exp, double real, double tol, const char\* caller, int line) {

    double diff = exp - real;

    double absdiff = diff;

    /\* avoid using fabs and linking with a math lib \*/

    if(diff < 0) {

      absdiff \*= -1;

    }

    if (absdiff > tol) {

        CTEST\_ERR("%s:%d  expected %0.3e, got %0.3e (diff %0.3e, tol %0.3e)", caller, line, exp, real, diff, tol);

    }

}

void assert\_dbl\_far(double exp, double real, double tol, const char\* caller, int line) {

    double diff = exp - real;

    double absdiff = diff;

    /\* avoid using fabs and linking with a math lib \*/

    if(diff < 0) {

      absdiff \*= -1;

    }

    if (absdiff <= tol) {

        CTEST\_ERR("%s:%d  expected %0.3e, got %0.3e (diff %0.3e, tol %0.3e)", caller, line, exp, real, diff, tol);

    }

}

void assert\_null(void\* real, const char\* caller, int line) {

    if ((real) != NULL) {

        CTEST\_ERR("%s:%d  should be NULL", caller, line);

    }

}

void assert\_not\_null(const void\* real, const char\* caller, int line) {

    if (real == NULL) {

        CTEST\_ERR("%s:%d  should not be NULL", caller, line);

    }

}

void assert\_true(int real, const char\* caller, int line) {

    if ((real) == 0) {

        CTEST\_ERR("%s:%d  should be true", caller, line);

    }

}

void assert\_false(int real, const char\* caller, int line) {

    if ((real) != 0) {

        CTEST\_ERR("%s:%d  should be false", caller, line);

    }

}

void assert\_fail(const char\* caller, int line) {

    CTEST\_ERR("%s:%d  shouldn't come here", caller, line);

}

static int suite\_all(struct ctest\* t) {

    (void) t; // fix unused parameter warning

    return 1;

}

static int suite\_filter(struct ctest\* t) {

    return strncmp(suite\_name, t->ssname, strlen(suite\_name)) == 0;

}

static uint64\_t getCurrentTime(void) {

    struct timeval now;

    gettimeofday(&now, NULL);

    uint64\_t now64 = (uint64\_t) now.tv\_sec;

    now64 \*= 1000000;

    now64 += ((uint64\_t) now.tv\_usec);

    return now64;

}

static void color\_print(const char\* color, const char\* text) {

    if (color\_output)

        printf("%s%s" ANSI\_NORMAL "\n", color, text);

    else

        printf("%s\n", text);

}

#ifdef CTEST\_SEGFAULT

#include <signal.h>

static void sighandler(int signum)

{

    const char msg\_color[] = ANSI\_BRED "[SIGSEGV: Segmentation fault]" ANSI\_NORMAL "\n";

    const char msg\_nocolor[] = "[SIGSEGV: Segmentation fault]\n";

    const char\* msg = color\_output ? msg\_color : msg\_nocolor;

    write(STDOUT\_FILENO, msg, strlen(msg));

    /\* "Unregister" the signal handler and send the signal back to the process

     \* so it can terminate as expected \*/

    signal(signum, SIG\_DFL);

    kill(getpid(), signum);

}

#endif

int ctest\_main(int argc, const char \*argv[]);

\_\_attribute\_\_((no\_sanitize\_address)) int ctest\_main(int argc, const char \*argv[])

{

    static int total = 0;

    static int num\_ok = 0;

    static int num\_fail = 0;

    static int num\_skip = 0;

    static int idx = 1;

    static ctest\_filter\_func filter = suite\_all;

#ifdef CTEST\_SEGFAULT

    signal(SIGSEGV, sighandler);

#endif

    if (argc == 2) {

        suite\_name = argv[1];

        filter = suite\_filter;

    }

#ifdef CTEST\_NO\_COLORS

    color\_output = 0;

#else

    color\_output = isatty(1);

#endif

    uint64\_t t1 = getCurrentTime();

    struct ctest\* ctest\_begin = &CTEST\_IMPL\_TNAME(suite, test);

    struct ctest\* ctest\_end = &CTEST\_IMPL\_TNAME(suite, test);

    // find begin and end of section by comparing magics

    while (1) {

        struct ctest\* t = ctest\_begin-1;

        if (t->magic != CTEST\_IMPL\_MAGIC) break;

        ctest\_begin--;

    }

    while (1) {

        struct ctest\* t = ctest\_end+1;

        if (t->magic != CTEST\_IMPL\_MAGIC) break;

        ctest\_end++;

    }

    ctest\_end++;    // end after last one

    static struct ctest\* test;

    for (test = ctest\_begin; test != ctest\_end; test++) {

        if (test == &CTEST\_IMPL\_TNAME(suite, test)) continue;

        if (filter(test)) total++;

    }

    for (test = ctest\_begin; test != ctest\_end; test++) {

        if (test == &CTEST\_IMPL\_TNAME(suite, test)) continue;

        if (filter(test)) {

            ctest\_errorbuffer[0] = 0;

            ctest\_errorsize = MSG\_SIZE-1;

            ctest\_errormsg = ctest\_errorbuffer;

            printf("TEST %d/%d %s:%s ", idx, total, test->ssname, test->ttname);

            fflush(stdout);

            if (test->skip) {

                color\_print(ANSI\_BYELLOW, "[SKIPPED]");

                num\_skip++;

            } else {

                int result = setjmp(ctest\_err);

                if (result == 0) {

                    if (test->setup && \*test->setup) (\*test->setup)(test->data);

                    if (test->data)

                        test->run.unary(test->data);

                    else

                        test->run.nullary();

                    if (test->teardown && \*test->teardown) (\*test->teardown)(test->data);

                    // if we got here it's ok

#ifdef CTEST\_COLOR\_OK

                    color\_print(ANSI\_BGREEN, "[OK]");

#else

                    printf("[OK]\n");

#endif

                    num\_ok++;

                } else {

                    color\_print(ANSI\_BRED, "[FAIL]");

                    num\_fail++;

                }

                if (ctest\_errorsize != MSG\_SIZE-1) printf("%s", ctest\_errorbuffer);

            }

            idx++;

        }

    }

    uint64\_t t2 = getCurrentTime();

    const char\* color = (num\_fail) ? ANSI\_BRED : ANSI\_GREEN;

    char results[80];

    snprintf(results, sizeof(results), "RESULTS: %d tests (%d ok, %d failed, %d skipped) ran in %" PRIu64 " ms", total, num\_ok, num\_fail, num\_skip, (t2 - t1)/1000);

    color\_print(color, results);

    return num\_fail;

}

#endif

#ifdef \_\_cplusplus

}

#endif

#endif

func-test.c

#include <ctest.h>

#include <libconverter/converter.h>

#include <libconverter/strings.h>

#include <stdio.h>

#include <stdlib.h>

char FileArray[1024][1024] = {};

int PosScript = 0;

int CountStr = 0;

CTEST(concat\_suite, str1\_null)

{

    char \*s1 = NULL;

    char \*s2 = "test";

    ASSERT\_STR(NULL, concat(s1, s2));

}

CTEST(concat\_suite, str2\_null)

{

    char \*s2 = NULL;

    char \*s1 = "test";

    ASSERT\_STR(NULL, concat(s1, s2));

}

CTEST(concat\_suite, concat\_right)

{

    char \*s2 = "test";

    char \*s1 = "test";

    ASSERT\_STR("testtest", concat(s1, s2));

}

CTEST(concat3\_suite, str1\_null)

{

    char \*s1 = NULL;

    char \*s2 = "test";

    char \*s3 = "test";

    ASSERT\_STR(NULL, concat3(s1, s2, s3));

}

CTEST(concat3\_suite, str2\_null)

{

    char \*s2 = NULL;

    char \*s1 = "test";

    char \*s3 = "test";

    ASSERT\_STR(NULL, concat3(s1, s2, s3));

}

CTEST(concat3\_suite, str3\_null)

{

    char \*s3 = NULL;

    char \*s2 = "test";

    char \*s1 = "test";

    ASSERT\_STR(NULL, concat3(s1, s2, s3));

}

CTEST(concat3\_suite, concat3\_right)

{

    char \*s1 = "test";

    char \*s2 = "test";

    char \*s3 = "test";

    ASSERT\_STR("testtesttest", concat3(s1, s2, s3));

}

CTEST(pos\_suite, str1\_null)

{

    char \*s1 = NULL;

    char \*s2 = "test";

    ASSERT\_EQUAL(-2, Pos(s1, s2));

}

CTEST(pos\_suite, str2\_null)

{

    char \*s1 = "test";

    char \*s2 = NULL;

    ASSERT\_EQUAL(-2, Pos(s1, s2));

}

CTEST(pos\_suite, dont\_find)

{

    char \*s1 = "test";

    char \*s2 = "ddd";

    ASSERT\_EQUAL(-1, Pos(s1, s2));

}

CTEST(pos\_suite, find)

{

    char \*s1 = "testtest";

    char \*s2 = "test";

    ASSERT\_EQUAL(0, Pos(s1, s2));

}

CTEST(getStartString\_suite, right)

{

    char \*s1 = "privettest";

    char \*s2 = malloc(5 \* sizeof(char));

    getStartString(s1, s2, 5);

    ASSERT\_STR("privet", s2);

}

CTEST(getEndString\_suite, right)

{

    char \*s1 = "privettest";

    char \*s2 = malloc(4 \* sizeof(char));

    getEndString(s1, s2, 5);

    ASSERT\_STR("test", s2);

}

CTEST(getSubString\_suite, right)

{

    char \*s1 = "privetpokattest";

    char \*s2 = malloc(4 \* sizeof(char));

    getSubString(s1, s2, 5, 9);

    ASSERT\_STR("poka", s2);

}

CTEST(ValueCitation\_suite, dont\_find)

{

    char \*s1 = "test";

    ASSERT\_EQUAL(0, ValueCitation(s1));

}

CTEST(ValueCitation\_suite, value1)

{

    char \*s1 = "> test";

    ASSERT\_EQUAL(1, ValueCitation(s1));

}

CTEST(ValueCitation\_suite, value2)

{

    char \*s1 = ">> test";

    ASSERT\_EQUAL(2, ValueCitation(s1));

}

CTEST(ValueCitation\_suite, value3)

{

    char \*s1 = ">>> test";

    ASSERT\_EQUAL(3, ValueCitation(s1));

}

CTEST(ValueCitation\_suite, value4)

{

    char \*s1 = ">>>> test";

    ASSERT\_EQUAL(4, ValueCitation(s1));

}

CTEST(ValueCitation\_suite, value5)

{

    char \*s1 = ">>>>> test";

    ASSERT\_EQUAL(5, ValueCitation(s1));

}

CTEST(valueList\_suite, dont\_find)

{

    char \*s1 = "test";

    ASSERT\_EQUAL(0, valueList(s1));

}

CTEST(valueList\_suite, find\_ol)

{

    char \*s1 = "1. test";

    ASSERT\_EQUAL(2, valueList(s1));

}

CTEST(valueList\_suite, find\_ul\_plus)

{

    char \*s1 = "+ test";

    ASSERT\_EQUAL(1, valueList(s1));

}

CTEST(valueList\_suite, find\_ul\_minus)

{

    char \*s1 = "- test";

    ASSERT\_EQUAL(1, valueList(s1));

}

CTEST(valueList\_suite, find\_ul\_star)

{

    char \*s1 = "\* test";

    ASSERT\_EQUAL(1, valueList(s1));

}

CTEST(valueHeader\_suite, dont\_find)

{

    char \*s1 = "test";

    ASSERT\_EQUAL(0, valueHeader(s1));

}

CTEST(valueHeader\_suite, value1)

{

    char \*s1 = "# test";

    ASSERT\_EQUAL(1, valueHeader(s1));

}

CTEST(valueHeader\_suite, value2)

{

    char \*s1 = "## test";

    ASSERT\_EQUAL(2, valueHeader(s1));

}

CTEST(valueHeader\_suite, value3)

{

    char \*s1 = "### test";

    ASSERT\_EQUAL(3, valueHeader(s1));

}

CTEST(valueHeader\_suite, value4)

{

    char \*s1 = "#### test";

    ASSERT\_EQUAL(4, valueHeader(s1));

}

CTEST(valueHeader\_suite, value5)

{

    char \*s1 = "##### test";

    ASSERT\_EQUAL(5, valueHeader(s1));

}

CTEST(valueHeader\_suite, value6)

{

    char \*s1 = "###### test";

    ASSERT\_EQUAL(6, valueHeader(s1));

}

CTEST(convertStringElement\_suite, LinkText)

{

    char \*s1 = "test[text](link)test";

    char \*s2 = "<a href=\"link\">text</a>";

    char \*s3 = malloc(28 \* sizeof(char));

    s3 = convertStringElement(s1, s2, 4, 14);

    ASSERT\_STR("test<a href=\"link\">text</a>test", s3);

}

CTEST(convertStringElement\_suite, Link)

{

    char \*s1 = "test<link>test";

    char \*s2 = "<a href=\"link\">link</a>";

    char \*s3 = malloc(24 \* sizeof(char));

    s3 = convertStringElement(s1, s2, 4, 8);

    ASSERT\_STR("test<a href=\"link\">link</a>test", s3);

}

CTEST(convertStringElement\_suite, img)

{

    char \*s1 = "test![text](link)test";

    char \*s2 = "<img=\"link\"alt=\"text\">";

    char \*s3 = malloc(28 \* sizeof(char));

    s3 = convertStringElement(s1, s2, 4, 15);

    ASSERT\_STR("test<img=\"link\"alt=\"text\">test", s3);

}

CTEST(findHtml\_suite, dont\_find)

{

    char \*s1 = "testtest";

    ASSERT\_EQUAL(0, findHtml(s1));

}

CTEST(findHtml\_suite, find\_open\_link)

{

    char \*s1 = "test<a>test";

    ASSERT\_EQUAL(1, findHtml(s1));

}

CTEST(findHtml\_suite, find\_close\_link)

{

    char \*s1 = "test</a>test";

    ASSERT\_EQUAL(1, findHtml(s1));

}

CTEST(findHtml\_suite, find\_open\_list)

{

    char \*s1 = "test<li>test";

    ASSERT\_EQUAL(1, findHtml(s1));

}

CTEST(findHtml\_suite, find\_close\_list)

{

    char \*s1 = "test</li>test";

    ASSERT\_EQUAL(1, findHtml(s1));

}

CTEST(findHtml\_suite, find\_open\_strong)

{

    char \*s1 = "test<strong>test";

    ASSERT\_EQUAL(1, findHtml(s1));

}

CTEST(findHtml\_suite, find\_close\_strong)

{

    char \*s1 = "test</strong>test";

    ASSERT\_EQUAL(1, findHtml(s1));

}

CTEST(findHtml\_suite, find\_open\_italic)

{

    char \*s1 = "test<italic>test";

    ASSERT\_EQUAL(1, findHtml(s1));

}

CTEST(findHtml\_suite, find\_close\_italic)

{

    char \*s1 = "test</italic>test";

    ASSERT\_EQUAL(1, findHtml(s1));

}

CTEST(findHtml\_suite, find\_open\_citation)

{

    char \*s1 = "<blockquote>\ntesttest";

    ASSERT\_EQUAL(1, findHtml(s1));

}

CTEST(findHtml\_suite, find\_close\_citation)

{

    char \*s1 = "</blockquote>\n test";

    ASSERT\_EQUAL(1, findHtml(s1));

}

CTEST(findHtml\_suite, find\_open\_list\_ol)

{

    char \*s1 = "test<ol>test";

    ASSERT\_EQUAL(1, findHtml(s1));

}

CTEST(findHtml\_suite, find\_close\_list\_ol)

{

    char \*s1 = "test</ol>test";

    ASSERT\_EQUAL(1, findHtml(s1));

}

CTEST(findHtml\_suite, find\_open\_list\_ul)

{

    char \*s1 = "test<ul>test";

    ASSERT\_EQUAL(1, findHtml(s1));

}

CTEST(findHtml\_suite, find\_close\_list\_ul)

{

    char \*s1 = "test</ul>test";

    ASSERT\_EQUAL(1, findHtml(s1));

}

CTEST(findMask\_suite, dont\_find)

{

    char \*s1 = "test";

    ASSERT\_STR(NULL, findMask(s1));

}

CTEST(findMask\_suite, find\_open\_boldItalic)

{

    char \*s1 = "test<strong></italic>test";

    ASSERT\_STR("<strong></italic>", findMask(s1));

}

CTEST(findMask\_suite, find\_close\_boldItalic)

{

    char \*s1 = "test</italic></strong>test";

    ASSERT\_STR("</italic></strong>", findMask(s1));

}

CTEST(findMask\_suite, find\_open\_bold)

{

    char \*s1 = "test<strong>test";

    ASSERT\_STR("<strong>", findMask(s1));

}

CTEST(findMask\_suite, find\_close\_bold)

{

    char \*s1 = "test</strong>test";

    ASSERT\_STR("</strong>", findMask(s1));

}

CTEST(findMask\_suite, find\_open\_italic)

{

    char \*s1 = "test<italic>test";

    ASSERT\_STR("<italic>", findMask(s1));

}

CTEST(findMask\_suite, find\_close\_italic)

{

    char \*s1 = "test</italic>test";

    ASSERT\_STR("</italic>", findMask(s1));

}

test.c

#define CTEST\_MAIN

#include <ctest.h>

int main(int argc, const char\*\* argv)

{

    return ctest\_main(argc, argv);

}

strings.h

char \*concat(char \*s1, char \*s2);

char \*concat3(char \*s1, char \*s2, char \*s3);

int Pos(const char \*const str1, const char \*const str2);

void getStartString(const char str[], char destination[], int pos);

void getEndString(const char str[], char destination[], int pos);

void getSubString(const char str[], char destination[], int startPos, int endPos);

strings.c

#include "strings.h"

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

char \*concat(char \*s1, char \*s2)

{

    if ((s1 == NULL) || (s2 == NULL))

    {

        return NULL;

    }

    size\_t len1 = strlen(s1);

    size\_t len2 = strlen(s2);

    char \*result = malloc(len1 + len2 + 1);

    if (!result)

    {

        fprintf(stderr, "malloc() failed: insufficient memory!\n");

        return NULL;

    }

    memcpy(result, s1, len1);

    memcpy(result + len1, s2, len2 + 1);

    return result;

}

char \*concat3(char \*s1, char \*s2, char \*s3)

{

    if ((s1 == NULL) || (s2 == NULL) || (s3 == NULL))

    {

        return NULL;

    }

    size\_t len1 = strlen(s1);

    size\_t len2 = strlen(s2);

    size\_t len3 = strlen(s3);

    char \*result = malloc(len1 + len2 + len3 + 1);

    if (!result)

    {

        fprintf(stderr, "malloc() failed: insufficient memory!\n");

        return NULL;

    }

    memcpy(result, s1, len1);

    memcpy(result + len1, s2, len2);

    memcpy(result + len1 + len2, s3, len3 + 1);

    return result;

}

int Pos(const char \*const str1, const char \*const str2)

{

    int res = -1;

    int flag = 0;

    int i = 0;

    int j = 0;

    if ((str1 == NULL) || (str2 == NULL))

        res = -2;

    else

    {

        int len1 = (strlen(str1));

        int len2 = (strlen(str2));

        while ((res < 0) && (i <= len1 - len2))

        {

            flag = 1;

            j = 0;

            while ((flag) && (j < len2))

            {

                if (str1[i + j] != str2[j])

                    flag = 0;

                j++;

            }

            if (flag)

                res = i;

            i++;

        }

    }

    return res;

}

void getStartString(const char str[], char destination[], int pos)

{

    int i = 0;

    while (i <= pos)

    {

        destination[i] = str[i];

        i++;

    }

}

void getEndString(const char str[], char destination[], int pos)

{

    int i = 0;

    int skip = 0;

    while (str[i] != '\0')

    {

        if (i > pos)

        {

            destination[i - skip] = str[i];

        }

        else

        {

            skip++;

        }

        i++;

    }

}

void getSubString(const char str[], char destination[], int startPos, int endPos)

{

    int i = 0;

    int skip = 0;

    while (str[i] != '\0')

    {

        if (i > endPos)

        {

            break;

        }

        if (i > startPos)

        {

            destination[i - skip] = str[i];

        }

        else

        {

            skip++;

        }

        i++;

    }

}

converter.h

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

int ValueCitation(char \*source);

void convertCitation(char \*source);

int valueList(char \*source);

void convertList(char \*source);

int valueHeader(char \*source);

void convertHeader(char \*source, int value);

char \*convertStringElement(char \*source, char \*convertedElement, int posStart, int posEnd);

void convertLinkText(char \*source);

void replaceOnTag(char \*source, char \*destination, char \*mask, char \*tag);

int findHtml(char \*source);

char \*findMask(char \*source);

void convertLink(char \*source);

void convertSelectionText(char \*source, char \*mask, char \*openTag, char \*closeTag);

void convertImg(char \*source);

void runScript(char \*source);

void runScriptCitation(char \*source);

void convert();

converter.c

#include "converter.h"

#include "strings.h"

char fileArray[1024][1024] = {};

int posScript = 0;

int countStr = 0;

int ValueCitation(char \*source)

{

    int res = 0;

    char \*arrayCitations[5];

    arrayCitations[0] = "> ";

    arrayCitations[1] = ">> ";

    arrayCitations[2] = ">>> ";

    arrayCitations[3] = ">>>> ";

    arrayCitations[4] = ">>>>> ";

    for (int i = 0; i < 5; i++)

    {

        if ((strstr(source, arrayCitations[i]) != NULL) && (Pos(source, arrayCitations[i]) == 0))

        {

            res = i + 1;

        }

    }

    return res;

}

void convertCitation(char \*source)

{

    int value = ValueCitation(source);

    if (value > 0)

    {

        char \*text = malloc(strlen(source) - value);

        getEndString(source, text, value);

        char \*convertedElement = concat("<blockquote>\n", text);

        for (int i = 1; i < value; i++)

        {

            convertedElement = concat("<blockquote>\n", convertedElement);

        }

        strcpy(fileArray[posScript], convertedElement);

        for (int i = posScript + 1; i < countStr; i++)

        {

            int count = ValueCitation(fileArray[i]);

            if (count != 0)

            {

                char \*str = malloc(strlen(fileArray[i]) - count);

                getEndString(fileArray[i], str, count);

                if (count > value)

                {

                    for (int j = 0; j < count - value; j++)

                    {

                        str = concat("<blockquote>\n", str);

                    }

                    value = count;

                }

                strcpy(fileArray[i], str);

            }

            else if (strlen(fileArray[i]) == 1)

            {

                for (int j = 0; j < value; j++)

                {

                    strcpy(fileArray[i - 1], concat(fileArray[i - 1], "</blockquote>\n"));

                }

                break;

            }

        }

    }

}

int valueList(char \*source)

{

    int value = 0;

    if (strlen(source) > 2)

    {

        if ((source[0] >= '1') && (source[0] <= '9') && (Pos(source, ". ") < 2))

        {

            value = 2;

        }

        if (((Pos(source, "+ ") < 1) && (strstr(source, "+ ") != NULL)) || ((Pos(source, "- ") < 1) && (strstr(source, "- ") != NULL)) || ((Pos(source, "\* ") < 1) && (strstr(source, "\* ") != NULL)))

        {

            value = 1;

        }

    }

    return value;

}

void convertList(char \*source)

{

    int value = valueList(source);

    if (value > 0)

    {

        char \*text = malloc(strlen(source) - value);

        getSubString(source, text, value, Pos(source, "\n") - 1);

        char \*convertedElement = concat3("<li>", text, "</li>\n");

        if (value == 1)

        {

            convertedElement = concat("<ul>\n", convertedElement);

        }

        if (value == 2)

        {

            convertedElement = concat("<ol>\n", convertedElement);

        }

        strcpy(fileArray[posScript], convertedElement);

        for (int i = posScript + 1; i < countStr; i++)

        {

            if ((valueList(fileArray[i]) != value) || (strlen(fileArray[i]) == 1))

            {

                if (value == 1)

                {

                    strcpy(fileArray[i - 1], concat(fileArray[i - 1], "</ul>\n"));

                }

                if (value == 2)

                {

                    strcpy(fileArray[i - 1], concat(fileArray[i - 1], "</ol>\n"));

                }

                break;

            }

            else if (valueList(fileArray[i]) == value)

            {

                char \*str = malloc(strlen(fileArray[i]) - value + 11);

                getSubString(fileArray[i], str, value, Pos(fileArray[i], "\n") - 1);

                str = concat3("<li>", str, "</li>\n");

                strcpy(fileArray[i], str);

            }

        }

    }

}

int valueHeader(char \*source)

{

    int value = 0;

    char \*arrayHeaders[6];

    arrayHeaders[0] = "# ";

    arrayHeaders[1] = "## ";

    arrayHeaders[2] = "### ";

    arrayHeaders[3] = "#### ";

    arrayHeaders[4] = "##### ";

    arrayHeaders[5] = "###### ";

    for (int i = 0; i < 6; i++)

    {

        if ((strstr(source, arrayHeaders[i]) != NULL) && (Pos(source, arrayHeaders[i]) == 0))

        {

            value = i + 1;

        }

    }

    return value;

}

void convertHeader(char \*source, int value)

{

    if (value > 0)

    {

        char \*arrayHtmlHeadersStart[6];

        arrayHtmlHeadersStart[0] = "<h1>";

        arrayHtmlHeadersStart[1] = "<h2>";

        arrayHtmlHeadersStart[2] = "<h3>";

        arrayHtmlHeadersStart[3] = "<h4>";

        arrayHtmlHeadersStart[4] = "<h5>";

        arrayHtmlHeadersStart[5] = "<h6>";

        char \*arrayHtmlHeadersEnd[6];

        arrayHtmlHeadersEnd[0] = "</h1>\n";

        arrayHtmlHeadersEnd[1] = "</h2>\n";

        arrayHtmlHeadersEnd[2] = "</h3>\n";

        arrayHtmlHeadersEnd[3] = "</h4>\n";

        arrayHtmlHeadersEnd[4] = "</h5>\n";

        arrayHtmlHeadersEnd[5] = "</h6>\n";

        size\_t len = strlen(source);

        char \*text = malloc(len - value);

        getSubString(source, text, value, Pos(source, "\n") - 1);

        char \*result = concat3(arrayHtmlHeadersStart[value - 1], text, arrayHtmlHeadersEnd[value - 1]);

        strcpy(fileArray[posScript], result);

    }

}

char \*convertStringElement(char \*source, char \*convertedElement, int posStart, int posEnd)

{

    char \*startStr = malloc(posStart + 1); // Если что тут не было +1

    char \*finishStr = malloc(strlen(source) - posEnd - 1);

    getStartString(source, startStr, posStart - 1);

    getEndString(source, finishStr, posEnd + 1);

    return concat3(startStr, convertedElement, finishStr);

    // strcpy(destination, result);

}

void convertLinkText(char \*source)

{

    if (strstr(source, "](") != NULL)

    {

        char arrayHtmlLinkStart1[9] = "<a href=\"";

        char arrayHtmlLinkStart2[3] = "\">";

        char arrayHtmlLinkEnd[5] = "</a>";

        char \*link = malloc(Pos(source, ")") - Pos(source, "](") - 2);

        char \*text = malloc(Pos(source, "]") - Pos(source, "[") - 1);

        getSubString(source, link, Pos(source, "](") + 1, Pos(source, ")") - 1);

        getSubString(source, text, Pos(source, "["), Pos(source, "]") - 1);

        char \*convertedElement = concat3(arrayHtmlLinkStart1, link, arrayHtmlLinkStart2);

        convertedElement = concat3(convertedElement, text, arrayHtmlLinkEnd);

        char \*result = convertStringElement(source, convertedElement, Pos(source, "["), Pos(source, ")") - 1);

        strcpy(fileArray[posScript], result);

        if (strstr(source, "](") != NULL)

        {

            convertLinkText(source);

        }

    }

}

void replaceOnTag(char \*source, char \*destination, char \*mask, char \*tag)

{

    int posMask = Pos(source, mask);

    char \*leftStr = malloc(Pos(source, mask) + strlen(mask) - 1);

    char \*rightStr = malloc(strlen(source) - Pos(source, mask));

    getStartString(source, leftStr, posMask - 1);

    getEndString(source, rightStr, posMask + strlen(mask) - 1);

    char \*result = concat3(leftStr, tag, rightStr);

    strcpy(destination, result);

}

int findHtml(char \*source)

{

    int res = 0;

    char \*htmlTags[14];

    htmlTags[0] = "<a>";

    htmlTags[1] = "</a>";

    htmlTags[2] = "<li>";

    htmlTags[3] = "</li>";

    htmlTags[4] = "<strong>";

    htmlTags[5] = "</strong>";

    htmlTags[6] = "<italic>";

    htmlTags[7] = "</italic>";

    htmlTags[8] = "<blockquote>";

    htmlTags[9] = "</blockquote>";

    htmlTags[10] = "<ol>";

    htmlTags[11] = "</ol>";

    htmlTags[12] = "<ul>";

    htmlTags[13] = "</ul>";

    for (int i = 0; i < 14; i++)

    {

        if (strstr(source, htmlTags[i]) != NULL)

        {

            res = 1;

        }

    }

    return res;

}

char \*findMask(char \*source)

{

    char \*masks[7];

    masks[6] = "<strong>";

    masks[5] = "</strong>";

    masks[4] = "<italic>";

    masks[3] = "</italic>";

    masks[1] = "<strong></italic>";

    masks[2] = "</italic></strong>";

    for (int i = 0; i < 7; i++)

    {

        if (strstr(source, masks[i]) != NULL)

        {

            return masks[i];

        }

    }

    return NULL;

}

void convertLink(char \*source)

{

    if ((ValueCitation(source) == 0) && (strstr(source, "<") != NULL) && (strstr(source, ">") != NULL))

    {

        char htmlLinkStart1[9] = "<a href=\"";

        char htmlLinkStart2[3] = "\">";

        char htmlLinkEnd[5] = "</a>";

        if (findHtml(source) > 0)

        {

            if (strstr(source, "<li>") != NULL)

            {

                char \*str = malloc(strlen(source));

                getSubString(source, str, 3, Pos(source, "</li>") - 1);

                if ((strstr(str, "<") != NULL) && (strstr(str, ">") != NULL))

                {

                    char \*link = malloc(Pos(str, ">") - Pos(str, "<") - 1);

                    getSubString(str, link, Pos(str, "<"), Pos(str, ">") - 1);

                    char \*convertedElement = concat3(htmlLinkStart1, link, htmlLinkStart2);

                    convertedElement = concat3(convertedElement, link, htmlLinkEnd);

                    char \*result = convertStringElement(source, convertedElement, Pos(str, "<") + 4, Pos(str, ">") + 3);

                    strcpy(fileArray[posScript], result);

                }

            }

            if (findMask(source) != NULL)

            {

                char \*mask = malloc(strlen(findMask(source)));

                mask = findMask(source);

                char \*left = malloc(Pos(source, mask));

                char \*right = malloc(strlen(source) - Pos(source, mask) + 1);

                getStartString(source, left, Pos(source, mask) - 1);

                getEndString(source, right, Pos(source, mask) + strlen(mask));

                if ((strstr(right, "<") != NULL) && (strstr(right, ">") != NULL))

                {

                    char \*link = malloc(Pos(right, ">") - Pos(right, "<") - 1);

                    getSubString(right, link, Pos(right, "<"), Pos(right, ">") - 1);

                    char \*convertedElement = concat3(htmlLinkStart1, link, htmlLinkStart2);

                    convertedElement = concat3(convertedElement, link, htmlLinkEnd);

                    char \*result = convertStringElement(source, convertedElement, Pos(right, "<") + strlen(left) + strlen(mask) + 1, Pos(right, ">") + strlen(left) + strlen(mask));

                    strcpy(fileArray[posScript], result);

                }

                if ((strstr(left, "<") != NULL) && (strstr(left, ">") != NULL))

                {

                    char \*link = malloc(Pos(left, ">") - Pos(left, "<") - 1);

                    getSubString(left, link, Pos(left, "<"), Pos(left, ">") - 1);

                    char \*convertedElement = concat3(htmlLinkStart1, link, htmlLinkStart2);

                    convertedElement = concat3(convertedElement, link, htmlLinkEnd);

                    char \*result = convertStringElement(source, convertedElement, Pos(left, "<"), Pos(left, ">") - 1);

                    strcpy(fileArray[posScript], result);

                }

            }

        }

        if (findHtml(source) == 0)

        {

            if ((strstr(source, "<") != NULL) && (strstr(source, ">") != NULL))

            {

                char \*link = malloc(Pos(source, ">") - Pos(source, "<") - 1);

                getSubString(source, link, Pos(source, "<"), Pos(source, ">") - 1);

                char \*convertedElement = concat3(htmlLinkStart1, link, htmlLinkStart2);

                convertedElement = concat3(convertedElement, link, htmlLinkEnd);

                char \*result = convertStringElement(source, convertedElement, Pos(source, "<"), Pos(source, ">") - 1);

                strcpy(fileArray[posScript], result);

            }

        }

    }

}

void convertSelectionText(char \*source, char \*mask, char \*openTag, char \*closeTag)

{

    if (strstr(source, mask) != NULL)

    {

        char \*text1 = malloc(strlen(source) - Pos(source, mask));

        getSubString(source, text1, Pos(source, mask) + strlen(mask) - 1, strlen(source));

        if (strstr(text1, mask) != NULL)

        {

            char \*text = malloc(Pos(text1, mask));

            getSubString(text1, text, -1, Pos(text1, mask) - 1);

            if (strlen(text) > 0)

            {

                int lengthTags = strlen(openTag) + strlen(closeTag);

                char \*convertedElement = malloc(strlen(text) + lengthTags);

                convertedElement = concat3(openTag, text, closeTag);

                char \*result = convertStringElement(source, convertedElement, Pos(source, mask), Pos(source, text) + strlen(text) + strlen(mask) - 2);

                if (Pos(result, mask) != -1)

                {

                    convertSelectionText(result, mask, openTag, closeTag);

                }

                else

                {

                    strcpy(fileArray[posScript], result);

                }

            }

        }

        else

        {

            for (int i = posScript + 1; i < countStr - 1; i++)

            {

                if ((ValueCitation(fileArray[i]) > 0) || (strstr(fileArray[i], "<li>") != NULL) || (valueHeader(fileArray[i]) > 0) || (valueList(fileArray[i]) > 0) || (strlen(fileArray[i]) == 1))

                {

                    break;

                }

                char \*str = fileArray[i];

                if (Pos(str, mask) != -1)

                {

                    replaceOnTag(source, fileArray[posScript], mask, openTag);

                    replaceOnTag(str, fileArray[i], mask, closeTag);

                    break;

                }

            }

        }

    }

}

void convertImg(char \*source)

{

    if (strstr(source, "!["))

    {

        char arrayHtmlImgStart1[10] = "<img src=\"";

        char arrayHtmlImgStart2[3] = "\" ";

        char arrayHtmlImgEnd2[3] = "\">";

        char \*left = malloc(Pos(source, "!["));

        char \*right = malloc(strlen(source) - Pos(source, "![") + 1);

        getStartString(source, left, Pos(source, "![") + 1);

        getEndString(source, right, Pos(source, "![") + 1);

        char \*path = malloc(Pos(right, ")") - Pos(right, "](") - 2);

        char \*text = malloc(Pos(right, "]"));

        getSubString(right, path, Pos(right, "](") + 1, Pos(right, ")") - 1);

        getSubString(right, text, -1, Pos(right, "]") - 1);

        char \*convertedElement = concat3(arrayHtmlImgStart1, path, arrayHtmlImgStart2);

        convertedElement = concat3(convertedElement, "alt=\"", text);

        convertedElement = concat(convertedElement, arrayHtmlImgEnd2);

        char \*result = convertStringElement(source, convertedElement, Pos(source, "!["), Pos(right, ")") + strlen(left) - 1);

        strcpy(fileArray[posScript], result);

        if (strstr(source, "![") != NULL)

        {

            convertImg(source);

        }

    }

}

void runScript(char \*source)

{

    convertLink(source);

    convertImg(source);

    convertLinkText(source);

    convertList(source);

    convertSelectionText(source, "\*\*\*", "<strong><italic>", "</italic></strong>");

    convertSelectionText(source, "\*\*", "<strong>", "</strong>");

    convertSelectionText(source, "\*", "<italic>", "</italic>");

    convertHeader(source, valueHeader(source));

}

void runScriptCitation(char \*source)

{

    convertCitation(source);

}

void convert()

{

    char fileNameStart[30];

    FILE \*file;

    char \*estr;

    printf("Введите название файла: \n");

    scanf("%s", fileNameStart);

    file = fopen(fileNameStart, "r");

    int i = 0;

    while (1)

    {

        char \*arr = fileArray[i];

        estr = fgets(arr, 1024, file);

        if (estr == NULL)

        {

            if (feof(file) != 0)

            {

                printf("\nЧтение файла закончено\n");

                break;

            }

            else

            {

                printf("\nОшибка чтения из файла\n");

                break;

            }

        }

        i++;

    }

    countStr = i;

    while (posScript != i)

    {

        runScript(fileArray[posScript]);

        posScript++;

    }

    posScript = 0;

    FILE \*ffinish;

    char fileNameConvert[30];

    printf("Введите название итогового файла: \n");

    scanf("%s", fileNameConvert);

    ffinish = fopen(fileNameConvert, "w");

    while (posScript != i)

    {

        runScriptCitation(fileArray[posScript]);

        fprintf(ffinish, "%s", fileArray[posScript]);

        posScript++;

    }

    fclose(file);

    fclose(ffinish);

}

main.c

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <libconverter/converter.h>

int main()

{

  convert();

  return 0;

}

Makefile

CC = gcc

APP\_NAME = converter

LIB\_NAME = libconverter

TEST\_NAME = converter-test

CFLAGS = -Wall -Wextra -Werror

CPPFLAGS = -I src -MP -MMD

CPPFLAGST = -I ctests -MP -MMD

LDFLAGS =

LDLIBS =

BIN\_DIR = bin

OBJ\_DIR = obj

SRC\_DIR = src

TEST\_DIR = test

APP\_PATH = $(BIN\_DIR)/$(APP\_NAME)

TEST\_PATH = $(BIN\_DIR)/$(TEST\_NAME)

LIB\_PATH = $(OBJ\_DIR)/$(SRC\_DIR)/$(LIB\_NAME)/$(LIB\_NAME).a

SRC\_EXT = c

APP\_SOURCES = $(shell find $(SRC\_DIR)/$(APP\_NAME) -name '\*.$(SRC\_EXT)')

APP\_OBJECTS = $(APP\_SOURCES:$(SRC\_DIR)/%.$(SRC\_EXT)=$(OBJ\_DIR)/$(SRC\_DIR)/%.o)

TEST\_SOURCES = $(shell find $(TEST\_DIR) -name '\*.$(SRC\_EXT)')

TEST\_OBJECTS = $(TEST\_SOURCES:$(TEST\_DIR)/%.$(SRC\_EXT)=$(OBJ\_DIR)/$(TEST\_DIR)/%.o)

LIB\_SOURCES = $(shell find $(SRC\_DIR)/$(LIB\_NAME) -name '\*.$(SRC\_EXT)')

LIB\_OBJECTS = $(LIB\_SOURCES:$(SRC\_DIR)/%.$(SRC\_EXT)=$(OBJ\_DIR)/$(SRC\_DIR)/%.o)

DEPS = $(APP\_OBJECTS:.o=.d) $(LIB\_OBJECTS:.o=.d) $(TEST\_OBJECTS:.o=.d)

.PHONY: all

all: $(APP\_PATH)

-include $(DEPS)

$(APP\_PATH): $(APP\_OBJECTS) $(LIB\_PATH)

    $(CC) $(CFLAGS) $(CPPFLAGS) $^ -o $@ $(LDFLAGS) $(LDLIBS)

$(LIB\_PATH): $(LIB\_OBJECTS)

    ar rcs $@ $^

$(OBJ\_DIR)/%.o: %.c

    $(CC) -c $(CFLAGS) $(CPPFLAGS) $(CPPFLAGST) $< -o $@

.PHONY: clean

clean:

    $(RM) $(APP\_PATH) $(TEST\_PATH) $(LIB\_PATH)

    find $(OBJ\_DIR) -name '\*.o' -exec $(RM) '{}' \;

    find $(OBJ\_DIR) -name '\*.d' -exec $(RM) '{}' \;

.PHONY: test

test: $(TEST\_PATH)

-include $(DEPS)

$(TEST\_PATH): $(TEST\_OBJECTS) $(LIB\_PATH)

    $(CC) $(CFLAGS) $(CPPFLAGST) $^ -o $@ $(LDFLAGS) $(LDLIBS)