Assignment 6

C/C++ Programming II

Exercise 1 *(2 points – C Program)*

Exclude any existing source code files that may already be in your IDE project and add a new one, naming it **C2A6E1_GetPointers.c**. Also add instructor-supplied source code file **C2A6E1_main-Driver.c**. <u>Do not write a `main` function!</u> `main` already exists in the instructor-supplied file and it will use the code you write.

File **C2A6E1_GetPointers.c** must contain functions named `GetPrintfPointer` and `GetPutsPointer`.

`GetPrintfPointer` syntax:

```
   int (*GetPrintfPointer(void))(const char *format, ...);
```

Parameters:
    none
Synopsis:
    Declares a pointer named `pPrintf` of appropriate type to point to the standard library `printf` function and initializes it to point to that function.
Return:
    the initialized pointer named `pPrintf`, which points to the standard library `printf` function

`GetPutsPointer` syntax:

```
   int (*GetPutsPointer(void))(const char *str);
```

Parameters:
    none
Synopsis:
    Declares a pointer named `pPuts` of appropriate type to point to the standard library `puts` function and initializes it to point to that function.
Return:
    the initialized pointer named `pPuts`, which points to the standard library `puts` function

Never explicitly write a prototype for a library function. Instead, use `#include` to include the appropriate standard library header file, which will already contain the needed prototype.

Submitting your solution

Send both source code files to the Assignment Checker with the subject line **C2A6E1_*ID***, where ***ID*** is your 9-character UCSD student ID.

See the course document titled "Preparing and Submitting Your Assignments" for additional exercise formatting, submission, and Assignment Checker requirements.

Hints:
Look up the standard library `printf` and `puts` functions in your IDE's built-in help, any good C programming text book, or online, and examine their prototypes.

Exercise 2 *(4 points – C++ Program)*

Exclude any existing source code files that may already be in your IDE project and add two new ones, naming them **C2A6E2_GetValues.cpp** and **C2A6E2_SortValues.cpp**. Also add instructor-supplied source code file **C2A6E2_main-Driver.cpp**. Do not write a main function! main already exists in the instructor-supplied file and it will use the code you write.

File **C2A6E2_GetValues.cpp** must contain a function named GetValues.
GetValues syntax:

```
   float *GetValues(float *first, size_t elements);
```

Parameters:
first – a pointer to the first element of an array of **float**s
elements – the number of elements in that array

Synopsis:
Prompts the user to input elements whitespace-separated floating point values, which it then reads with cin and stores into the successive elements of the array in first starting with element 0.

Return:
a pointer to the first element of the array

File **C2A6E2_SortValues.cpp** must contain a function named SortValues.
SortValues syntax:

```
   float *SortValues(float *first, size_t elements);
```

Parameters:
first – a pointer to the first element of an array of **float**s
elements – the number of elements in that array

Synopsis:
Sorts the array in first in descending order using the “bubble sort” algorithm

Return:
a pointer to the first element of the sorted array

- Use no global variables or global information about the array in either function.
- Do not use the syntax pointer[offset] or *(pointer + offset) to access array elements. Use *pointer or *pointer++ instead.
- Use the following test values. Copying/pasting them from this document to the user prompt is an easy way to avoid retyping them each time:
  1st prompt: `1.2   3.4   5   6   7.7   8e4   22.6e-4   11.22   .00   0.4`
  2nd prompt: `-20   4   +16.8   -.0003   32.79   76   -6e6`
  3rd prompt: `1  2  3  4  5`

Submitting your solution

Send all three source code files to the Assignment Checker with the subject line **C2A6E2_*ID***, where ***ID*** is your 9-character UCSD student ID.

See the course document titled “Preparing and Submitting Your Assignments” for additional exercise formatting, submission, and Assignment Checker requirements.

Hints:
Do not declare any arrays or create any dynamically.

Exercise 3 *(6 points – C Program)*

Exclude any existing source code files that may already be in your IDE project and add a new one, naming it **C2A6E3_DisplayClassStatus.c**. Also add instructor-supplied source code file **C2A6E3_main-Driver.c**. <u>Do not write a main function!</u> main already exists in the instructor-supplied file and it will use the code you write.

A certain school keeps two sets of student names for every class taught. One set is for individuals who have registered (registrants) and the other is for individuals (registered or not) who have attended the first class meeting (attendees). Each set is kept in an appropriately-named ragged array as follows:

```
    const char *names[] = { "Al", "Ned Nasty", "Sweet L. Sally", etc. };
```

File **C2A6E3_DisplayClassStatus.c** must contain functions named `Compare`, `SortStudents`, and `DisplayClassStatus`.

`Compare` syntax:

```
    int Compare(const void *elemA, const void *elemB);
```

Parameters:

`elemA` – a pointer to an element of a *names* array
`elemB` – a pointer to an element of a *names* array

Synopsis:

Compares the names represented by `elemA` and `elemB` using the standard library function `strcmp`.

Return:

<0 if the name represented by `elemA` is less than the name represented by `elemB`;
0 if the name represented by `elemA` is equal to the name represented by `elemB`;
>0 if the name represented by `elemA` is greater than the name represented by `elemB`.

`SortStudents` syntax:

```
    void SortStudents(const char *studentList[], size_t studentCount);
```

Parameters:

`studentList` – A pointer to the first element of a *names* array
`studentCount` – The number of elements in the array

Synopsis:

Uses the standard library `qsort` function and the `Compare` function above to sort the array in `studentList` into alphabetical order. No variables other than the two parameters may be declared.

Return:

void

`DisplayClassStatus` syntax:

```
    void DisplayClassStatus( const char *registrants[], size_t registrantCount,
                             const char *attendees[],   size_t attendeeCount);
```

Parameters:

`registrants` – pointer to the first element of a registrants *names* array
`registrantCount` – the number of elements in the registrants *names* array
`attendees` – pointer to the first element of an attendees *names* array
`attendeeCount` – the number of elements in the attendees *names* array

Synopsis:

1. Determines and displays which of the registrants did not attend the first meeting by repeatedly calling the standard library `bsearch` function to search the attendees array for each name in the registrants array.

2. Determines and displays which of the attendees were not registered by repeatedly calling `bsearch` to search the registrants array for each name in the attendees array.
3. Do not sort any arrays. Simply search them “as is”.
4. Results must be displayed in the following format, using the phrases “Not present:” and “Not registered:” as shown to differentiate the two groupings.

```
Not present:
   Orphan Annie
   Toto The Dog
   Madonna
Not registered:
   Little Mary
   Big John
   Tiny Tim
```

Return:

void

The same comparison function must be used for both `qsort` and `bsearch`.

IMPORTANT: One purpose of this exercise is to illustrate the erroneous results that are usually obtained when `bsearch` is used on an unsorted array. My driver code will accomplish this by calling your `DisplayClassStatus` function both before and after calling your `SortStudents` function. For this reason your `DisplayClassStatus` function must do no sorting.

Submitting your solution

Send both source code files to the Assignment Checker with the subject line **C2A6E3_*ID***, where ***ID*** is your 9-character UCSD student ID.

See the course document titled “Preparing and Submitting Your Assignments” for additional exercise formatting, submission, and Assignment Checker requirements.

Hints:

The first argument of `bsearch` must always be the address of (a pointer to) the object to be searched for, not the value of that object.

Exercise 4 *(8 points – C Program)*

Exclude any existing source code files that may already be in your IDE project and add two new ones, naming them **C2A6E4_OpenFile.c** and **C2A6E4_List.c**. Also add instructor-supplied source code files **C2A6E4_List-Driver.h** and **C2A6E4_main-Driver.c**. Do not write a `main` function! `main` already exists in the instructor-supplied implementation file and it will use the code you write.

Regarding data type `List`, which is used in this exercise…

`List` is a typedef'd data type that is defined in instructor-supplied header file

C2A6E4_List-Driver.h

Any file that uses this data type must include this header file using `#include`.

File **C2A6E4_OpenFile.c** must contain a function named `OpenFile`.

`OpenFile` syntax:

```
FILE *OpenFile(const char *fileName);
```

Parameters:

`fileName` – a pointer to the name of the file to be opened

Synopsis:

Opens the file named in `fileName` in the read-only text mode. If the open fails an error message is output to `stderr` and the program is terminated with an error exit code. The error message must mention the name of the failing file.

Return:

a pointer to the open file if the open succeeds; otherwise, the function does not return.

File **C2A6E4_List.c** must contain functions named `CreateList`, `PrintList`, `FreeList`.

`CreateList` syntax:

```
List *CreateList(FILE *fp);
```

Parameter:

`fp` – a pointer to an open text file containing zero or more whitespace-separated words (strings)

Synopsis:

Creates a singly-linked list from strings it reads from the text file represented by parameter `fp`. Each list node represents a unique case-dependent string and the number of times it occurred in the file. This is the simplest algorithm and is recommended:

```
1. Attempt to read a string from the file. If successful:
    A. Search the list for that string.
        i. If found:
            a. Increment the node’s string count.
        ii. else:
            a. Allocate a new node, and then
            b. allocate memory for the string (including its \0), point the node’s char pointer to that allocation, and copy the string into it.
            c. Set the node’s string count to 1.
            d. Push the node onto the list.
    B. Repeat from step 1.
2. Else, return the list’s “head” pointer.
```

Return:

the list's head pointer.

Examples – Number of nodes created if file contains:

`Fly fly!` (2 nodes)   `Fly Fly!` (2 nodes)   `Fly fly !` (3 nodes)   `Fly Fly !` (2 nodes)

`PrintList` syntax:

```
List *PrintList(const List *head);
```

Parameter:

head – the head pointer to the previously-described list

Synopsis:

Displays a non-sorted table of the data attributes from the list whose head pointer is passed to it, starting at the head of the list. The display must be in the format illustrated below, in which the first character in each string is aligned and the least significant digits of the occurrence counts are aligned. There are no blank lines between entries. For example:

```
the       107 ea
White      25 ea
White?      4 ea
if         16 ea
etc...
```

Return:

head

`FreeList` syntax:

```
void FreeList(List *head);
```

Parameter:

head – the head pointer to the previously-described list

Synopsis:

Frees all dynamic allocations in the list.

Return:

void

Restrictions:

The `FreeList` function must call no functions or macros other than the standard library *free* function, which it may call as needed.

General Exercise Requirements:

- <u>Never dynamically allocate space for a new node or string until you have first:</u>
    1. read a string from the text file, and then
    2. searched the existing list for it and not found it there.
- Use no dynamic allocations other than those necessary for each node and its string.
- Allocate space for a node and its string separately, **<u>allocating for the node before the string</u>**.
- Allocate **<u>exactly</u>** the right amount of memory needed for each string, including its \0.
- Do not sort the list.
- Do not attempt to read the entire input file into your program at once.
- Test the program on instructor-supplied data file **TestFile1.txt**, which must be placed in the program's "working directory".

Submitting your solution

Send all four source code files to the Assignment Checker with the subject line **C2A6E4_*ID***, where ***ID*** is your 9-character UCSD student ID.

See the course document titled "Preparing and Submitting Your Assignments" for additional exercise formatting, submission, and Assignment Checker requirements.

Hints:

Include each string's null terminator when allocating memory and copying. When deleting a node **<u>always</u>** free its string before freeing the node itself. Freeing the node first results in a memory leak.

Get a Consolidated Assignment Report (optional)

If you would like to receive a consolidated report containing the results of the most recent version of each exercise submitted for this assignment, send an empty email to the assignment checker with the subject line **C2A6_*ID***, where ***ID*** is your 9-character UCSD student ID. Inspect the report carefully since it is what I will be grading. You may resubmit exercises and report requests as many times as you wish before the assignment deadline.