```
1   //
2   // Ray Mitchell, U99999999
3   // MeanOldTeacher@MeanOldTeacher.com
4   // C/C++ Programming II
5   // Section 149123, Ray Mitchell
6   // June 25, 2019
7   // C2A7E1_main.c
8   // Windows 10 Professional
9   // Visual Studio 2019 Professional
10  //
11  // This file contains functions:
12  //    main: Calls functions necessary to get each string, determine the hash
13  //       bin, insert the string into the bin's tree, display all strings, and
14  //       delete the hash table;
15  //    SafeMalloc: Dynamically allocate memory; contains built-in failure test.
16  //    OpenFile: Open file specified by its parameter in the read-only mode.
17  //    BuildTree: Inserts a string into a specified tree or updates a node.
18  //    PrintTree: Displays the strings in a specified tree.
19  //    FreeTree: Frees a specified tree.
20  //    HashFunction: Determines the proper hash bin for a string.
21  //    CreateTable: Creates an empty hash table.
22  //    PrintTable: Displays all strings in all hash table trees.
23  //    FreeTable: Frees all trees in the hash table and the hash table itself.
24  // This file also contains definitions of structure types NODE, BIN, & TABLE.
25  //
26
27  #include <stdio.h>
28  #include <stdlib.h>
29  #include <string.h>
30
31  #define LINE_LEN 256             // size of input buffer
32  #define BUFFMT "%255"            // field width for input buffer scan
33  #define MIN_ARGS 3               // fewest command line arguments
34  #define FILE_ARG_IX 1            // index file name argument
35  #define BINS_ARG_IX 2            // index of bin count argument
36
37  //
38  // A NODE structure is used to represent each node in the tree.
39  //
40  typedef struct Node NODE;
41  struct Node
42  {
43     char *strng;
44     size_t count;                 // number of occurrences of this string
45     NODE *left, *right;           // pointers to left and right children
46  };
47
48  //
49  // A BIN structure type used as each hash table bin descriptor.
50  //
51  typedef struct                   // type of table array elements
52  {
53     size_t nodes;                 // # of list nodes for this bin
54     NODE *firstNode;              // 1st node in this bin's list
55  } BIN;
56
57  //
58  // The syntax and functionality of SafeMalloc is identical to that of malloc
59  // with the following exception: If SafeMalloc fails to obtain the requested
60  // memory it prints an error message to stderr and terminates the program with
61  // an error code.
```

```
62   //
63   static void *SafeMalloc(size_t size)
64   {
65      void *vp;
66
67      //
68      // Request <size> bytes of dynamically allocated memory and terminate the
69      // program with an error message and code if the allocation fails.
70      //
71      if ((vp = malloc(size)) == NULL)
72      {
73         fputs("Out of memory\n", stderr);
74         exit(EXIT_FAILURE);
75      }
76      return(vp);
77   }
78
79   //
80   // Open the file named in <fileName> in the "read only" mode and return its
81   // FILE pointer if the open succeeds.  If it fails display an error message
82   // and terminate the program with an error code.
83   //
84   FILE *OpenFile(const char *fileName)
85   {
86      FILE *fp;
87
88      // Open the file named in <fileName> in the read-only mode.
89      if ((fp = fopen(fileName, "r")) == NULL)
90      {
91         // Print an error message and terminate with an error code.
92         fprintf(stderr, "File \"%s\" didn't open.\n", fileName);
93         exit(EXIT_FAILURE);
94      }
95      return fp;
96   }
97
98   //
99   // BuildTree will search the binary tree at pNode for a node representing the
100  // string in str.  If found, its string count will be incremented.  If not
101  // found, a new node for that string will be created, put in alphabetical order,
102  // and its count set to 1.  A pointer to the node for string str is returned.
103  //
104  NODE *BuildTree(NODE *pNode, char *str, BIN *pBin)
105  {
106     if (pNode == NULL)                              // string not found
107     {
108        size_t length = strlen(str) + 1;             // length of string
109
110        pNode = (NODE *)SafeMalloc(sizeof(NODE));    // allocate a node
111        pNode->strng = (char *)SafeMalloc(length);
112        memcpy(pNode->strng, str, length);           // copy string
113        pNode->count = 1;                            // 1st occurrence
114        pNode->left = pNode->right = NULL;           // no subtrees
115        ++pBin->nodes;                               // increment node count
116     }
117     else
118     {
119        int result = strcmp(str, pNode->strng);      // compare strings
120
121        if (result == 0)                             // new string == current
```

```
122            ++pNode->count;                              // increment occurrence
123         else if (result < 0)                            // new string < current
124            pNode->left = BuildTree(pNode->left, str, pBin);   // traverse left
125         else                                            // new string > current
126            pNode->right = BuildTree(pNode->right, str, pBin); // traverse right
127      }
128      return(pNode);
129   }
130
131   //
132   // PrintTree recursively prints the binary tree in pNode alphabetically.
133   //
134   void PrintTree(const NODE *pNode)
135   {
136      if (pNode != NULL)                                  // if child exists
137      {
138         PrintTree(pNode->left);                          // traverse left
139         printf("%4d  %s\n", (int)pNode->count, pNode->strng);
140         PrintTree(pNode->right);                         // traverse right
141      }
142   }
143
144   //
145   // FreeTree recursively frees the binary tree in pNode.
146   //
147   void FreeTree(NODE *pNode)
148   {
149      if (pNode != NULL)                                  // if child exists
150      {
151         FreeTree(pNode->left);                           // traverse left
152         FreeTree(pNode->right);                          // traverse right
153         free(pNode->strng);                              // free the string
154         free(pNode);                                     // free the node
155      }
156   }
157
158   //
159   // A TABLE structure type used as the hash table descriptor.
160   //
161   typedef struct
162   {
163      size_t bins;                                  // bins in hash table
164      BIN *firstBin;                                // first bin
165   } TABLE;
166
167   //
168   // Returns a hash value in the range 0 through <bins>-1 based upon the number of
169   // characters in the string in <key>.
170   //
171   int HashFunction(const char *key, size_t bins)// derive bin# from key
172   {
173      return((int)(strlen(key) % bins));         // bin# = character count % bins
174   }
175
176   //
177   // CreateTable creates and initializes the hash table and its bins.
178   //
179   TABLE *CreateTable(size_t bins)
180   {
181      TABLE *hashTable = (TABLE *)SafeMalloc(sizeof(TABLE));   // descriptor
```

```
182      hashTable->bins = bins;                                    // bin count
183      // alloc bins
184      hashTable->firstBin = (BIN *)SafeMalloc(bins * sizeof(BIN));
185      BIN *end = hashTable->firstBin + bins;                     // end of bins
186
187      for (BIN *bin = hashTable->firstBin; bin < end; ++bin)     // init. bins
188      {
189         bin->nodes = 0;                                         // no nodes
190         bin->firstNode = NULL;                                  // no list
191      }
192      return(hashTable);
193   }
194
195   //
196   // PrintTable prints the hash table.
197   //
198   void PrintTable(const TABLE *hashTable)
199   {
200      BIN *end = hashTable->firstBin + hashTable->bins;        // end of bins
201      for (BIN *bin = hashTable->firstBin; bin < end; ++bin)   // visit bins
202      {
203         printf("%d entries for bin %d:\n",
204            (int)bin->nodes, (int)(bin - hashTable->firstBin));
205         // visit nodes
206         PrintTree(bin->firstNode);
207      }
208   }
209
210   //
211   // FreeTable frees the hash table.
212   //
213   void FreeTable(TABLE *hashTable)
214   {
215      BIN *end = hashTable->firstBin + hashTable->bins;         // end of bins
216      for (BIN *bin = hashTable->firstBin; bin < end; ++bin)    // visit bins
217         FreeTree(bin->firstNode);
218      free(hashTable->firstBin);                          // free all bins
219      free(hashTable);                                    // free table descriptor
220   }
221
222   //
223   // The main function creates a hash table based upon the whitespace-separated
224   // strings in the input file.  The input file and the number of bins desired
225   // must be specified on the command line in that order.  After creation the
226   // contents of the table are displayed and the table is freed.
227   //
228   int main(int argc, char *argv[])
229   {
230      // Read file name from command line.
231      char fileName[LINE_LEN];
232      if (argc < MIN_ARGS || sscanf(argv[FILE_ARG_IX], BUFFMT "s", fileName) != 1)
233      {
234         fprintf(stderr, "No file name specified on command line\n");
235         return EXIT_FAILURE;
236      }
237      FILE *fp = OpenFile(fileName);
238
239      // Read bin count from command line.
240      int howManyBins;                                  // number of bins to create
241      if (sscanf(argv[BINS_ARG_IX], "%d", &howManyBins) != 1)
```

```
242      {
243         fprintf(stderr, "No bin count specified on command line\n");
244         return EXIT_FAILURE;
245      }
246      TABLE *hashTable = CreateTable((size_t)howManyBins);     // alloc table
247
248      //
249      // The following loop will read one string at a time from stdin until EOF is
250      // reached.  For each string read the BuildTree function will be called to
251      // update the hash table.
252      //
253      char buf[LINE_LEN];                          // word string buffer
254      while (fscanf(fp, BUFFMT "s", buf) != EOF)   // get string from file
255      {
256         // Set a pointer to the appropriate bin.
257         BIN *pBin = &hashTable->firstBin[HashFunction(buf, (size_t)howManyBins)];
258         pBin->firstNode = BuildTree(pBin->firstNode, buf, pBin); // add string
259      }
260      fclose(fp);
261      PrintTable(hashTable);                       // print all strings
262      FreeTable(hashTable);                        // free the table
263      return EXIT_SUCCESS;
264   }
```

```
1   //
2   // Ray Mitchell, U99999999
3   // MeanOldTeacher@MeanOldTeacher.com
4   // C/C++ Programming II
5   // Section 149123, Ray Mitchell
6   // June 25, 2019
7   // C2A7E2_ListHex.cpp
8   // Windows 10 Professional
9   // Visual Studio 2019 Professional
10  //
11  // This file contains function ListHex, which displays the hexadecimal value of
12  // every byte in a file.
13  //
14
15  #include <fstream>
16  #include <iomanip>
17  #include <iostream>
18  using namespace std;
19
20  //
21  // Display the hexadecimal values of all bytes in the file in <inFile>. Each
22  // byte will be represented as two hexadecimal characters and there will be
23  // bytesPerLine bytes per line (except possibly on the last line). Bytes will be
24  // separated by 1 space and will be 0-filled on the left if the value of the
25  // byte does not exceed F.
26  //
27
28  // Version 1: Reads block-at-a-time
29  void ListHex(ifstream &inFile, int bytesPerLine)
30  {
31     char *bytePtr = new char[(unsigned)bytesPerLine];  // for a line of bytes
32     cout << hex << setfill('0');                       // set up display format
33     do
34     {
35        bool lineIsEmpty = true;
36
37        // Read bytesPerLine maximum
38        inFile.read(bytePtr, bytesPerLine);
39        for (int byteIx = 0; byteIx < inFile.gcount(); ++byteIx)
40        {
41           if (!lineIsEmpty)                       // if not first byte on line...
42              cout << ' ';                         // ...display a leading space
43           else
44              lineIsEmpty = false;
45           // display the byte
46           cout << setw(2) << (int)(unsigned char)bytePtr[byteIx];
47        }
48        if (!lineIsEmpty)                          // avoid an empty line
49           cout << '\n';
50     } while (inFile.gcount() == bytesPerLine);
51
52     if (inFile.gcount())                          // avoid a double newline
53        cout << '\n';
54     delete[] bytePtr;
55  }
56
57  #if 0
58  // Version 2: Reads byte-at-a-time
59  static void ListHex(ifstream &inFile, int bytesPerLine)
60  {
61     cout << hex << setfill('0');              // set up display format
```

```
62      int byte, bytesOnThisLine = 0;
63      while ((byte = inFile.get()) != EOF)      // 1 byte/iteration until EOF
64      {
65         if (bytesOnThisLine != 0)              // if not first byte on line...
66            cout << ' ';                        // ...display a leading space
67         cout << setw(2) << byte;               // display the byte
68
69         if (++bytesOnThisLine == bytesPerLine) // reset if at end of line
70         {
71            bytesOnThisLine = 0;
72            cout << '\n';
73         }
74      }
75      if (bytesOnThisLine != 0)                 // avoid a double newline
76         cout << '\n';
77   }
78   #endif
```

```
1   //
2   // Ray Mitchell, U99999999
3   // MeanOldTeacher@MeanOldTeacher.com
4   // C/C++ Programming II
5   // Section 149123, Ray Mitchell
6   // June 25, 2019
7   // C2A7E2_OpenFileBinary.cpp
8   // Windows 10 Professional
9   // Visual Studio 2019 Professional
10  //
11  // This file contains function OpenFileBinary, which opens a file in the binary
12  // read-only mode.
13  //
14
15  #include <cstdlib>
16  #include <fstream>
17  #include <iostream>
18  using namespace std;
19
20  //
21  // Open the file named in <fileName> using the object referenced by <inFile>.
22  // If it fails display an error message and terminate the program with an error
23  // code.  The file must be opened in the binary mode.
24  //
25  void OpenFileBinary(const char *fileName, ifstream &inFile)
26  {
27     // Open file for read only in the binary mode.
28     inFile.open(fileName, ios_base::binary);
29     // If open fails print an error message and terminate with an error code.
30     if (!inFile.is_open())
31     {
32        cerr << "File \"" << fileName << "\" didn't open.\n";
33        exit(EXIT_FAILURE);
34     }
35  }
```

```c
//
// Ray Mitchell, U99999999
// MeanOldTeacher@MeanOldTeacher.com
// C/C++ Programming II
// Section 149123, Ray Mitchell
// June 25, 2019
// C2A7E3_ReverseEndian.c
// Windows 10 Professional
// Visual Studio 2019 Professional
//
// 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)
{
   //
   // 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 (char *head = (char *)ptr, *tail = head + size - 1;
      tail > head; --tail, ++head)
   {
      char temp = *head;
      *head = *tail;
      *tail = temp;
   }
   return ptr;
}
```

```c
//
// Ray Mitchell, U99999999
// MeanOldTeacher@MeanOldTeacher.com
// C/C++ Programming II
// Section 149123, Ray Mitchell
// June 25, 2019
// C2A7E4_OpenTemporaryFile.c
// Windows 10 Professional
// Visual Studio 2019 Professional
//
// 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)
{
   // Open a temporary file and test for failure.
   FILE *fp;
   if ((fp = tmpfile()) == NULL)
   {
      fprintf(stderr, "Temporary file didn't open.\n");
      exit(EXIT_FAILURE);
   }
   return fp;
}
```

```c
//
// Ray Mitchell, U99999999
// MeanOldTeacher@MeanOldTeacher.com
// C/C++ Programming II
// Section 149123, Ray Mitchell
// June 25, 2019
// C2A7E4_ProcessStructures.c
// Windows 10 Professional
// Visual Studio 2019 Professional
//
// 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: Writes 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>.
//
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);
```

```
62      }
63      return (struct Test *)ptr;
64  }
```

```
1   //
2   // Ray Mitchell, U99999999
3   // MeanOldTeacher@MeanOldTeacher.com
4   // C/C++ Programming II
5   // Section 149123, Ray Mitchell
6   // June 25, 2019
7   // C2A7E4_ReverseEndian.c
8   // Windows 10 Professional
9   // Visual Studio 2019 Professional
10  //
11  // This file contains function ReverseEndian, which reverses the byte order of
12  // a specified object.
13  //
14
15  #include <stddef.h>
16
17  //
18  // Reverse the endianness (big-to-little / little-to-big) of the <size>-byte
19  // scalar object in <ptr>, then return <ptr>.
20  //
21  void *ReverseEndian(void *ptr, size_t size)
22  {
23     //
24     // Set <head> and <tail> to point to the bytes at each end of the object in
25     // <ptr>.  If <head> is greater than <tail> swap the bytes they point to then
26     // move <head> and <tail> toward each by 1 byte each.  Repeat this process as
27     // long as <head> is greater than <tail>.
28     //
29     for (char *head = (char *)ptr, *tail = head + size - 1;
30        tail > head; --tail, ++head)
31     {
32        char temp = *head;
33        *head = *tail;
34        *tail = temp;
35     }
36     return ptr;
37  }
```