Parity.c

/\* Parity.c - Returns PAR\_EVEN if parity is Even

Returns PAR\_ODD if parity is Odd

Return Par\_ERROR if error in input

Copyright 2019 Stephane Durette

\*/

#include "Parity.h" // Header file includes Prototypes defined in Parity.cpp

PARITY ParityOnBytes(char\* buf, int iBufLen) {

int iSum = 0;

for (int i = 0; i < iBufLen; i++, buf++) {

if (\*buf == '1') {

iSum++;

}

else if (\*buf != '0') {

return(PAR\_ERROR);

}

}

if (iSum % 2 == 0) {

return PAR\_EVEN;

}

return PAR\_ODD;

} // End of ParityOnBytes()

PARITY Parity(unsigned char\* buf, int iBufLen) {

int sum = 0;

unsigned char iMask = 0x80;

for (int i = 0; i < iBufLen;) {

if ((\*buf & iMask) == iMask) {

sum++;

}

iMask >>= 1;

if (iMask == 0) {

iMask = 0x80;

buf++;

i++;

}

}

if (sum % 2 == 0) {

return(PAR\_EVEN);

}

return PAR\_ODD;

} // End of Parity()

Main.cpp

/\* main.cpp : Testing mainline for Lab #2

\* By: Michael Galle & Jack Cole

\*/

#include <stdio.h>

#include <string.h>

#include <malloc.h>

#include <stdlib.h>

#include "Parity.h" // Header file includes Prototypes defined in Parity.cpp

/\* PackToBits() : Utility function to convert the ASCII form to packed binary

\*

\* NOTE: malloc()'s the buffer to hold the packed binary, and

\* returns the pointer. CALLER SHOULD FREE THE BUFFER WHEN DONE.

\* Returns NULL if malloc() fails, or if string contains bad characters.

\* Returns the output length (rounded up) via pOutLen;

\*

\* NOTE2: Policy is that output buffer and length are rounded up to even bytes.

\*

\*/

static unsigned char \* PackToBits( char \*buf, int iBufLen, int \*pOutLen )

// static: in local variables, this is used to store the variable in statically allocated memory instead of automatically allocated memory

// static: in global variables and functions, this is used to set the scope to the containing file

{

unsigned char \*p, \*pCur;

int iOutLen;

unsigned char iMask;

int i;

// The packed form will be 1/8th as long as the ASCII form (one bit per char vs one byte), taken to the next largest integer

iOutLen = (iBufLen + 7) / 8; // iOutLen is the number of elements to be allocated

// Allocate the space for this

p = (unsigned char \*)calloc( iOutLen, sizeof(unsigned char)); // p is an unsigned char pointer --> Notice it is (casted) as an unsigned char pointer

// calloc() allocates the requested memory (and sets it to zero - unlike malloc() ) and returns a pointer to it, returns NULL if request fails

// Declaration of calloc(): void \* calloc(nitems, size) --> nitems = number of elements to allocate, size = size of each element

// Walk through the input string setting the bitset

pCur = p;

for( i=0, iMask = 0x80; i < iBufLen; ++i, ++buf ) {

// Check for a '1' or a '0' or something illegal

if (\*buf == '1') { \*pCur |= iMask; } // \*pCur dereferences pCur so indicates/sets the value it points to in memory

// a |= b is the same as a = a | b (a OR b) (if either is a 1 then the result is 1)

else if( \*buf != '0' ) { // calloc() sets allocated memory to zero so if not 0 then we have a buffer overrun

// ERROR: Must free p or we have a memory leak

free(p);

return(NULL);

}

// Prepare mask for next time, and chack for start of new byte

iMask >>= 1; // Shift all bits by one position to the right

if (iMask == 0) {

// Mask must be reset, and new data processed

iMask = 0x80;

++pCur;

}

}

// Done, we don't have to clear the rest of the final byte, because it was done in calloc().

\*pOutLen = iOutLen; // Store the value of iOutLen in the memory location pointed to by pOutLen (\* dereferences pointer)

return(p);

}

int main(int argc, char \*argv[])

{

int i, rc, iPackedLen;

unsigned char \*bPackedString;

unsigned char bZeros[6] = { 0 };

int iBit;

unsigned char iMask;

// Ensure there is at least one argument supplied by user

if (argc < 2) {

fprintf(stderr, "Usage: Parity <string1> <string2> ..., where strings must be a series of 1's and 0's.\n");

return(0);

}

// Test the byte-oriented (ASCII) version - 'ParityOnBytes' (LAB 1)

printf("Byte Oriented ASCII version - Lab 1 results: \n");

for (i = 1; i < argc; ++i) {

rc = ParityOnBytes(argv[i], strlen(argv[i]));

printf("For %s, ParityOnBytes() is %s.\n", argv[i], (rc == PAR\_EVEN) ? "even" : ((rc == PAR\_ODD) ? "odd" : "ERROR"));

}

printf("\n\n");

// Test the bit-oriented version - 'Parity' (LAB 2)

// To do this we first need to convert the ASCII characters to bits - Not trivial since the length could be ANYTHING ( need to use PackToBits() )

printf("Bit-oriented version - Lab 2 results: \n");

for( i=1; i < argc; ++i ) {

// Convert from string to packed bits

bPackedString = PackToBits( argv[i], strlen( argv[i] ), &iPackedLen ); // argv[i] is the ith char string input by user (i.e. 1100 ), i elements are separated by spaces

// &iPackedLen is the memory location of the int iPackedLen which is passed to PackToBits() and stores the value iOutLen (length of buf in bits)

if( bPackedString ) { // i.e. if not NULL

rc = Parity( bPackedString, iPackedLen );

printf("For %s, Parity() is %s. \n", argv[i], ( rc == PAR\_EVEN ) ? "even" : ( ( rc == PAR\_ODD ) ? "odd" : "ERROR"));

free( bPackedString);

} else {

fprintf(stderr, "Bad test string, or unable to allocate space for %s. \n", argv[i] );

}

}

printf("Press any key to end program ...");

getchar(); // Keep console open until user presses enter

return(0);

Parity.h

/\* Parity.h - Defines Enumerated data type PAR\_EVEN, PAR\_ODD, PAR\_ERROR

Copyright 2019 Stephane Durette

\*/

#ifndef PARITY\_H

#define PARITY\_H

enum PARITY {

PAR\_EVEN, PAR\_ODD, PAR\_ERROR

}; // Enumerated data type

// Function Prototype

PARITY ParityOnBytes(char\* buf, int iBufLen);

static unsigned char\* PackToBits(char\* buf, int iBufLen, int\* pOutLen);

PARITY Parity(unsigned char\* buf, int iBufLen);

#endif

Testing

