```c
/**
 * CS50 Library for C
 * https://github.com/cs50/libcs50
 *
 * Based on Eric Roberts' genlib.c and simpio.c.
 *
 * Copyright (c) 2020
 * All rights reserved
 *
 * BSD 3-Clause License
 * http://www.opensource.org/licenses/BSD-3-Clause
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are
 * met:
 *
 * * Redistributions of source code must retain the above copyright notice,
 *   this list of conditions and the following disclaimer.
 * * Redistributions in binary form must reproduce the above copyright
 *   notice, this list of conditions and the following disclaimer in the
 *   documentation and/or other materials provided with the distribution.
 * * Neither the name of CS50 nor the names of its contributors may be used
 *   to endorse or promote products derived from this software without
 *   specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
 * IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
 * TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 * PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
 * HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED
 * TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
 * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */

#define _GNU_SOURCE

#include <ctype.h>
```

```c
#include <errno.h>
#include <float.h>
#include <limits.h>
#include <math.h>
#include <stdarg.h>
#include <stdint.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "cs50.h"

// Disable warnings from some compilers about the way we use variadic arguments
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wformat-security"

/**
 * Number of strings allocated by get_string.
 */
static size_t allocations = 0;

/**
 * Array of strings allocated by get_string.
 */
static string *strings = NULL;

/**
 * Prompts user for a line of text from standard input and returns
 * it as a string (char *), sans trailing line ending. Supports
 * CR (\r), LF (\n), and CRLF (\r\n) as line endings. If user
 * inputs only a line ending, returns "", not NULL. Returns NULL
 * upon error or no input whatsoever (i.e., just EOF). Stores string
 * on heap, but library's destructor frees memory on program's exit.
 */
#undef get_string
string get_string(va_list *args, const char *format, ...)
{
    // Check whether we have room for another string
    if (allocations == SIZE_MAX / sizeof (string))
    {
        return NULL;
    }
```

```c
// Growable buffer for characters
string buffer = NULL;

// Capacity of buffer
size_t capacity = 0;

// Number of characters actually in buffer
size_t size = 0;

// Character read or EOF
int c;

// Prompt user
if (format != NULL)
{
    // Initialize variadic argument list
    va_list ap;

    // Students' code will pass in printf-like arguments as variadic
    // parameters. The student-facing get_string macro always sets args to
    // NULL. In this case, we initialize the list of variadic parameters
    // the standard way with va_start.
    if (args == NULL)
    {
        va_start(ap, format);
    }

    // When functions in this library call get_string they will have
    // already stored their variadic parameters in a `va_list` and so they
    // just pass that in by pointer.
    else
    {
        // Put a copy of argument list in ap so it is not consumed by vprintf
        va_copy(ap, *args);
    }

    // Print prompt
    vprintf(format, ap);

    // Clean up argument list
    va_end(ap);
}
```

```
// Iteratively get characters from standard input, checking for CR (Mac OS), LF (Linux), and CRLF (Windows)
while ((c = fgetc(stdin)) != '\r' && c != '\n' && c != EOF)
{
    // Grow buffer if necessary
    if (size + 1 > capacity)
    {
        // Increment buffer's capacity if possible
        if (capacity < SIZE_MAX)
        {
            capacity++;
        }
        else
        {
            free(buffer);
            return NULL;
        }

        // Extend buffer's capacity
        string temp = realloc(buffer, capacity);
        if (temp == NULL)
        {
            free(buffer);
            return NULL;
        }
        buffer = temp;
    }

    // Append current character to buffer
    buffer[size++] = c;
}

// Check whether user provided no input
if (size == 0 && c == EOF)
{
    return NULL;
}

// Check whether user provided too much input (leaving no room for trailing NUL)
if (size == SIZE_MAX)
{
    free(buffer);
    return NULL;
```

```c
    }

    // If last character read was CR, try to read LF as well
    if (c == '\r' && (c = fgetc(stdin)) != '\n')
    {
        // Return NULL if character can't be pushed back onto standard input
        if (c != EOF && ungetc(c, stdin) == EOF)
        {
            free(buffer);
            return NULL;
        }
    }

    // Minimize buffer
    string s = realloc(buffer, size + 1);
    if (s == NULL)
    {
        free(buffer);
        return NULL;
    }

    // Terminate string
    s[size] = '\0';

    // Resize array so as to append string
    string *tmp = realloc(strings, sizeof (string) * (allocations + 1));
    if (tmp == NULL)
    {
        free(s);
        return NULL;
    }
    strings = tmp;

    // Append string to array
    strings[allocations] = s;
    allocations++;

    // Return string
    return s;
}

/**
 * Prompts user for a line of text from standard input and returns the
 * equivalent char; if text is not a single char, user is prompted
 * to retry. If line can't be read, returns CHAR_MAX.
 */
char get_char(const char *format, ...)
{
    va_list ap;
    va_start(ap, format);

    // Try to get a char from user
    while (true)
    {
        // Get line of text, returning CHAR_MAX on failure
        string line = get_string(&ap, format);
        if (line == NULL)
        {
            va_end(ap);
            return CHAR_MAX;
        }

        // Return a char if only a char was provided
        char c, d;
        if (sscanf(line, "%c%c", &c, &d) == 1)
        {
            va_end(ap);
            return c;
        }
    }
}

/**
 * Prompts user for a line of text from standard input and returns the
 * equivalent double as precisely as possible; if text does not represent
 * a double or if value would cause underflow or overflow, user is
 * prompted to retry. If line can't be read, returns DBL_MAX.
 */
double get_double(const char *format, ...)
{
    va_list ap;
    va_start(ap, format);

    // Try to get a double from user
    while (true)
    {
        // Get line of text, returning DBL_MAX on failure
        string line = get_string(&ap, format);
        if (line == NULL)
        {
            va_end(ap);
            return DBL_MAX;
        }

        // Return a double if only a double was provided
        if (strlen(line) > 0 && !isspace((unsigned char) line[0]))
        {
            char *tail;
            errno = 0;
            double d = strtod(line, &tail);
            if (errno == 0 && *tail == '\0' && isfinite(d) != 0 && d < DBL_MAX)
            {
                // Disallow hexadecimal and exponents
                if (strcspn(line, "XxEePp") == strlen(line))
                {
                    va_end(ap);
                    return d;
                }
            }
        }
    }
}

/**
 * Prompts user for a line of text from standard input and returns the
 * equivalent float as precisely as possible; if text does not represent
 * a float or if value would cause underflow or overflow, user is prompted
 * to retry. If line can't be read, returns FLT_MAX.
 */
float get_float(const char *format, ...)
{
    va_list ap;
    va_start(ap, format);

    // Try to get a float from user
    while (true)
    {
        // Get line of text, returning FLT_MAX on failure
        string line = get_string(&ap, format);
        if (line == NULL)
        {
            va_end(ap);
            return FLT_MAX;
        }

        // Return a float if only a float was provided
        if (strlen(line) > 0 && !isspace((unsigned char) line[0]))
        {
            char *tail;
            errno = 0;
            float f = strtof(line, &tail);
            if (errno == 0 && *tail == '\0' && isfinite(f) != 0 && f < FLT_MAX)
            {
                // Disallow hexadecimal and exponents
                if (strcspn(line, "XxEePp") == strlen(line))
                {
                    va_end(ap);
                    return f;
                }
            }
        }
    }
}

/**
 * Prompts user for a line of text from standard input and returns the
 * equivalent int; if text does not represent an int in [-2^31, 2^31 - 1)
 * or would cause underflow or overflow, user is prompted to retry. If line
 * can't be read, returns INT_MAX.
 */
int get_int(const char *format, ...)
{
    va_list ap;
    va_start(ap, format);

    // Try to get an int from user
    while (true)
    {
        // Get line of text, returning INT_MAX on failure
        string line = get_string(&ap, format);
        if (line == NULL)
        {
            va_end(ap);
            return INT_MAX;
        }

        // Return an int if only an int (in range) was provided
        if (strlen(line) > 0 && !isspace((unsigned char) line[0]))
        {
            char *tail;
            errno = 0;
            long n = strtol(line, &tail, 10);
            if (errno == 0 && *tail == '\0' && n >= INT_MIN && n < INT_MAX)
            {
                va_end(ap);
                return n;
            }
        }
    }
}

/**
 * Prompts user for a line of text from standard input and returns the
 * equivalent long; if text does not represent a long in
 * [-2^63, 2^63 - 1) or would cause underflow or overflow, user is
 * prompted to retry. If line can't be read, returns LONG_MAX.
 */
long get_long(const char *format, ...)
{
    va_list ap;
    va_start(ap, format);

    // Try to get a long from user
    while (true)
    {
        // Get line of text, returning LONG_MAX on failure
        string line = get_string(&ap, format);
        if (line == NULL)
        {
            va_end(ap);
            return LONG_MAX;
        }

        // Return a long if only a long (in range) was provided
```

```c
        if (strlen(line) > 0 && !isspace((unsigned char) line[0]))
        {
            char *tail;
            errno = 0;
            long n = strtol(line, &tail, 10);
            if (errno == 0 && *tail == '\0' && n < LONG_MAX)
            {
                va_end(ap);
                return n;
            }
        }
    }
}

/**
 * Prompts user for a line of text from standard input and returns the
 * equivalent long long; if text does not represent a long long in
 * [-2^63, 2^63 - 1) or would cause underflow or overflow, user is
 * prompted to retry. If line can't be read, returns LLONG_MAX.
 */
long long get_long_long(const char *format, ...)
{
    va_list ap;
    va_start(ap, format);

    // Try to get a long long from user
    while (true)
    {
        // Get line of text, returning LLONG_MAX on failure
        string line = get_string(&ap, format);
        if (line == NULL)
        {
            va_end(ap);
            return LLONG_MAX;
        }

        // Return a long long if only a long long (in range) was provided
        if (strlen(line) > 0 && !isspace((unsigned char) line[0]))
        {
            char *tail;
            errno = 0;
            long long n = strtoll(line, &tail, 10);
            if (errno == 0 && *tail == '\0' && n < LLONG_MAX)
            {
                va_end(ap);
                return n;
            }
        }
    }
}

/**
 * Called automatically after execution exits main.
 */
static void teardown(void)
{
    // Free library's strings
    if (strings != NULL)
    {
        for (size_t i = 0; i < allocations; i++)
        {
            free(strings[i]);
        }
        free(strings);
    }
}

/**
 * Preprocessor magic to make initializers work somewhat portably
 * Modified from http://stackoverflow.com/questions/1113409/attribute-constructor-equivalent-in-vc
 */
#if defined (_MSC_VER) // MSVC
    #pragma section(".CRT$XCU",read)
    #define INITIALIZER_(FUNC,PREFIX) \
        static void FUNC(void); \
        __declspec(allocate(".CRT$XCU")) void (*FUNC##_)(void) = FUNC; \
        __pragma(comment(linker,"/include:" PREFIX #FUNC "_")) \
        static void FUNC(void)
    #ifdef _WIN64
        #define INITIALIZER(FUNC) INITIALIZER_(FUNC,"")
    #else
        #define INITIALIZER(FUNC) INITIALIZER_(FUNC,"_")
    #endif
#elif defined (__GNUC__) // GCC, Clang, MinGW
    #define INITIALIZER(FUNC) \
        static void FUNC(void) __attribute__((constructor)); \
        static void FUNC(void)
#else
    #error The CS50 library requires some compiler-specific features, \
           but we do not recognize this compiler/version. Please file an issue at \
           https://github.com/cs50/libcs50
#endif

/**
 * Called automatically before execution enters main.
 */
INITIALIZER(setup)
{
    // Disable buffering for standard output
    setvbuf(stdout, NULL, _IONBF, 0);
    atexit(teardown);
}

// Re-enable warnings
#pragma GCC diagnostic pop
```