CSE310 Project 1: Linux, C++, Command Line Argument, stdin/stdout/stderr, File I/O, Formatted Output, Memory Management, Linked List, Modular Design

This is your first programming project. It should be written using the standard `C++` programming language, and compiled using the `g++` compiler on a Linux platform. Your project will be graded on Gradescope,which uses the Ubuntu 22.04 version of Linux. If you compile your project on `general.asu.edu` using the compiler commands provided in the sample `Makefile`, you should expect the same behavior of your project on Gradescope. You are advised to implement your project on `general.asu.edu`.

The first thing you need to do is to know how to login to `general.asu.edu` remotely. Please watch the `Programming Tutorial` video to make sure you understand everything taught in that video. This video can be found in Module 1 on Canvas. You may need to activate your service at https://selfsub.asu.edu/.

In this project, I have given you many useful codes. These codes demonstrate the usage of many important functions. **If you want to succeed in future projects, you need to understand everything in the codes provided to you with this project.**

Contents

1 Modular Design

Each module consists of a **header file** and its corresponding **implementation file** (the main module does not need a header file). The header file has extension `.h` and the implementation file has extension `.cpp`. Other than the extensions, the header file and its corresponding implementation file of a module should have the same file name. The header file defines the data structures and prototypes of the functions. The implementation file implements the functions.

For this project, you should have four modules, with the header files named `structs.h`, `util.h`, `list_read.h`, `list_write.h`, and the implementation files `list_read.cpp`, `list_write.cpp`, `main.cpp`, and `util.cpp`. Seven of these eight files are provided to you. You are required to write `list_write.cpp`, following the prototype defined in `list_write.h`. You need to understand everything in these files provided to you and be able to modify them as needed in future projects.

2 Makefile

A `Makefile` is provided to you. You should use the provided Makefile to compile your project. We will grade your project using the provided Makefile.

3 Command Line Arguments

In `main.cpp`, the function `main` takes two parameters `argc` and `argv`, in that order. Here `argc` is the number of commandline arguments, and `argv[0]`, `argv[1]`, ..., `argv[argc-1]` are the commandline arguments. The following example illustrates some basics.

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]){
  printf("argc=%d\n", argc);
  for (int i=0; i<argc; i++){
    printf("The str value of argv[%d] is %s\n", i, argv[i]);
    printf("The int value of argv[%d] is %d\n\n", i, atoi(argv[i]));
  }
  return 0;
}
```

If you are not familiar with command line arguments, you may type the above text in a file named `test.cpp`, and use `g++ test.cpp` to produce the executable file named `a.out`. You may then try the following example executions to learn about the meanings of `argc` and `argv[]`.

```
./a.out
./a.out SCAI @ ASU Spring 2024
```

4 File I/O, Formatted Output

The function `main` shows you how to perform proper file I/O operations and use formatted output. Special attention should be paid to the functions `fopen` and `fclose`. The following example can help you to understand these.

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]){
  FILE *fp1, *fp2;
  int n, v1, v2, v3; float x; double y;

  if (argc < 3){
```

```
    printf("Usage: %s input_file output_file\n", argv[0]);
    exit (1);
  }
  fp1 = fopen(argv[1], "r");
  if (fp1 == NULL) {
    fprintf(stderr, "Error: cannot open file %s\n", argv[1]);
    exit (1);
  }
  fp2 = fopen(argv[2], "w");
  if (fp2 == NULL) {
    fprintf(stderr, "Error: cannot open file %s\n", argv[2]);
    exit (1);
  }
  v1=fscanf(fp1, "%d", &n); v2=fscanf(fp1, "%f", &x); v3=fscanf(fp1, "%lf", &y);
  fprintf(fp2, "v1=%d, v2=%d, v3=%d\n", v1, v2, v3);
  fprintf(fp2, "n=%d, n=%4d\n", n, n);
  fprintf(fp2, "x=%f, x=%8.3f\n", x, x);
  fprintf(fp2, "y=%lf, y=%8.3lf\n", y, y);
  fclose(fp1); fclose(fp2);
  return 0;
}
```

5 Memory Management

The provided files also have examples for dynamic memory allocation and release. Pay attention to `calloc`, `malloc`, and `free`. The following example may be helpful to you.

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]){
  FILE *fp;
  int i, n, *A;

  fp = fopen("INPUT.txt", "r");
  if (fp == NULL) {
```

```
    fprintf(stderr, "Error: cannot open file INPUT.txt\n");
    exit (1);
  }
  fscanf(fp, "%d", &n);
  A = (int *) malloc(n*sizeof(int));
  if (A == NULL) {
    fprintf(stderr, "Error: cannot allocate memory\n");
    exit (1);
  }
  for (i=0; i<n; i++) fscanf(fp, "%d", &A[i]);
  for (i=0; i<n-1; i++) printf("%d ", A[i]); printf("%d\n", A[n-1]);
  free(A);
  return 0;
}
```

6 Data Structures

The following defines the data structures `NODE` and `LIST`

```
#ifndef _structs_h
#define _structs_h 1

typedef struct TAG_NODE{
    double      key;
    TAG_NODE    *next;
}NODE;

typedef struct TAG_LIST{
    NODE        *head;
    NODE        *tail;
}LIST;

#endif
```

You should use these two data structures as defined in the above.

7 nextInstruction

Please carefully study the function `nextInstruction` defined in `util.h` and implemented in `util.cpp`, and its application in `main.cpp`. You will need to fully understand it, and be able to modify it as needed in future projects. While each instruction is a string of characters, some of the instructions take an argument while others do not. Please pay attention to these.

8 Valid Executions

A valid execution of your project has the following form:

`./PJ1 <InputFile> <OutputFile>`

where

- `PJ1` is the executable file of your project,
- `<InputFile>` is the name of the input file,
- `<OutputFile>` is the name of the intended output file.

Your program should check whether the execution is valid. If the execution is not valid, your program should print out an error message to `stderr` and stop. Note that your program should not crash when the execution is not valid.

9 Flow of the Project

9.1 Read in the list from `argv[1]`

Upon a valid execution, your program should open the input file (specified by `argv[1]`) and read in the list. The input file contains the key values of the list. For example, if the content of the input file is the following

```
1
2

3 4


3
```

then the key values of the list will be $1, 2, 3, 4, 3$, in the given order.

9.2 Loop over Instructions

Your program should expect the following instructions from `stdin` and act accordingly:

(a) `Stop`

On reading `Stop`, the program stops.

(b) `Print`

On reading the `Print` instruction, your program should do the following:

(b-i) Write the content of the list to `stdout`, using the

```
%lf\n
```

format for each key value.

(b-ii) Wait for the next instruction from `stdin`.

Note that you should implement the function

```
void listPrint(LIST *)
```

in `list_read.cpp` for this purpose.

(c) `Write`

On reading the `Write` instruction, your program should do the following:

(c-i) Open the output file specified by `argv[2]` for writing.

(c-ii) Write the content of the list to the output file, using the

```
%lf\n
```

format for each key value.

(c-iii) Close the output file.

(c-iv) Wait for the next instruction from `stdin`.

(d) `Max`

On reading the `Max` instruction, your program should do the following:

(d-i) Write to `stdout` the maximum key on the list (if the list is not `NULL` and contains at least one node). Refer to test cases for the output format.

(d-ii) Wait for the next instruction from `stdin`.

This should be implemented in the function

```
double listMax(LIST *)
```

in `list_read.cpp` for this purpose.

(e) Min
On reading the `Min` instruction, your program should do the following:

(e-i) Write to `stdout` the minimum key on the list (if the list is not `NULL` and contains at least one node). Refer to test cases for the output format.

(e-ii) Wait for the next instruction from `stdin`.

This should be implemented in the function

```
double listMin(LIST *)
```

in `list_read.cpp` for this purpose.

(f) Sum
On reading the `Sum` instruction, your program should do the following:

(f-i) Write to `stdout` the sum of the keys on the list (if the list is not `NULL` and contains at least one node). Refer to test cases for the output format.

(f-ii) Wait for the next instruction from `stdin`.

This should be implemented in the function

```
double listSum(LIST *)
```

in `list_read.cpp` for this purpose.

(g) Insert <KEY>
On reading the `Insert` instruction, your program should do the following:

(g-i) Allocate memory for a new node. Set the key field of the new node to the value of `<KEY>` associated with the `Insert` instruction. Insert the new node at head of the list. Write a message to `stdout` (refer to test cases for format).

(g-ii) Wait for the next instruction from `stdin`.

Note that you should implement the function

```
NODE * listInsert(LIST *, double)
```

in `list_write.cpp` for this purpose.

(h) `Append` <KEY>

On reading the `Append` instruction, your program should do the following:

(h-i) Allocate memory for a new node. Set the key field of the new node to the value of `<KEY>` associated with the `Append` instruction. Append the new node at tail of the list. Write a message to `stdout` (refer to test cases for format).

(h-ii) Wait for the next instruction from `stdin`.

Note that you should implement the function

```
NODE * listAppend(LIST *, double)
```

in `list_write.cpp` for this purpose.

(i) `Search` <KEY>

On reading the `Search` instruction, your program should do the following:

(i-i) Search for the first node on the list whose key field is equal to the value of `<KEY>` associated with the `Search` instruction. If such a node exists, return a pointer to this node. If such a node does not exist, return `NULL`. Write a message to `stdout` (refer to test cases for format).

(i-ii) Wait for the next instruction from `stdin`.

This is implemented in the function

```
NODE * listSearch(LIST *, double)
```

in `list_write.cpp` for this purpose.

(j) `Delete` <KEY>

On reading the `Delete` instruction, your program should do the following:

(j-i) Search for the first node on the list whose key field is equal to the value of `<KEY>` associated with the `Delete` instruction. If such a node exists, delete it, and release the memory for the deleted node. Write a message to `stdout` (refer to test cases for format).

(j-ii) Wait for the next instruction from `stdin`.

Note that you should implement the function

```
NODE * listDelete(LIST *, double)
```

in `list_write.cpp` for this purpose.

(k) `Invalid instruction`

On reading an invalid instruction, your program should do the following:

(k-i) Write the following to `stderr`:

```
Invalid instruction: <Instruction>
```

where `<Instruction>` is the instruction returned.

(k-ii) Wait for the next instruction from `stdin`.

10 Format of the Files and Input/Output

In this project, you are required to use `%lf` as the format for `double`. Refer to posted test cases.

11 Submission

You should submit your project to Gradescope via the link on Canvas. Submit only one file, i.e., `list_write.cpp`. You should put your name and ASU ID at the top of `list_write.cpp`, as a comment.

Submissions are always due before `11:59pm` on the deadline date. Do not expect the clock on your machine to be synchronized with the one on Canvas/Gradescope. It is your responsibility to submit your project well before the deadline. **Since you have more than 20 days to work on this project, no extension request (too busy, other business, being sick, Internet issues on submission day, need more accommodations, etc.) is a valid one.**

The instructor and the TAs will offer more help to this project early on, and will not answer emails/questions near the project due date that are clearly in the very early stage of the project. So, please manage your time, and start working on this project immediately. **You are requested to submit a version of your project on Gradescope on each of the Check Point dates specified on Canvas.**

12 Grading

All programs will be compiled and graded on Gradescope. **If your program does not compile and work on Gradescope, you will receive 0 on this project**. If your program works well on `general.asu.edu`, there should not be much problems. The maximum possible points for this project is 100. The following shows how you can have points deducted.

1. **Non-working program**: If your program does not compile or does not execute on Gradescope, you will receive a 0 on this project. Do not claim "my program works perfectly on my PC, but I do not know how to use Gradescope."

2. **Posted test cases**: For each of the 20 posted test cases that your program fails, 4 points will be marked off.

3. **UN-posted test cases**: For each of the 5 un-posted test cases that your program fails, 4 points will be marked off.

13 Examples

In this section, I provide some examples. All examples assume that the input file is named `I-file` has the following content:

```
1
2

3 4

3
```

13.1 Example 1

Execution line is the following:

```
./PJ1
```

This is an invalid execution. The program writes the following error message to `stderr` and terminates.

```
Usage: PJ1 <ifile> <ofile>
```

13.2 Example 2

Execution line is the following:

```
./PJ1 I-file My-O-file
```

The instructions from `stdin` are as follows:

```
Print
Max
Write
Stop
```

This is a valid execution. The program writes the following to `stdout`:

```
1.000000
2.000000
3.000000
4.000000
3.000000
Max=4.000000
```

writes the following to the file `My-O-file`:

```
1.000000
2.000000
3.000000
4.000000
3.000000
```

and terminates.

13.3 Example 3

The instructions from `stdin` are as follows:

```
Print
Append 1
Append 2
Insert 1
Print
Delete 4
Print
Write
Stop
```

stored in a file named `Instructions`. The execution line is the following:

```
./PJ1 I-file My-O-file < Instructions
```

This is a valid execution. The program writes the following to `stdout` and terminates.

```
1.000000
2.000000
3.000000
4.000000
3.000000
Node with key 1.000000 appended
Node with key 2.000000 appended
Node with key 1.000000 inserted
1.000000
1.000000
2.000000
3.000000
4.000000
3.000000
1.000000
2.000000
Query 4.000000 FOUND in list
Node with key 4.000000 deleted
```

```
1.000000
1.000000
2.000000
3.000000
3.000000
1.000000
2.000000
```

writes the following to `My-O-file` and terminate.

```
1.000000
1.000000
2.000000
3.000000
3.000000
1.000000
2.000000
```

14 Developing Your Project on general.asu.edu

You are strongly recommended to develop your project on `general.asu.edu`. This section provides some basic steps to help you to do so. It is assume that you already know how to transfer files between your local computer and `general.asu.edu` and know how to login to `general.asu.edu` remotely, and know some basics of `Linux` and the `vi` editor. You can learn the above by watching the `Programming Tutorial` video in Module 1.

14.1 Starting with the files in PJ01-Help.zip

The `Linux` command

```
unzip PJ01-Help.zip
```

will create a directory named `PJ01-Help`. Go to this directory with the `cd` command.

You can use the `Linux` command `ls` to see the files and directories in this directory. You can use the `Linux` command

```
make
```

to compile the provided codes, and produce the executable `PJ1`.

You need to make changes to the file `list_write.cpp` according to the specifications of this project.

14.2 Executing Interactively

You are recommended to execute your program interactively. Copy the files `I-file`, `Instructions`, `O-file`, and `Output` from one of the test cases (e.g. `test01`) to the current directory. In this document, we will assume that we are using `test01`. The content of `Instructions` for this test case is

```
Print
Write
Stop
```

To execute your program interactively, you use the following `Linux` command:

```
./PJ1 I-file My-O-file
```

then enter the three instructions in the given order. The following is what I saw on my screen after the execution.

```
gxue1@general[1299]% ./PJ1 I-file My-O-file
Print
1.000000
2.000000
3.000000
4.000000
3.000000
Write
Stop
gxue1@general[1300]%
```

I can use `cat` to see what is in the file `My-O-file`:

```
gxue1@general[1300]% cat My-O-file
1.000000
2.000000
3.000000
```

```
4.000000
3.000000
gxue1@general[1301]%
```

14.3 Executing in Batch Mode

Once you are familiar with the execution, you can try running it in batch batch mode, and use `redirection`. To do so, you can use the following `Linux` command:

```
./PJ1 I-file My-O-file < Instructions > My-Output
```

There are two `redirection`s in the above. The first `redirection` takes the instructions in the file `Instructions` as if there are entered from `stdin`. The second `redirection` directs `stdout` to the file named `My-Output`. You can use `cat` or `vi` to see the contents of `My-O-file` and `My-Output`.

14.4 Using the Text Cases

The shell script `My-Execution` can be used to check your program against a given test case. Assume that you have copied the files in `test01` to the current directory `PJ01-Help`, you can use the following command to see whether your program has the expected behavior for the given test case:

```
./My-Execution
```

The following is my execution using the provided `list_write.cpp`:

```
gxue1@general[1301]% ./My-Execution
./PJ1 I-file My-O-file < Instructions > My-Output
========
diff O-file My-O-file
========
diff Output My-Output
5a6,19
> BGN listAppend
>       You need to rewrite this function
> END listAppend
> BGN listInsert
>       You need to rewrite this function
> END listInsert
```

```
> BGN listDelete
>       You need to rewrite this function
> END listDelete
> 1.000000
> 2.000000
> 3.000000
> 4.000000
> 3.000000
========
gxue1@general[1302]%
```

We can see that the files `My-O-file` and `O-file` are identical. However, the files `My-Output` and `Output` are different. Therefore, the current `list_write.cpp` will NOT pass the chosen test case.

After I revised `list_write.cpp` according the the specifications of this project, I will get the following result:

```
gxue1@general[1302]% ./My-Execution
./PJ1 I-file My-O-file < Instructions > My-Output
========
diff O-file My-O-file
========
diff Output My-Output
========
gxue1@general[1303]%
```

This shows that (1) `O-file` and `My-O-file` are identical, and (2) `Output` and `My-Output` are identical. Hence the revised program will pass the current test case.

You can use the above procedure to check your program against all 20 posted test cases.