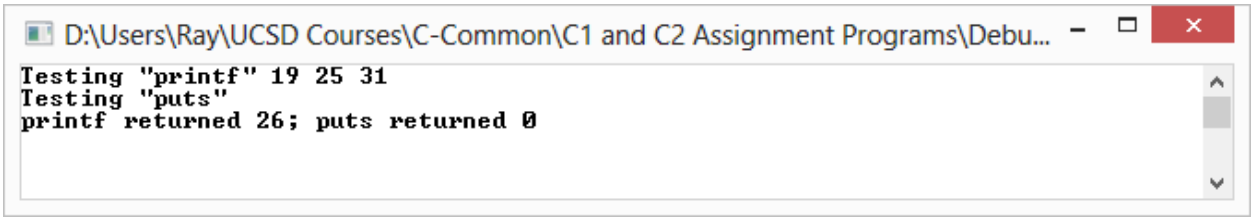
Exercise 1 *(2 points – C Program)*

```c
/*
 * ...the usual title block Student/Course/Assignment/Compiler information goes here...
 *
 * This file contains functions:
 *    GetPrintfPointer: Creates and returns a pointer to the printf function;
 *    GetPutsPointer: Creates and returns a pointer to the puts function;
 */

#include <stdio.h>

/*
 * This function returns a pointer to the standard library
 * printf function and does nothing else.
 */
int (*GetPrintfPointer(void))(const char *format, ...)
{
   int (*pPrintf)(const char *format, ...) = printf;
   return pPrintf;
}

/*
 * This function returns a pointer to the standard library
 * puts function and does nothing else.
 */
int (*GetPutsPointer(void))(const char *str)
{
   int (*pPuts)(const char *str) = puts;
   return pPuts;
}
```

C2A6E1 Screen Shot

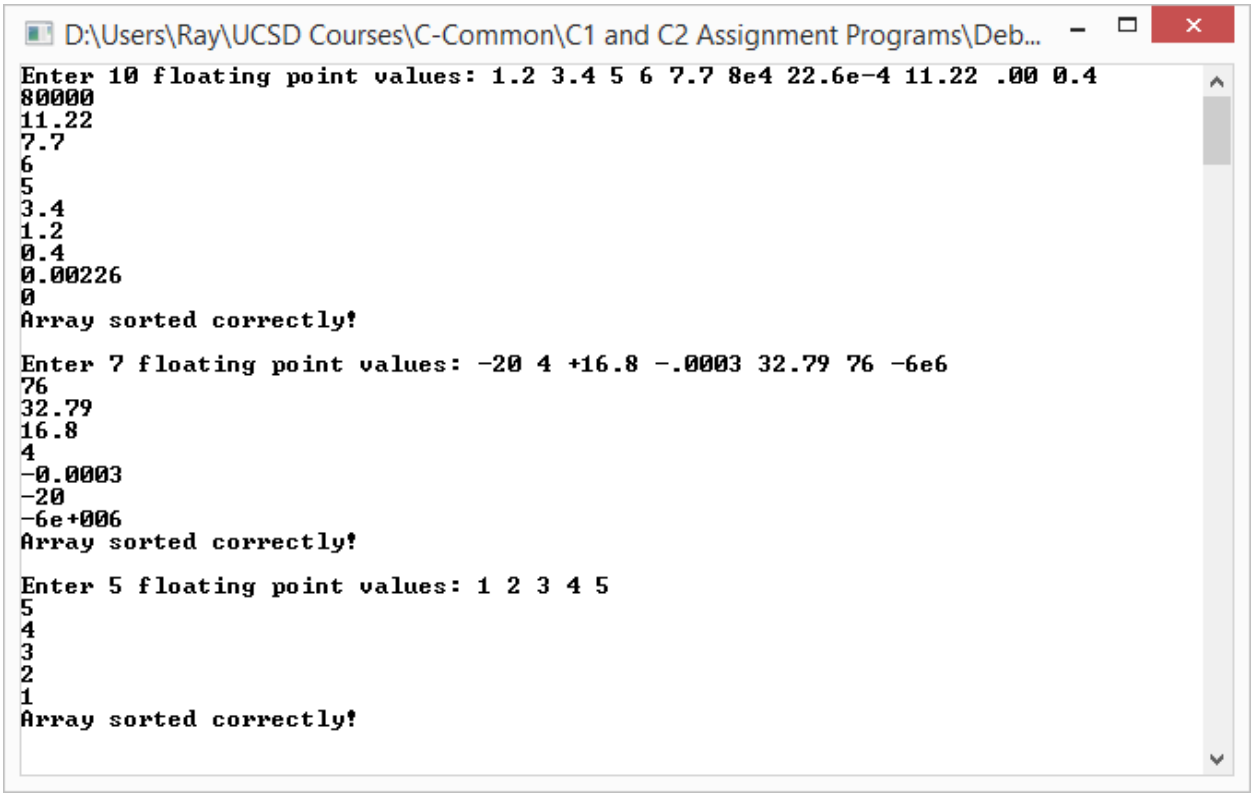


Exercise 2 *(4 points – C++ Program)*

```
  ************************************** FILE C2A6E2_GetValues.cpp **************************************
//
// ...the usual title block Student/Course/Assignment/Compiler information goes here...
//
// This file contains function GetValues, which prompts the user to input
// values used to populate the array represented by its first parameter.
//

#include <iostream>

//
// Prompt the user to enter the number of floating point values specified
// by <elements> and store them in the array in <first>.  <first> is
// returned.
//
float *GetValues(float *first, size_t elements)
{
   std::cout << "Enter " << int(elements) << " floating point values: ";
   // Get one user entry per iteration and store in successive array elements.
   for (float *end = first + elements; first < end; ++first)
      std::cin >> *first;
   return first - elements;
}


  ************************************** FILE C2A6E2_SortValues.cpp **************************************
//
// ...the usual title block Student/Course/Assignment/Compiler information goes here...
//
// This file contains function SortValues, which sorts the elements of
// the array represented by its first parameter.
//

#include <cstddef>

//
// Sort the <elements> in the array in <first> in descending numerical
// order using the Bubble Sort algorithm.  <first> is returned.
//
float *SortValues(float *first, size_t elements)
{
   float *last = &first[elements - 1];
   bool swapped;

   // Loop until no swap occurs during a complete pass through the array.
   do
   {
      swapped = false;
      //
      // One complete set of loop iterations represents one complete pass
      // through the array.
      //
      for (float *ptr = first, *next = first + 1; ptr < last; ++ptr, ++next)
      {
```

```
            if (*ptr < *next)       // need to exchange
            {
               float temp = *ptr;   // do the...
               *ptr = *next;        // ...3 step...
               *next = temp;        // ...swap
               swapped = true;      // indicate swap occurred
            }
         }
         --last;
      } while (swapped);

      return first;
   }
```

C2A6E2 Screen Shot

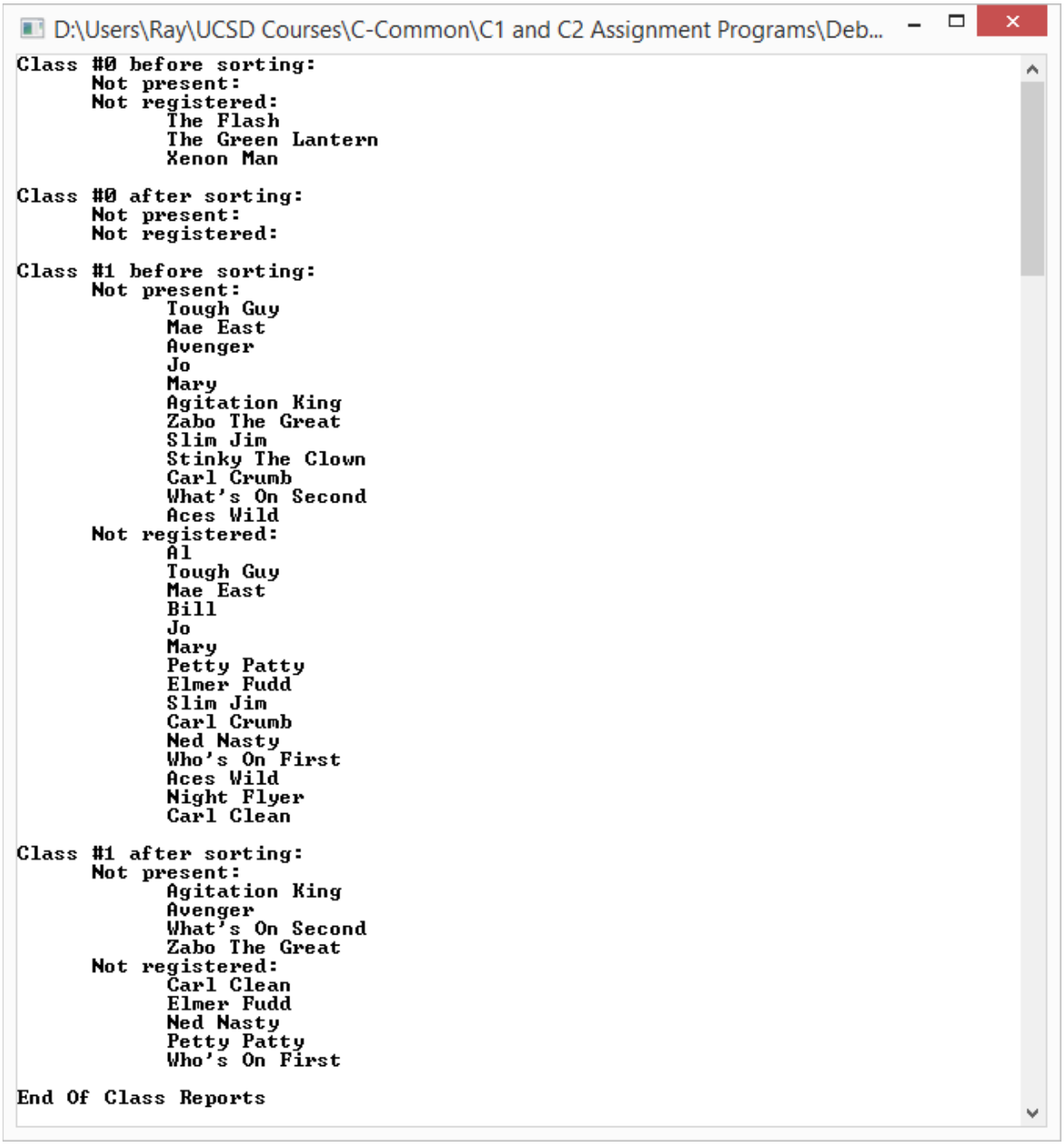


Exercise 3 *(6 points – C Program)*

```c
/*
 * ...the usual title block Student/Course/Assignment/Compiler information goes here...
 *
 * This file contains functions:
 *    Compare: Compares the strings represented by its two parameters;
 *    SortStudents: Sorts strings in the array represented its 1st parameter;
 *    DisplayClassStatus: Displays specific strings from the arrays represented
 *       by its two pointer parameters.
 */

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

/*
 * Compare the C-strings in <elemA> and <elemB> using the standard library
 * function strcmp and return the result directly from strcmp.
 */
int Compare(const void *elemA, const void *elemB)
{
   return(strcmp(*(char **)elemA, *(char **)elemB));
}

/*
 * Call the standard library function qsort to sort <studentCount> C-strings
 * in the array in <studentList>.
 */
void SortStudents(const char *studentList[], size_t studentCount)
{
   qsort((void *)studentList, studentCount, sizeof(*studentList), Compare);
}

/*
 * Use the standard library bsearch function to search the array of C-strings
 * in <attendees> for each C-string in the array in <registrants>.  Then use
 * the standard library bsearch function to search the array of C-strings in
 * <registrants> for each C-string in the array in <attendees>.  In each
 * case display the C-strings NOT found.
 */
void DisplayClassStatus(const char *registrants[], size_t registrantCount,
                        const char *attendees[], size_t attendeeCount)
{
   const char **cpp, **end;

   /*
    * For each name searched for bsearch returns a pointer to the appropriate
    * element if the name is found and a null pointer if not.  Each element
    * is a pointer to a char and, therefore, the first bsearch argument must
    * be a pointer to a pointer to a char (type casted to a void pointer).
    */

   /* See how many registrants are listed in the attendees array. */
   printf("      Not present:\n");
   for (cpp = registrants, end = cpp + registrantCount; cpp < end; ++cpp)
      /* If the name was not found in the array, display it. */
```

```
      if (!bsearch((void *)cpp, (void *)attendees, attendeeCount,
                   sizeof(*attendees), Compare))
         printf("            %s\n", *cpp);

   /* See how many attendees are listed in the registrants array. */
   printf("      Not registered:\n");
   for (cpp = attendees, end = cpp + attendeeCount; cpp < end; ++cpp)
      /* If the name was not found in the array, display it. */
      if (!bsearch((void *)cpp, (void *)registrants, registrantCount,
                   sizeof(*registrants), Compare))
         printf("            %s\n", *cpp);
}
```

C2A6E3 Screen Shot



Exercise 4 *(8 points – C Program)*

```
  ****************************************** FILE C2A6E4_OpenFile.c ******************************************
/*
 * ...the usual title block Student/Course/Assignment/Compiler information goes here...
 *
 * This file contains function OpenFile, which opens the file specified by its
 * parameter in the read-only mode.
 */

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

/*
 * 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;
}

  ******************************************** FILE C2A6E4_List.c ********************************************
/*
 * ...the usual title block Student/Course/Assignment/Compiler information goes here...
 *
 * This file contains functions:
 *    SafeMalloc: Dynamically allocates memory; built-in failure testing;
 *    CreateList: Creates a singly-linked list from text file strings;
 *    PrintList: Prints contents of singly-linked list of text file strings;
 *    FreeList: Frees the singly-linked list of text file strings;
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "C2A6E4_List-Driver.h"

#define BUFSIZE 256                             /* size of input buffer */
#define BUFFMT "%255"                           /* scanf field for buffer */

/*
 * 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);
}

/*
 * Create a singly-linked list where each node represents a unique
 * whitespace-separated string from the text file in <fp>.  A new node is
 * created and pushed at the head of the list if a string not already in
 * the list is read from the file.  If a node for that string already
 * exists its occurrence count is merely incremented.
 */
List *CreateList(FILE *fp)
{
   List *head;                                  /* pointer into list */
   char buf[BUFSIZE];                           /* for string input */

   /* loop to get strings for insertion at top of list */
   for (head = NULL; fscanf(fp, BUFFMT "s", buf) != EOF;)
   {
      List *p;                                  /* pointer into list */
      /* loop to find duplicates; order of logical && operands is critical */
      for (p = head; p != NULL && strcmp(p->str, buf); p = p->next)
         ;
      if (p != NULL)                            /* found same string */
         ++p->count;                            /* incr. string count */
      else                                      /* add new string at top */
      {
         size_t length = strlen(buf) + 1;       /* string len + the '\0' */;

         p = (List *)SafeMalloc(sizeof(List));  /* allocate new node */
         p->str = (char *)SafeMalloc(length);   /* alloc string storage */
         memcpy(p->str, buf, length);           /* copy string */
         p->count = 1;                          /* init. string count */
         /* Push the node onto the list (2 steps). */
         p->next = head;                        /* next = head pointer */
         head = p;                              /* point head to node */
      }
   }
   return head;
}

/*
 * Print the string and the number of occurrences of it represented by each
 * node in the list in head-to-tail order.
 */
```

```
List *PrintList(const List *head)
{
   const List *p;                                    /* pointer into list */

   for (p = head; p != NULL; p = p->next)            /* printing loop */
      printf("%-15s%4d ea\n", p->str, p->count);     /* string & count */

   return (List *)head;
}


/*
 * Free all dynamically-allocated memory in the list in head-to-tail order.
 */
void FreeList(List *head)
{
   while (head)                        /* loop to free dynamic storage */
   {
      List *pTmp = head;               /* save pointer to current node */

      head = head->next;               /* get pointer to next node */
      free(pTmp->str);                 /* free current node's str storage */
      free(pTmp);                      /* free current node's storage */
   }
}
```

C2A6E4 Screen Shots are on the next page…

C2A6E4 Screen Shots

D:\Users\Ray\UCSD ...

```
separated          1 ea
which              1 ea
from               1 ea
literals           1 ea
adjacent           1 ea
any                1 ea
concatenated       1 ea
automatically      1 ea
space.             2 ea
argument,          1 ea
two                1 ea
occurs             1 ea
comment            1 ea
if                 1 ea
Thus,              1 ea
resulting          2 ea
single             2 ea
reduced            2 ea
tokens             2 ea
between            2 ea
white              4 ea
Any                1 ea
ignored.           1 ea
last               1 ea
following          1 ea
token              2 ea
first              1 ea
preceding          1 ea
space              3 ea
White              1 ea
definition.        1 ea
within             1 ea
stringizing        1 ea
combination        1 ea
of                 4 ea
occurrence         1 ea
each               1 ea
replaces           1 ea
then               1 ea
literal            2 ea
literal.           2 ea
as                 1 ea
treated            1 ea
and                3 ea
marks              1 ea
```

D:\Users\Ray\UCSD ...

```
quotation          1 ea
enclosed           1 ea
invocation         1 ea
by                 2 ea
passed             1 ea
argument           4 ea
actual             5 ea
definition,        1 ea
the               13 ea
in                 5 ea
parameter          2 ea
formal             2 ea
a                  6 ea
precedes           1 ea
it                 3 ea
If                 1 ea
arguments.         1 ea
take               1 ea
that               1 ea
macros             1 ea
with               2 ea
only               2 ea
used               1 ea
is                 7 ea
It                 1 ea
constants.         1 ea
string             6 ea
to                 3 ea
expansion)         1 ea
(after             1 ea
parameters         1 ea
macro              4 ea
converts           1 ea
(#)                1 ea
operator           2 ea
"stringizing"      1 ea
or                 1 ea
number-sign        1 ea
The                3 ea
```