Exercise 1 *(6 points – C Program)*

```c
/*
 * ...the usual title block Student/Course/Assignment/Compiler information goes here...
 *
 * This file contains functions:
 *    main: Calls functions necessary to get each string, determine the hash
 *       bin, insert the string into the bin's tree, display all strings, and
 *       delete the hash table;
 *    SafeMalloc: Dynamically allocate memory; contains built-in failure test;
 *    OpenFile: Open file specified by its parameter in the read-only mode;
 *    BuildTree: Inserts a string into a specified tree or updates a node;
 *    PrintTree: Displays the strings in a specified tree;
 *    FreeTree: Frees a specified tree;
 *    HashFunction: Determines the proper hash bin for a string;
 *    CreateTable: Creates an empty hash table;
 *    PrintTable: Displays all strings in all hash table trees;
 *    FreeTable: Frees all trees in the hash table and the hash table itself;
 * This file also contains definitions of structure types NODE, BIN, & TABLE.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define LINE_LEN 256            /* size of input buffer */
#define BUFFMT "%255"           /* field width for input buffer scan */
#define MIN_ARGS 3              /* fewest command line arguments */
#define FILE_ARG_IX 1           /* index file name argument */
#define BINS_ARG_IX 2           /* index of bin count argument */

/*
 * A NODE structure is used to represent each node in the tree.
 */
typedef struct Node NODE;
struct Node
{
   char *strng;
   size_t count;                /* number of occurrences of this string */
   NODE *left, *right;          /* pointers to left and right children */
};

/*
 * A BIN structure type used as each hash table bin descriptor.
 */
typedef struct                  /* type of table array elements */
{
   size_t nodes;                /* # of list nodes for this bin */
   NODE *firstNode;             /* 1st node in this bin's list */
} BIN;

/*
 * The syntax and functionality of SafeMalloc is identical to that of malloc
 * with the following exception: If SafeMalloc fails to obtain the requested
 * memory it prints an error message to stderr and terminates the program
 * with an error code.
 */
```

```c
static void *SafeMalloc(size_t size)
{
   void *vp;

   /*
    * Request <size> bytes of dynamically-allocated memory and terminate the
    * program with an error message and code if the allocation fails.
    */
   if ((vp = malloc(size)) == NULL)
   {
      fputs("Out of memory\n", stderr);
      exit(EXIT_FAILURE);
   }
   return(vp);
}

/*
 * Open the file named in <fileName> in the "read only" mode and return its
 * FILE pointer if the open succeeds.  If it fails display an error message
 * and terminate the program with an error code.
 */
FILE *OpenFile(const char *fileName)
{
   FILE *fp;

   /* Open the file named in <fileName> in the read-only mode. */
   if ((fp = fopen(fileName, "r")) == NULL)
   {
      /* Print an error message and terminate with an error code. */
      fprintf(stderr, "File \"%s\" didn't open.\n", fileName);
      exit(EXIT_FAILURE);
   }
   return fp;
}

/*
 * BuildTree will search the binary tree at pNode for a node representing the
 * string in str.  If found, its string count will be incremented.  If not
 * found, a new node for that string will be created, put in alphabetical
 * order, and its count set to 1.  A pointer to the node for string str is
 * returned.
 */
NODE *BuildTree(NODE *pNode, char *str, BIN *pBin)
{
   if (pNode == NULL)                           /* string not found */
   {
      size_t length = strlen(str) + 1;          /* length of string */

      pNode = (NODE *)SafeMalloc(sizeof(NODE));  /* allocate a node */
      pNode->strng = (char *)SafeMalloc(length);
      memcpy(pNode->strng, str, length);        /* copy string */
      pNode->count = 1;                         /* 1st occurrence */
      pNode->left = pNode->right = NULL;        /* no subtrees */
      ++pBin->nodes;                        /* increment bin's node count */
   }
   else
   {
```

```
      int result = strcmp(str, pNode->strng);      /* compare strings */

      if (result == 0)                             /* new string == current */
         ++pNode->count;                           /* increment occurrence */
      else if (result < 0)                         /* new string < current */
         pNode->left = BuildTree(pNode->left, str, pBin);/* traverse left */
      else                                         /* new string > current */
         pNode->right = BuildTree(pNode->right, str, pBin);/* traverse right */
   }
   return(pNode);
}

/*
 * PrintTree recursively prints the binary tree in pNode alphabetically.
 */
void PrintTree(const NODE *pNode)
{
   if (pNode != NULL)                           /* if child exists */
   {
      PrintTree(pNode->left);                   /* traverse left */
      printf("%4d  %s\n", (int)pNode->count, pNode->strng);
      PrintTree(pNode->right);                  /* traverse right */
   }
}

/*
 * FreeTree recursively frees the binary tree in pNode.
 */
void FreeTree(NODE *pNode)
{
   if (pNode != NULL)                           /* if child exists */
   {
      FreeTree(pNode->left);                    /* traverse left */
      FreeTree(pNode->right);                   /* traverse right */
      free(pNode->strng);                       /* free the string */
      free(pNode);                              /* free the node */
   }
}

/*
 * A TABLE structure type used as the hash table descriptor.
 */
typedef struct
{
   size_t bins;                         /* bins in hash table */
   BIN *firstBin;                       /* first bin */
} TABLE;

/*
 * Returns a hash value in the range 0 through <bins>-1 based
 * upon the number of characters in the string in <key>.
 */
int HashFunction(const char *key, size_t bins)  /* get bin value from key */
{
   return((int)(strlen(key) % bins));    /* value is character count % bins */
}

```

```c
/*
 * CreateTable creates and initializes the hash table and its bins.
 */
TABLE *CreateTable(size_t bins)
{
   TABLE *hashTable;
   BIN *bin, *end;

   hashTable = (TABLE *)SafeMalloc(sizeof(TABLE));   /* alloc desc struct */
   hashTable->bins = bins;                           /* how many bins */
   /* alloc bins */
   hashTable->firstBin = (BIN *)SafeMalloc(bins * sizeof(BIN));
   end = hashTable->firstBin + bins;                 /* end of bins */

   for (bin = hashTable->firstBin; bin < end; ++bin) /* initialize bins */
   {
      bin->nodes = 0;                                /* no list nodes */
      bin->firstNode = NULL;                         /* no list */
   }
   return(hashTable);
}

/*
 * PrintTable prints the hash table.
 */
void PrintTable(const TABLE *hashTable)
{
   BIN *bin, *end;

   end = hashTable->firstBin + hashTable->bins;       /* end of bins */
   for (bin = hashTable->firstBin; bin < end; ++bin)  /* visit bins */
   {
      printf("%d entries for bin %d:\n",
         (int)bin->nodes, (int)(bin - hashTable->firstBin));
      /* visit nodes */
      PrintTree(bin->firstNode);
   }
}

/*
 * FreeTable frees the hash table.
 */
void FreeTable(TABLE *hashTable)
{
   BIN *bin, *end;

   end = hashTable->firstBin + hashTable->bins;     /* end of bins */
   for (bin = hashTable->firstBin; bin < end; ++bin)/* visit bins */
      FreeTree(bin->firstNode);
   free(hashTable->firstBin);                       /* free all bins */
   free(hashTable);                                 /* free table descriptor */
}

/*
 * The main function creates a hash table based upon the whitespace-separated
 * strings in the input file.  The input file and the number of bins desired
 * must be specified on the command line in that order.  After creation the
```

```
 * contents of the table are displayed and the table is freed.
 */
int main(int argc, char *argv[])
{
   char buf[LINE_LEN];              /* word string buffer */
   char fileName[LINE_LEN];         /* file name buffer */
   int howManyBins;                 /* number of bins to create */
   TABLE *hashTable;                /* pointer to hash table */
   FILE *fp;

   /* Read file name from command line */
   if (argc < MIN_ARGS || sscanf(argv[FILE_ARG_IX], BUFFMT "s", fileName) != 1)
   {
      fprintf(stderr, "No file name specified on command line\n");
      return EXIT_FAILURE;
   }
   fp = OpenFile(fileName);

   /* Read bin count from command line */
   if (sscanf(argv[BINS_ARG_IX], "%d", &howManyBins) != 1)
   {
      fprintf(stderr, "No bin count specified on command line\n");
      return EXIT_FAILURE;
   }
   hashTable = CreateTable((size_t)howManyBins);  /* allocate table array */

   /*
    * The following loop will read one string at a time from stdin until
    * EOF is reached.  For each string read the BuildList function will
    * be called to update the hash table.
    */
   while (fscanf(fp, BUFFMT "s", buf) != EOF) /* get string from file */
   {
      /* Set a pointer to the appropriate bin */
      BIN *pBin = &hashTable->firstBin[HashFunction(buf, (size_t)howManyBins)];
      pBin->firstNode = BuildTree(pBin->firstNode, buf, pBin); /* add string */
   }
   fclose(fp);
   PrintTable(hashTable);                     /* print all strings */
   FreeTable(hashTable);                      /* free the table */
   return EXIT_SUCCESS;
}
```

C2A7E1 Screen Shots are on the next page…

C2A7E1 Screen Shots

D:\Users\Ray\UCSD Courses\...

```
6 entries for bin 0:
   1  arguments.
   1  constants.
   1  expansion)
   1  invocation
   1  occurrence
   1  parameters
6 entries for bin 1:
   6  a
   1  combination
   1  definition,
   1  definition.
   1  number-sign
   1  stringizing
12 entries for bin 2:
   1  If
   1  It
   1  as
   2  by
   1  concatenated
   1  if
   5  in
   7  is
   3  it
   4  of
   1  or
   3  to
9 entries for bin 3:
   1  "stringizing"
   1  (#)
   1  Any
   3  The
   3  and
   1  any
   1  automatically
  13  the
   1  two
9 entries for bin 4:
   1  each
   1  from
   1  last
   2  only
   1  take
   1  that
   1  then
   1  used
   2  with
9 entries for bin 5:
   1  Thus,
   1  White
   1  first
   4  macro
   1  marks
   3  space
   2  token
```

D:\Users\Ray\UCSD Courses\...

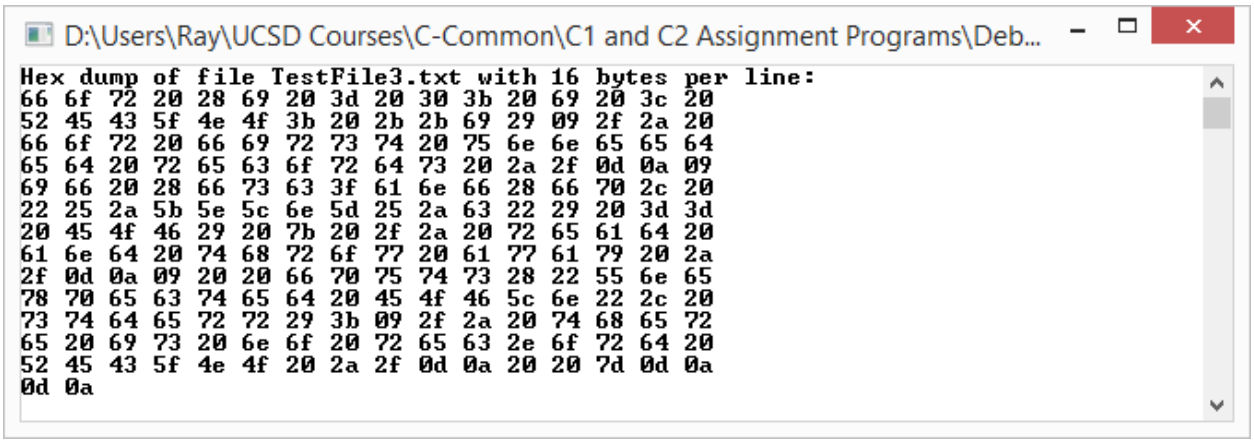
```
11 entries for bin 6:
   1  (after
   5  actual
   2  formal
   1  macros
   1  occurs
   1  passed
   2  single
   2  space.
   6  string
   2  tokens
   1  within
5 entries for bin 7:
   2  between
   1  comment
   2  literal
   2  reduced
   1  treated
10 entries for bin 8:
   1  adjacent
   4  argument
   1  converts
   1  enclosed
   1  ignored.
   2  literal.
   1  literals
   2  operator
   1  precedes
   1  replaces
7 entries for bin 9:
   1  argument,
   1  following
   2  parameter
   1  preceding
   1  quotation
   2  resulting
   1  separated
```

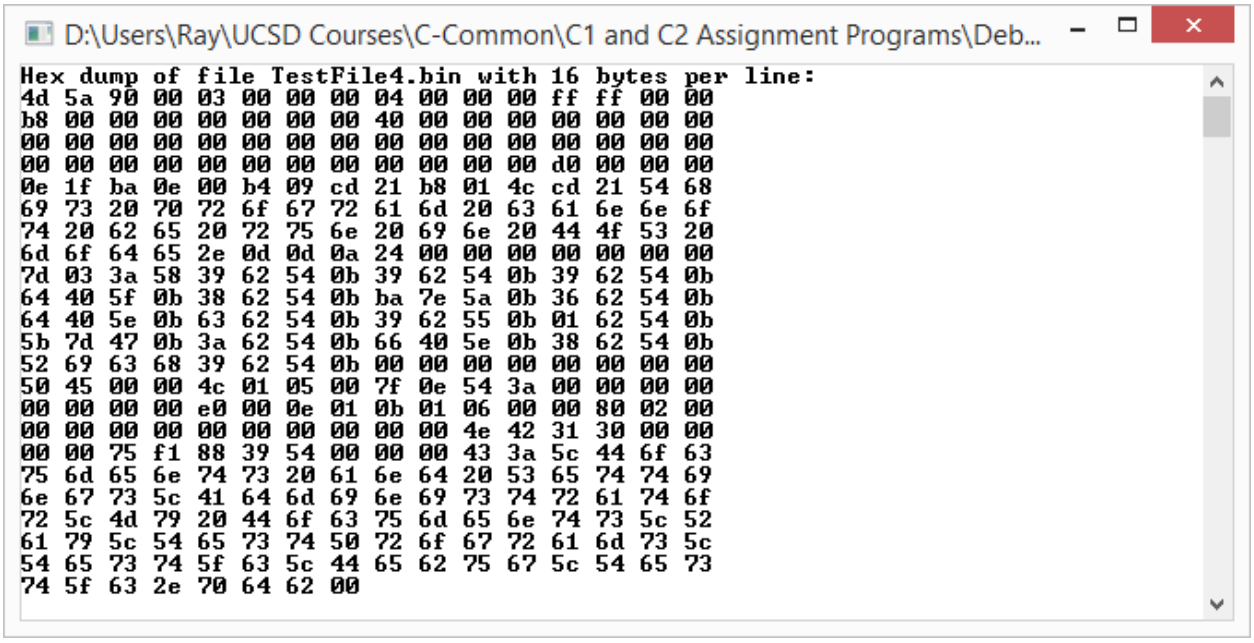
Exercise 2 *(4 points – C++ Program)*

```
   ********************************** FILE C2A7E2_OpenFileBinary.cpp **********************************
//
// ...the usual title block Student/Course/Assignment/Compiler information goes here...
//
// This file contains function OpenFileBinary, which opens a file
// in the binary read-only mode.
//

#include <cstdlib>
#include <fstream>
#include <iostream>
using namespace std;

//
// Open the file named in <fileName> using the object referenced by
// <inFile>.  If it fails display an error message and terminate the
// program with an error code.  The file must be opened in the binary
// mode.
//
void OpenFileBinary(const char *fileName, ifstream &inFile)
{
   // Open file for read only in the binary mode.
   inFile.open(fileName, ios_base::in | ios_base::binary);
   // If open fails print an error message and terminate with an error code.
   if (!inFile.is_open())
   {
      cerr << "File \"" << fileName << "\" didn't open.\n";
      exit(EXIT_FAILURE);
   }
}

    ************************************** FILE C2A7E2_ListHex.cpp **************************************
//
// ...the usual title block Student/Course/Assignment/Compiler information goes here...
//
// This file contains function ListHex, which displays the hexadecimal value of
// every byte in a file.
//

#include <fstream>
#include <iomanip>
#include <iostream>
using namespace std;

//
// Display the hexadecimal values of all bytes in the file in <inFile>.
// Each byte will be represented as two hexadecimal characters and there
// will be bytesPerLine bytes per line (except possibly on the last line).
// Bytes will be separated by 1 space and will be 0-filled on the left
// if the value of the byte does not exceed F.
//
void ListHex(ifstream &inFile, int bytesPerLine)
{
   cout << hex << setfill('0');              // set up display format
```

```
   int byte, bytesOnThisLine = 0;
   while ((byte = inFile.get()) != EOF)      // 1 byte/iteration until EOF
   {
      if (bytesOnThisLine != 0)              // if not first byte on line...
         cout << ' ';                        // ...display a leading space
      cout << setw(2) << byte;               // display the byte

      if (++bytesOnThisLine == bytesPerLine) // reset if at end of line
      {
         bytesOnThisLine = 0;
         cout << '\n';
      }
   }
   if (bytesOnThisLine != 0)                 // avoid a double newline
      cout << '\n';
}
```

C2A7E2 Screen Shots





C2A7E2 Screen Shots continue on the next page…

C2A7E2 Screen Shots, continued

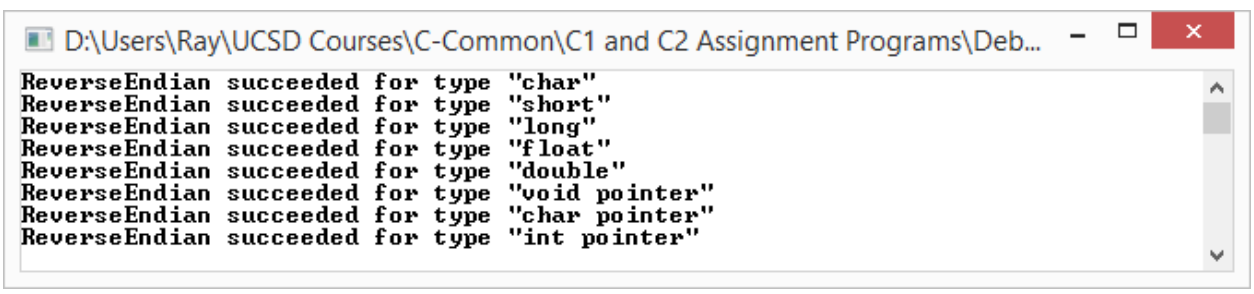
D:\Users\Ray\UCSD Courses\C-Common\C1 and C2 Assignment Programs\Deb...

```
Hex dump of file TestFile4.bin with 23 bytes per line:
4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00 b8 00 00 00 00 00 00
00 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 d0 00 00 00 0e 1f ba 0e 00
b4 09 cd 21 b8 01 4c cd 21 54 68 69 73 20 70 72 6f 67 72 61 6d 20 63
61 6e 6e 6f 74 20 62 65 20 72 75 6e 20 69 6e 20 44 4f 53 20 6d 6f 64
65 2e 0d 0d 0a 24 00 00 00 00 00 00 00 7d 03 3a 58 39 62 54 0b 39 62
54 0b 39 62 54 0b 64 40 5f 0b 38 62 54 0b ba 7e 5a 0b 36 62 54 0b 64
40 5e 0b 63 62 54 0b 39 62 55 0b 01 62 54 0b 5b 7d 47 0b 3a 62 54 0b
66 40 5e 0b 38 62 54 0b 52 69 63 68 39 62 54 0b 00 00 00 00 00 00 00
00 50 45 00 00 4c 01 05 00 7f 0e 54 3a 00 00 00 00 00 00 00 00 e0 00
0e 01 0b 01 06 00 00 80 02 00 00 00 00 00 00 00 00 00 00 00 4e 42 31
30 00 00 00 00 75 f1 88 39 54 00 00 00 43 3a 5c 44 6f 63 75 6d 65 6e
74 73 20 61 6e 64 20 53 65 74 74 69 6e 67 73 5c 41 64 6d 69 6e 69 73
74 72 61 74 6f 72 5c 4d 79 20 44 6f 63 75 6d 65 6e 74 73 5c 52 61 79
5c 54 65 73 74 50 72 6f 67 72 61 6d 73 5c 54 65 73 74 5f 63 5c 44 65
62 75 67 5c 54 65 73 74 5f 63 2e 70 64 62 00
```

Exercise 3 *(4 points – C Program)*

```c
/*
 * ...the usual title block Student/Course/Assignment/Compiler information goes here...
 *
 * This file contains function ReverseEndian, which reverses the byte order of
 * a specified object.
 */

#include <stddef.h>

/*
 * Reverse the endianness (big-to-little / little-to-big) of the <size>-byte
 * scalar object in <ptr>, then return <ptr>.
 */
void *ReverseEndian(void *ptr, size_t size)
{
   char *head, *tail;

   /*
    * Set <head> and <tail> to point to the bytes at each end of the object
    * in <ptr>.  If <head> is greater than <tail> swap the bytes they point
    * to then move <head> and <tail> toward each by 1 byte each.  Repeat
    * this process as long as <head> is greater than <tail>.
    */
   for (head = (char *)ptr, tail = head + size - 1; tail > head; --tail, ++head)
   {
      char temp = *head;
      *head = *tail;
      *tail = temp;
   }
   return ptr;
}
```

C2A7E3 Screen Shot



Exercise 4 *(6 points – C Program)*

```
   ********************************** FILE C2A7E4_OpenTemporaryFile.c **********************************
/*
 * ...the usual title block Student/Course/Assignment/Compiler information goes here...
 *
 * This file contains function OpenTemporaryFile, which opens a binary
 * read/write temporary file with an implementation defined name.
 */

#include <stdio.h>
#include <stdlib.h>

/*
 * Open a temporary file and return its FILE pointer if the open succeeds.
 * If it fails display an error message and terminate the program with an
 * error code.
 */
FILE *OpenTemporaryFile(void)
{
   FILE *fp;

   /* Open a temporary file and test for failure. */
   if ((fp = tmpfile()) == NULL)
   {
      fprintf(stderr, "Temporary file didn't open.\n");
      exit(EXIT_FAILURE);
   }
   return fp;
}


   ************************************ FILE C2A7E4_ReverseEndian.c ************************************
/*
 * ...the usual title block Student/Course/Assignment/Compiler information goes here...
 *
 * This file contains function ReverseEndian, which reverses the byte order of
 * a specified object.
 */

#include <stddef.h>

/*
 * Reverse the endianness (big-to-little / little-to-big) of the <size>-byte
 * scalar object in <ptr>, then return <ptr>.
 */
void *ReverseEndian(void *ptr, size_t size)
{
   char *head, *tail;

   /*
    * Set <head> and <tail> to point to the bytes at each end of the object
    * in <ptr>.  If <head> is greater than <tail> swap the bytes they point
    * to then move <head> and <tail> toward each by 1 byte each.  Repeat
    * this process as long as <head> is greater than <tail>.
    */
   for (head = (char *)ptr, tail = head + size - 1; tail > head; --tail, ++head)
```

```
   {
      char temp = *head;
      *head = *tail;
      *tail = temp;
   }
   return ptr;
}


   ********************************** FILE C2A7E4_ProcessStructures.c **********************************
/*
 * ...the usual title block Student/Course/Assignment/Compiler information goes here...
 *
 * This file contains the following functions, all of which operate on
 * structures of type "struct Test":
 *    ReverseStructure: Reverses the endianness of a structure's members;
 *    ReadStructures: Reads structures from a file;
 *    WriteStructures: Writess structures to a file;
 */

#include <stdio.h>
#include <stdlib.h>
#include "C2A7E4_Test-Driver.h"

void *ReverseEndian(void *ptr, size_t size);

/*
 * Reverse the endianness on the three scalar members of the structure
 * in <ptr> and return <ptr>.
 */
struct Test *ReverseMembersEndian(struct Test *ptr)
{
   ReverseEndian(&ptr->flt, sizeof(ptr->flt)); /* reverse the float member */
   ReverseEndian(&ptr->dbl, sizeof(ptr->dbl)); /* reverse the double member */
   ReverseEndian(&ptr->vp, sizeof(ptr->vp));   /* reverse the void* member */
   return ptr;
}

/*
 * Read the number of structures specified by <count> from the file in
 * <fp> and store them in the array in <ptr>, then return <ptr>.
 */
struct Test *ReadStructures(struct Test *ptr, size_t count, FILE *fp)
{
   /* Read the structure(s) & test for failure. */
   if (fread(ptr, sizeof(*ptr), count, fp) != count)
   {
      fprintf(stderr, "Structure read failed.\n");
      exit(EXIT_FAILURE);
   }
   return ptr;
}

/*
 * Write the number of structures specified by <count> from the array in
 * <ptr> and store them in the file in <fp>, then return <ptr>.
 */
```

```c
struct Test *WriteStructures(const struct Test *ptr, size_t count, FILE *fp)
{
   /* Write the structure(s) & test for failure. */
   if (fwrite(ptr, sizeof(*ptr), count, fp) != count)
   {
      fprintf(stderr, "Structure write failed.\n");
      exit(EXIT_FAILURE);
   }
   return (struct Test *)ptr;
}
```

C2A7E4 Screen Shot (Values and padding are implementation-dependent)