Assignment 8

C/C++ Programming I

C1A8 General Information

Where Does a Program Look for Files When Attempting to Open Them?
Where Does a Program Create New Files?
Where Should You Put Instructor-Supplied Data Files?

What is a "Working Directory"?
A program's "Working Directory" is the directory it uses for any files it opens or creates if their names are specified without a path, and you must place any instructor-supplied data file(s) (.txt or .bin extensions) your program needs in that directory. Its default location differs between IDEs and operating systems and it's important to know where it is and how to change it. For further information please refer to the **Determining/Changing the "Working Directory"** topic in the version of the course document titled "Using the Compiler's IDE..." that is applicable to the IDE you are using.

Opening Files – Testing for Failure/Success

Always check the success/failure status of opening a file before using it <u>or opening another file</u>.

Supplying Information to a Program via its "Command Line"

It is often more appropriate to supply information to a program via "command line arguments" than by user prompts. Such arguments can be provided regardless of how a program is being run, whether it be from within an IDE, a system command window, a GUI icon, or a batch file. For this course I strongly recommend using an IDE for running all programs.

If you are not familiar with using command line arguments first review note 8.3 for information on how to process them within any program, then review the appropriate version of the course document titled "Using the Compiler's IDE...", which illustrates implementing an arbitrary command line in several ways <u>including implementing command arguments containing spaces</u>.
It is important to note that command line redirection information (note 4.2), if any, is only visible to the operating system and will not be among the command line arguments available to the program being run.

Get a Consolidated Assignment 8 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 **C1A8_*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.

C1A8E0 *(6 points total - 1 point per question – No program required)*

Assume language standards compliance and any necessary support code unless stated otherwise. Testing erroneous or implementation dependent code by running it can be misleading. These are not trick questions and each has only one correct answer. Major applicable course book notes are listed.

1. *stdin*, *stdout*, and *stderr*:
   (Note 10.1)
   A. may cause a compiler error in C if
      using namespace std;
      is not declared first.
   B. are the names of three standard files.
   C. are all of data type *FILE*
   D. may not all be of data type *FILE **
   E. none of the above

2. What is the most serious problem?
```
struct Savings {
    double principal, interest;
 public:
    double total()
        {return(principal + interest);}
};
Savings savings;
cout << "Total account value is "
   << savings.total() << "\n"
   "Interest = " << savings.interest;
```
   (Note 5.11)
   A. Garbage values will be output.
   B. *total* isn't an **inline** function.
   C. *total* accesses private data members.
   D. *total* isn't called with a *Savings* object.
   E. *savings.interest* is illegal.

3. If *ofstream oFile("myFile");* succeeds, how many bytes does *oFile << "/n/n\n"* write into the file?
   (Note 10.2)
   A. 0
   B. 3 or 4
   C. 4 or 5
   D. 5 or 6
   E. 3 or 6

4. Noting that *scanf* calls *ungetc* if necessary, what gets printed if *xyz* is type **int** and *65FFz* is entered from the keyboard?
```
scanf("%x", &xyz);
ungetc('A', stdin);
scanf("%x", &xyz);
printf("%x", xyz);
```
   (Note 10.5)
   A. *65FF*
   B. *a*
   C. FF
   D. nothing is printed
   E. Implementations may differ on this.

5. What is the most serious problem if the intent is to output only the characters in the file represented by *inFile*?
```
int ch;
do
    cout << (char)(ch = inFile.get());
while (ch != EOF);
```
   (Notes 4.3A & 4.3B)
   A. EOF must be tested using *inFile.eof()*.
   B. *get* is not capable of returning *EOF.*
   C. **while** (ch != EOF) may not detect *EOF* at all or false *EOF* indications may occur.
   D. *EOF* is checked in the wrong place.
   E. *ch* is type **int** but *get* returns type **char**.

6. What is the most serious problem if the intent is to write to the file?
```
ofstream oFile("myname");
oFile.write("ABCD", 4);
```
   (Notes 7.4, 10.4A, & 10.4B)
   A. The open's success/failure is not tested.
   B. *oFile.write("ABCD", 4);* is not legal syntax.
   C. The write operation will always fail.
   D. More than 4 characters might be written.
   E. The write should be *oFile << "ABCD";*

Submitting your solution

Using the format below place your answers in a plain text file named **C1A8E0_Quiz.txt** and send it to the assignment checker with the subject line **C1A8E0_*ID***, where ***ID*** is your 9-character UCSD student ID.

-- Place an appropriate "Title Block" here --
1. A
2. C
etc.

See the course document titled "How to Prepare and Submit Assignments" for additional exercise formatting, submission, and assignment checker requirements.

C1A8E1 *(4 points – C++ Program)*

Exclude any existing source code files that may already be in your IDE project and add three new ones, naming them **C1A8E1_SavingsAccount.h**, **C1A8E1_SavingsAccount.cpp**, and **C1A8E1_main.cpp**. Do not use #include to include either of the two **.cpp** files in each other or in any other file. However, you may use it to include any appropriate header file(s) you need. You must include **C1A8E1_SavingsAccount.h** in any file that needs its contents.

This exercise is not an example of rigorous banking practices, but is only to familiarize you with simple C++ **class** members. Assume the **class** below represents just the data members of a savings account:

```
class SavingsAccount       // Do not change the order or access level of this class's data members
{
    int type;
    string ownerName;      // You must handle the case where the name contains whitespace
    long IDnbr;
    double balance, closurePenaltyPercent;
};
```

File **C1A8E1_SavingsAccount.h** must contain the following:

1. the **class** SavingsAccount definition from above with the following added to its contents:
   a. the prototype only for member function GetInitialValues
   b. the prototype only for member function DisplayValues
   c. the entire definition of member function CalculatePenalty
2. the entire definition of function DisplayValues – placed outside **class** SavingsAccount

File **C1A8E1_SavingsAccount.cpp** must contain:

1. the entire definition of member function GetInitialValues

File **C1A8E1_main.cpp** must contain function main, which must:

1. declare a variable of type SavingsAccount (do not use dynamic allocation).
2. call functions GetInitialValues, DisplayValues, and CalculatePenalty in that order. You may respond to any prompts with whatever values you deem sufficient to test your program.
3. display the value returned from CalculatePenalty using the following format, where the question mark represents the actual value:
   Account closure penalty: ?

GetInitialValues: a non-inline member function, which must:

1. have a **void** parameter list and return **void**;
2. prompt the user for 5 values and initialize the 5 **class** data members to those values. The user may enter any values that can be represented in the members' data types. Prompt for each value separately and in the same order as the members appear in the class.

DisplayValues: a **const** inline member function, which must:

1. have a **void** parameter list and return **void**
2. not prompt for anything
3. use one cout to display the values of all data members in the same order as they appear in the **class**, using the following format, where the question marks represent the actual values:

```
Account type: ?
Owner name: ?
ID number: ?
Account balance: ?
Account closure penalty percent: ?
```

CalculatePenalty: a **const** inline member function, which must:

1. have a **void** parameter list and return an appropriate type

2. calculate the account closure penalty and return it to the caller. It will be the percent specified by `closurePenaltyPercent` of the account balance specified by `balance`. Do not use division.
3. <u>not</u> prompt or display anything

Manually re-run your program several times, testing with different values.

Submitting your solution

Send your three source code files to the assignment checker with the subject line **C1A8E1_*ID***, where ***ID*** is your 9-character UCSD student ID.

See the course document titled "How to Prepare and Submit Assignments" for additional exercise formatting, submission, and assignment checker requirements.

Hints:

1. See note 4.1. Some C/C++ functions that read text input leave the terminating newline character in the input buffer. If it is still in the buffer when the C++ `getline` function is called, `getline` reads only up through that character. Consider using `cin >> ws;` before the code that inputs the name of the account owner to eliminate any unwanted whitespace from the input buffer. `ws` is not a variable that you declare but is what is known as a "manipulator". It's in the `std` namespace and is available when you include standard C++ header file **iostream**.

2. 1.3% of 0.2 is not 0.26

3. Be sure to use "include guards" (note D.2) in header file **C1A8E1_SavingsAccount.h**. Use `#include` to include this file in files **C1A8E1_SavingsAccount.cpp** and **C1A8E1_main.cpp**.

C1A8E2 *(4 points – C Program)*

Exclude any existing source code files that may already be in your IDE project and add a new one, naming it **C1A8E2_main.c**. Write program in that file to display the contents of an arbitrary text file one group of lines at a time, including empty lines (if any).

The program must get two space-separated arguments from the command line (not from a user prompt or from values hard coded in the code itself). The first argument specifies the name of the desired text file while the second specifies the number of lines in a group. For example, command line arguments of

```
   TestFile1.txt  5
```

would tell the program to display the contents of a file named **TestFile1.txt** in 5-line groups.

When the program starts it must automatically display the first group of lines. At the end of every complete group the program must wait for a keyboard entry. If the entry is a newline (the `Enter` key) the next group of lines must be displayed. If the entry is any key other than `Enter` followed by `Enter` the program must exit immediately without displaying any more lines. If the last group contains fewer than the specified number of lines the program must display any that are present and immediately exit.

- Your program must not prompt the user in any way at any time.

- Before using any command line arguments the program must verify that exactly the correct number of them exists. If not, an error message must be output to `stderr` and the program terminated with an error exit code. The error message must indicate the reason for termination.

- If the specified file fails to open an error message must be output to `stderr` and the program terminated with an error exit code. The error message must indicate the reason for termination and mention the name of the failing file.

Manually re-run your program several times, testing with at least instructor-supplied data file **TestFile1.txt**, which must be placed in the program's "working directory". Try various line count values.

Submitting your solution

Send your source code file to the assignment checker with the subject line **C1A8E2_*ID***, where ***ID*** is your 9-character UCSD student ID.

See the course document titled "How to Prepare and Submit Assignments" for additional exercise formatting, submission, and assignment checker requirements.

Hints:

1. In C do you know how to convert a string that represents a numeric value into the numeric value it represents? If not, consider `sscanf`, `atoi`, `atol`, `atof`, `strtol`, `strtoul`, or `strtod`.

2. The value of the EOF macro cannot be reliably stored or detected using any form of data types **char** or **short**; use type **int** instead (Notes 4.3A & 4.3B).

3. Since the "end of file condition" only occurs when a read (or write) is attempted, testing for it before that attempt provides no useful information about what the outcome will be. Only end of file checks done after reading/writing and before any values resulting from it are actually used are valid for avoiding data problems. (Note 4.3B).

4. Not testing files for a successful open is unacceptable (Notes 10.3 & 10.4B).

C1A8E3 *(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 **C1A8E3_main.cpp**. Write a program in that file to perform a "Search and Replace All" operation. This operation consists of performing a case-sensitive search for all occurrences of an arbitrary sequence of characters within a file and substituting another arbitrary sequence in place of them.

One of the biggest problems some students have with this exercise occurs simply because they don't read the command line information in the appropriate version of the course document titled "Using the Compiler's IDE...".

Your program:
1. may assume, for simplicity:
    a. it will only be used with text files;
    b. the input and output file names will be different;
    c. no character sequence will be split across multiple lines;
2. must support multiple occurrences of the "search" character sequence on a single line;
3. must support cases where the length of the "search" character sequence is different from the length of the "replacement" character sequence;
4. must get the following information from the command line in the order listed (spaces must be allowed in all items):
    a. the name of the file to search (the input file);
    b. the name of the file to store the results in (the output file);
    c. the sequence of characters to search for;
    d. the sequence of characters to use as replacements.
5. must verify that exactly the correct number of command arguments exist before accessing them and exit with an error message and error code if they don't;
6. must not use data types `list`, `queue`, `stack`, `string`, or `vector`;
7. must not call the `strlen` function more than once for the same string, including inside a loop;
8. must not copy part/all of any string to any destination other than the output file itself;
9. must not declare more than one array or use dynamic memory allocation;
10. must not test for an empty input file or an input file that does not contain the search sequence.

Although not required, I recommend that you implement the algorithm provided in the hints.

Manually re-run your program several times, testing at least the following 2 cases with instructor-supplied data file **TestFile1.txt**, which must be placed in the program's "working directory". Choose any name you wish for the output file as long as it's different from the input file name:

Test#	Input File	Output File	Search For	Replace Each With
1	*TestFile1.txt*	**your choice**	*the*	*John Galt?*
2	*TestFile1.txt*	**your choice**	*string literal*	*TESTING*

For example (Test #1), if the input file contained,
These are the answers to neither of their questions!
the output file would contain,
These are John Galt? answers to neiJohn Galt?r of John Galt?ir questions!
Note that there were three replacements on one line and that "the" was part of another word in two of those cases.

Submitting your solution

Send your source code file to the assignment checker with the subject line **C1A8E3_*ID***, where ***ID*** is your 9-character UCSD student ID.

See the course document titled "How to Prepare and Submit Assignments" for additional exercise formatting, submission, and assignment checker requirements.

Hints:

1. The value of EOF cannot be correctly stored or detected using data type **char** or **unsigned char**; use type **int** instead (Notes 4.3A & 4.3B). Since EOF only occurs when a read (or write) is attempted, testing for it before such an attempt is meaningless and inappropriate. Only test after such an attempt before the value obtained from reading is actually used (Note 4.3B).

2. Not testing files for a successful opening is bad programming (Notes 10.3 & 10.4B).

3. It is not necessary to determine the length of the replacement string.

You may use any algorithm you wish to complete this exercise as long as none of the requirements/restrictions are violated. The following describes the algorithm I used and recommend. Drawing a diagram of memory as the algorithm progresses always helps:

General Algorithm Description:

```
For each line in the input file
{
    Set a pointer to first character in line;
    For each "search" sequence found in line using the strstr library function
    {
        Copy characters into the output file from the pointer to where the "search" sequence starts;
        Write the replacement string into the output file;
        Move the pointer to the next character in the input line after the "search" sequence ends;
    }
    Copy remainder of line into the output file;
}
```

Algorithm Step-by-Step Implementation Details:

1. Do all the standard things, including declaring needed variables and opening the input and output files.
2. Implement a loop statement that contains the following 3 statements (***a***, ***b***, and ***c***) in order:
   a. a statement that gets the next line from the input file and stores it into a character buffer. (The newline character must be discarded.) If the EOF condition occurs, terminate the loop. Otherwise, proceed to step ***b*** below.
   b. a "for" statement that does everything in steps ***i*** and ***ii*** below (in order):
      i. does the following in its "control" section. The "control" section is the portion of the "for" statement that is in parentheses just after the keyword `for`:
         1) The "initial expression" (Note 3.5) initializes a character pointer (I'll call it `cp1`) to point to the beginning of the character buffer you are reading each line into.
         2) The "controlling expression" (Note 3.5) assigns the return value of the `strstr` function to a different character pointer (I'll call it `cp2`). The first argument of `strstr` will be `cp1` and the second argument will be a pointer to the first character of the string you are looking for in the file.
         3) The "loop expression" (Note 3.5) is empty.
      ii. does the following in the loop "body":
         1) Uses the `write` function to write to the output file. The first argument of `write` will be `cp1` and the second will be `cp2-cp1`.
         2) Uses the << operator to write the replacement string to the output file.
         3) Updates `cp1` by assigning to it the sum of `cp2` and the length of the string you are searching for in the file.
   c. a statement that writes the string in `cp1` and a newline character to the output file.