



University of Glasgow | School of Computing Science

Assessed Coursework

Course Name	Systems Programming			
Coursework Number	1a: Chat Server Member Monitoring			
Deadline	Tim e:	16:30	Dat e:	30th Oct 2025
% Contribution to final course mark	16%			
Solo or Group	Solo	<input type="checkbox"/>	Group	
Anticipated Hours	20			
Submission Instructions	Described in section 5 below.			
Please Note: This Coursework cannot be Re-Done				

Code of Assessment Rules for Coursework Submission

Deadlines for the submission of coursework which is to be formally assessed will be published in course documentation, and work which is submitted later than the deadline will be subject to penalty, as set out below. The primary grade and secondary band awarded for coursework submitted after the published deadline will be calculated as follows:

- (i) in respect of work submitted not more than five working days after the deadline
 - a. the work will be assessed in the usual way;
 - b. the primary grade and secondary band so determined will then be reduced by two secondary bands for each working day (or part of a working day) the work was submitted late.
- (ii) work submitted more than five working days after the deadline will be awarded Grade H.

Penalties for late submission of coursework will not be imposed if a good cause is established for late submission. You should submit documents supporting a good cause via MyCampus.

Penalty for non-adherence to Submission Instructions is 2 bands

You must complete an “Own Work” form via
<https://webapps.dcs.gla.ac.uk/ETHICS> for all coursework
UNLESS submitted via Moodle

Chat Server Member Monitoring

1 Requirements

Modern chat servers need to efficiently track the members currently active in a channel. For the purposes of this assessment, you will build a command-line tool that processes a log of server events.

The log of events is a text file where each line contains a timestamp (in dd/mm/yyyy hh:mm format) a command, a username, and sometimes additional data. The valid commands are JOIN, LEAVE, and STATUS. The STATUS command is followed by a new status (ONLINE, AWAY, OFFLINE). These commands/status indicators are case-sensitive and must appear in uppercase.

The log might look like this:

```
05/11/2024 10:00 JOIN Alex
12/12/2024 14:30 JOIN Zion
13/12/2024 09:00 JOIN casey
20/12/2024 10:00 JOIN AlexTaylor
01/01/2025 11:00 LEAVE Zion
02/01/2025 12:00 JOIN Jordan
03/01/2025 15:00 STATUS Alex AWAY
04/01/2025 18:00 STATUS casey OFFLINE
05/01/2025 09:00 JOIN Alex
```

The output of the program should be a list of all **current** members, **sorted alphabetically by username**, along with their status, and the timestamp of the latest status update. Usernames are **case-sensitive**. Therefore, Harper, harper, and HARPER are three distinct users.

Given the above log, the program output for the `members` command should be:

```
Alex (ONLINE)
AlexTaylor (ONLINE)
Jordan (ONLINE)
casey (OFFLINE: last seen on 04/01/2025)
```

The key requirement of this assessment is to implement a data structure that maintains sorted alphabetical order internally. Therefore, your program's output should already be correctly sorted alphabetically when printed, as a direct result of traversing your data structure.

2 Specification

This section details the rules for processing the log file and the expected behavior of the underlying data structure for each command.

General Rules:

- **Usernames:** Usernames are handled **case-sensitively**. The program must treat **Alex** and **alex** as two distinct users.
- **Data Structure:** The underlying data structure for the member list must be a **Skip List**.
- **Sorting:** The primary data structure must maintain the list of members sorted alphabetically at all times. The output of the **members** command must reflect this internal sorted order.
- **Membership Logic:** The state of the member list is determined by the following command rules:
 - A **JOIN** command adds a user. Their status is set to **ONLINE**. If an existing user rejoins, their status is simply updated to **ONLINE** and their timestamp is updated to reflect the most recent join time.
 - A **LEAVE** command **permanently removes a user** from the list of members.
 - A **STATUS** command updates the status and timestamp of an existing user. If the user does not exist, the command is ignored.

Output Formatting:

The final report should print one line for each user. The format depends on the user's status and the time of their last activity relative to a preset date (10/10/2025 12:00).

- For **ONLINE** users, the format is: **Username (ONLINE)**
- For **AWAY** and **OFFLINE** users, the format is: **Username (STATUS: <last seen message>)** where **<last seen message>** is determined by comparing the user's last activity date to the current time:
 - If the last activity was today, the message is: "**last seen at hh:mm**"
 - If the last activity was within the same month, the message is: "**last seen X day(s) ago**"

SP Assessed Exercise 1a

- o Otherwise, the message is: "last seen on dd/mm/yyyy"

3 Design

You are provided with the source file for the `main()` driver in `server-monitor.c`, and header files for two abstract data types (ADTs): `date.h` and `memberlist.h`.

3.1 `date.h` (*comments are part of the required specification*)

```
#ifndef _DATE_H_INCLUDED_
#define _DATE_H_INCLUDED_

#include <time.h>

// Opaque structure for a date/timestamp.
typedef struct Date Date;

/*
 * date_create creates a Date structure from `datestr`.
 * `datestr` is expected to be of the form "dd/mm/yyyy hh:mm".
 * Returns a pointer to a new Date structure if successful, NULL otherwise.
 * The caller is responsible for destroying the returned date.
 */
Date *date_create(const char *datestr);

/*
 * date_valid checks if a Date structure is valid.
 * Returns 1 if valid, 0 otherwise.
 *
 * This function is used internally by functions that parse or copy
 * dates, such as date_create(), to verify that all fields
 * (day, month, year, hour, minute) fall within valid ranges before storing
 * or
 * using the date.
 */
int date_valid(const Date *d);

/*
 * date_duplicate creates a new copy of a Date structure.
 * Returns a pointer to the new Date structure, or NULL on failure.
 * The caller is responsible for destroying the returned date.
 */
Date *date_duplicate(const Date *d);

/*
 * date_format_last_seen returns a newly allocated string describing how
 * long
 * ago a user was last seen (last), relative to the current date (now).
 * - If the date is today, returns "last seen at hh:mm".
 * - If within the same month, returns "last seen X days ago".
 * - Otherwise, returns "last seen on dd/mm/yyyy".
 * The caller is responsible for freeing the returned string.
 * Returns NULL on memory allocation failure.
 */
char *date_format_last_seen(const Date *last, const Date *now);

/*
 * date_now creates a Date structure for the current system date and time.
 * This function should be implemented using <time.h> facilities such as
 * time(), localtime(), and struct tm.
```

SP Assessed Exercise 1a

```
* Returns a pointer to the new Date structure, or NULL on failure.  
* The caller is responsible for destroying the returned date.  
*  
* This function is used internally by date_format_last_seen()  
* to compare "last seen" times to the current system date and time.  
*/  
Date *date_now(void);  
  
/*  
 * date_destroy frees all memory associated with a Date structure.  
 */  
void date_destroy(Date *d);  
  
#endif /* _DATE_H_INCLUDED_ */
```

The `struct date`, and the corresponding `typedef Date`, define an opaque data structure for a date.

The constructor for this ADT is `date_create()`, which converts a `datestring` in the format “dd/mm/yyyy hh:mm” to a `Date` structure. You will have to use `malloc()` to allocate this `Date` structure to return to the user. Note: we advise against using `strtok()` as it can lead to undefined behaviour; instead, use `strdup()` or whatever else that works reliably.

`date_valid()` is used by the `date_create()` function to check if a date is valid. You do not need to check everything exhaustively, just make sure that all fields fall within valid ranges.

`date_duplicate()` is known as a copy constructor; it duplicates the `Date` argument on the heap (using `malloc()`) and returns it to the user.

`date_format_last_seen()` compares the date a user last changed their status (`last`) to the current date (`now`) and formats an output string. If the date is today, it returns "last seen at hh:mm". If within this month, it returns "last seen X days ago". Otherwise, it returns "last seen on dd/mm/yyyy".

`date_now()` creates a `Date` data structure for the current system date and time. You should implement this function using functions available in `time.h`, e.g. `time()`, `localtime()`.

`date_destroy()` returns the heap storage associated with the `Date` structure.

3.2 memberlist.h (*comments are part of the required specification*)

```

#ifndef _MEMBERLIST_H_INCLUDED_
#define _MEMBERLIST_H_INCLUDED_

#include "date.h"
#include <stdint.h>
#include <stdlib.h>

// Skip list parameters
/*
 * MAX_LEVEL sets the maximum height of the skip list towers.
 * Each node gets a random level between 1 and MAX_LEVEL.
 * A typical rule of thumb is MAX_LEVEL ≈ log2(N), where N is
 * the expected number of elements. Here we fix it to 16, which
 * is more than enough for lists up to about 65,000 members.
 */
#define MAX_LEVEL 32

/*
 * P is the probability used when assigning levels.
 * Each level is chosen with probability P of going "up" again.
 * With P = 0.5, about 1/2 of the nodes are at level ≥1,
 * 1/4 at level ≥2, 1/8 at level ≥3, and so on.
 */
#define P 0.5

// User datatypes

typedef enum { ONLINE, AWAY, OFFLINE } UserStatus;

typedef struct {
    char *username;
    UserStatus status;
    Date *last_activity_date;
} User;

// Opaque datatypes
/*
 * Implementation requirements (read carefully):
 *
 * You must implement the MemberList ADT as a skip list.
 * - Each MemberList contains:
 *     // current maximum level
 *     // one head pointer per level
 * - Each MemberNode contains:
 *     // dynamically allocated user data
 *     // array of forward pointers
 *
 * Each level of the skip list must be NULL-terminated.
 *
 * These structures must be defined in memberlist.c.
 * They are opaque to other modules (e.g. server-monitor.c).
 */
typedef struct memberlist MemberList;
typedef struct membernode MemberNode;
typedef struct memberiterator MemberIterator;

```

SP Assessed Exercise 1a

```
/*
 * select_level() randomly determines the height (level) of a new node's
tower.
 * This randomness is what gives the skip list its expected O(log n)
behavior.
 * Returns the new node's level.
 *
 * It is called internally by memberlist_add() when inserting new users.
 */
static inline int select_level() {
    int level = 0;
    while (((double)rand() / RAND_MAX) < P && level < MAX_LEVEL - 1) {
        level++;
    }
    return level;
}

// Construction/Destruction

/*
 * memberlist_create creates an empty MemberList.
 */
MemberList *memberlist_create();

/*
 * memberlist_destroy destroys the list structure, including all nodes,
 * usernames, and user data (including the Date object for each user).
 */
void memberlist_destroy(MemberList *mlist);

// CRUD operations

/*
 * Called by server-monitor.c when processing JOIN commands.
 *
 * memberlist_add adds a new user to the list or updates an existing one
to
 * ONLINE.
 * - If the user does not exist, a new entry is created with status
ONLINE.
 * - If the user exists, their status is updated to ONLINE and their
timestamp
 * is updated. The function makes its own internal copies of the username
and
 * date.
 * Returns 1 if successful, 0 on failure.
 *
 * Implementation notes:
 * - Use select_level() to determine the height of each new node.
 * - Ensure that all pointers (especially next[]) are correctly
initialized.
 * - Free any temporary memory on error to avoid leaks.
 */
int memberlist_add(MemberList *mlist, const char *username, const Date
*d);

/*
 * Called by server-monitor.c when processing LEAVE commands.
 *
 * memberlist_remove permanently removes a user from the list.
```

SP Assessed Exercise 1a

```
* This function deallocates the node, the username string, and all user
data.
* Returns 1 if successful, 0 if the user was not found.
*/
int memberlist_remove(MemberList *mlist, const char *username);

/*
* Called by server-monitor.c when processing STATUS commands.
*
* memberlist_update_status finds a user and updates their status and
timestamp.
* Returns 1 if successful, 0 if the user was not found.
*/
int memberlist_update_status(MemberList *mlist, const char *username,
                             UserStatus status, const Date *d);

// Iteration
/*
* Iterators provide a sequential view of the skip list for output
purposes.
* The iterator traverses level 0 of the skip list only.
* Used by server-monitor.c when printing the "members" list.
*/

/*
* memberlist_iter_create creates an iterator to traverse the list.
*/
MemberIterator *memberlist_iter_create(MemberList *mlist);

/*
* memberlist_iter_next returns the next node in the sequence.
*/
MemberNode *memberlist_iter_next(MemberIterator *iter);

/*
* memberlist_iter_destroy destroys the iterator.
*/
void memberlist_iter_destroy(MemberIterator *iter);

// Accessors

/*
* membernode_username returns the username from a node.
*/
const char *membernode_username(MemberNode *node);

/*
* membernode_status returns the status of the user from a node.
*/
UserStatus *membernode_status(MemberNode *node);

/*
* membernode_last_activity_date returns the user's last activity date
from a
* node.
*/
Date *membernode_last_activity_date(MemberNode *node);

#endif
```

SP Assessed Exercise 1a

`MemberList`, `MemberNode`, and `MemberIterator` are now opaque data structures. You are responsible for defining the contents of `struct memberlist`, `struct membernode`, `struct memberiterator` yourself within your `memberlist.c` implementation.

`memberlist_create()` creates an empty `MemberList` structure (your skip list) and returns it to the user.

`memberlist_destroy()` deallocates all heap storage associated with the `MemberList`, including all nodes, the usernames, and the `User` data and `Date` objects they contain.

`memberlist_add()` handles a `JOIN` command. If the user does not exist, a new node is created. If the user already exists, their status is updated to `ONLINE` and their `last_activity_date` is updated. The function should make its own internal, heap-allocated copy of the username string and the `Date` object.

`memberlist_remove()` handles a `LEAVE` command by permanently removing the user and all associated data from the list.

`memberlist_update_status()` handles a `STATUS` command by finding the specified user and updating their status and `last_activity_date`.

`memberlist_iter_create()`, `memberlist_iter_next()`, and `memberlist_iter_destroy()` provide an iterator pattern to safely traverse the member list.

`membernode_username()`, `membernode_status()`, and `membernode_last_activity_date()` are the accessor functions required to get data from a `MemberNode`. This allows the driver to function without knowing the internal layout of your `User` struct.

3.3 server-monitor.c

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

#include "date.h"
#include "memberlist.h"

#define USAGE "usage: %s [log-file] ...\\n"
#define LINE_BUFFER_SIZE 1024

// Helper function to convert a UserStatus enum to a string for printing
static const char *status_to_string(UserStatus status)
{
    switch (status) {
    case ONLINE:
        return "ONLINE";
    case AWAY:
        return "AWAY";
    case OFFLINE:
        return "OFFLINE";
    default:
        return "UNKNOWN";
    }
}

// Helper function to convert a string command to a UserStatus enum
static int string_to_status(const char *str, UserStatus *status)
{
    if (strcmp(str, "ONLINE") == 0) {
        *status = ONLINE;
        return 1;
    }
    if (strcmp(str, "AWAY") == 0) {
        *status = AWAY;
        return 1;
    }
    if (strcmp(str, "OFFLINE") == 0) {
        *status = OFFLINE;
        return 1;
    }
    return 0; // Invalid status string
}

// Processes a single log file, adding/updating/removing members from the list
static void process_file(FILE *fd, MemberList *mlist)
{
    char buffer[LINE_BUFFER_SIZE];

    // Each line format: "dd/mm/yyyy hh:mm COMMAND username [status]"
    while (fgets(buffer, sizeof(buffer), fd) != NULL) {
        if (strlen(buffer) <
            18) { // Basic sanity check for timestamp + space
            continue;
        }

        // Isolate the timestamp string
        char datestr[17];
        strncpy(datestr, buffer, 16);
        datestr[16] = '\\0';
    }
}
```

SP Assessed Exercise 1a

```
Date *d = date_create(datestr);
if (!d) {
    fprintf(stderr,
            "Warning: Skipping malformed date line: %s",
            buffer);
    continue;
}

// Use sscanf for the rest of the line
char command[32], username[256], extra[256];
int items = sscanf(buffer + 17, "%31s %255s %255s", command,
                    username, extra);

if (items < 2) {
    date_destroy(d);
    continue; // Not enough parts to be a valid command
}

int ret = 0;
if (strcmp(command, "JOIN") == 0) {
    ret = memberlist_add(mlist, username, d);
    if (!ret) {
        printf("Error adding user\n");
    }
} else if (strcmp(command, "LEAVE") == 0) {
    ret = memberlist_remove(mlist, username);
} else if (strcmp(command, "STATUS") == 0 && items == 3) {
    UserStatus new_status;
    if (string_to_status(extra, &new_status)) {
        memberlist_update_status(mlist, username,
                                new_status, d);
    }
}

// The memberlist makes its own copies, so we must destroy the
// temporary one
date_destroy(d);
}

int main(int argc, char *argv[])
{
    if (argc < 2) {
        fprintf(stderr, USAGE, argv[0]);
        return 1;
    }

    // Create a member list
    MemberList *mlist = memberlist_create();
    if (!mlist) {
        fprintf(stderr, "Error: Failed to create member list.\n");
        return 2;
    }

    // Process log file or process stdin if no file provided
    const char *filename = argv[1];
    FILE *fd;
    if (strcmp(filename, "-") == 0) {
        fd = stdin;
    } else {
```

SP Assessed Exercise 1a

```
        fd = fopen(filename, "r");
    }

    if (!fd) {
        fprintf(stderr, "Error opening file.\n");
        return 2;
    }
    process_file(fd, mlist);
    if (fd != stdin) {
        fclose(fd);
    }

    //Create current date for comparison
#ifndef LIVE
    Date * now = date_create("10/10/2025 12:00");
#else
    Date * now = date_now();
#endif
    // Output all members and their last status
    MemberIterator *it = memberlist_iter_create(mlist);
    if (it) {
        MemberNode *node;
        while ((node = memberlist_iter_next(it)) != NULL) {
            const char *username = membernode_username(node);
            UserStatus *status = membernode_status(node);
            Date *date = membernode_last_activity_date(node);
            char *date_str = date_format_last_seen(date, now);

            if (username && date_str) {
                if (*status == ONLINE) {
                    printf("%s (%s)\n", username, status_to_string(*status));
                }
                else {
                    printf("%s (%s: %s)\n", username,
                           status_to_string(*status), date_str);
                }
            }
            free(date_str); // date_format allocates a new string
        }
        memberlist_iter_destroy(it);
    }
    date_destroy(now);
    memberlist_destroy(mlist);
    return 0;
}
```

The main program is invoked as

```
./server-monitor [log_file]
```

If no file is present in the arguments, `stdin` will be processed. Additionally, if a filename is the string “-”, the program will process `stdin` at that point.

The mainline functionality of `server-monitor.c` consists of the following pseudocode:

```
create a MemberList
if no file args are provided
    process stding
else
```

```
open the file
process the file
close the file
get current date
create an iterator
while there is another entry in the iterator
    print out the member, status, and last updated date
destroy the iterator
destroy current date struct
destroy the member list
```

A static function `process` is provided to process all log entries in a particular log file. The functionality of `process` consists of the following pseudocode:

```
For each line in file
    process date, username, command
    if command is JOIN:
        call memberlist_add
    if command is LEAVE:
        call memberlist_remove
    if command is STATUS:
        call memberlist_update_status
```

4 Implementation

4.1 Implementation Requirements

1. Files to implement

Implement `date.c` and `memberlist.c`. Your implementations must match the function prototypes specified in the headers listed in Section 3 above.

2. Struct declarations

You do not need to redefine `typedefs` in your .c files; the provided .h header files already handle this. You should simply declare the necessary `structs`.

3. Compiler instructions

If you get `typedef` errors, try adding `-std=c11` to the compile command.

Your code must work correctly with the clang compiler without using any `--std` flags on the School's **stlinux01...12** servers, regardless of which platform you use for development and testing (e.g. your personal computer). Leave enough time to fully test on the **stlinux** machines before submission.

4. Data Structure to use

You should use a **skip list** as the basis of the **MemberList** structure.

Note that this is a randomized skip list. You should use the given `select_level()` function to select the level of a new node.

Please read the comments in Section 3 carefully and ensure that your implementation follows all the requirements.

4.2 Evaluation

Your marks for each source file will depend upon its i) design, ii) implementation, and iii) correctness during execution.

Memory leaks will incur penalties.

The driver program `server-monitor` will be tested against some *VERY LARGE* and log files to see if you have correctly implemented your skip list.

Note that if your code does not compile, **you will not receive any marks** for compilation and output, but you could still gain marks for implementation. The complete marking scheme is provided below in Section 6.

4.3 Testing your implementation

In addition to `server-monitor.c`, `date.h`, and `memberlist.h`, you are provided with `memberlist-linkedlist.o`, which is a **linked list implementation** of `memberlist.c`. This permits you to test your implementation of `date.c` against a working, **albeit less efficient**, implementation of `MemberList`.

You are also provided with two sample input files (`small.txt` and `large.txt`) and the respective output that your program should generate for those input files (`small.out` and `large.out`). The following commands should yield NO output if you have implemented your ADTs correctly:

```
% ./server-monitor small.txt | diff - small.out  
% ./server-monitor large.txt | diff - large.out
```

4.4 Implementation/Debugging advice

1. String manipulation

For string manipulation, we advise against using `strtok()` as it can cause undefined behaviour. Instead, use `strrchr()` along with `strdup()`.

2. Debugging on the stlinux servers

You can access `lldb` on the servers.

3. Checking for memory leaks

It is sufficient to check memory/leak sanitizing on a lab device or on your personal device. You do NOT have to check memory/leak sanitizing on the stlinux servers.

5 Submission – Please read carefully!

Submit your solutions electronically on Moodle, into the workshop set for Coursework 1:

<https://moodle.gla.ac.uk/mod/workshop/view.php?id=5507347>

We will be using the same workshop for Coursework 1b – Peer Code Review.

The servers are **NOT** a submission system. Instead, they serve as a unified platform where code from all students will be checked to avoid platform-specific discrepancies.

Your submission must be ANONYMOUS. There should NOT be anything in your code or report that can identify you e.g. your name, GUID, or any other personal information.

If you intentionally or accidentally identify yourself, you will be heavily penalised!

The files you submit are expected to be EXACTLY as follows, including filenames:

- date.c
- memberlist.c
- cw1report.pdf (in PDF only, contents are outlined below)

All your code should go into the two .c files listed above. The .h files should remain unchanged. Please check your code against the provided header files as they are.

You are responsible for the files you submit and will be marked accordingly. “*Oh, I submitted the wrong file*”, “*I forgot to add this line*”, etc. are **not** acceptable excuses.

Comments are not expected, but adding some might make the job of the person marking your submission a bit easier, so use your discretion.

The PDF report should include the following:

- An “**authorship statement**” that clearly indicates for each part of your submitted code whether *either* of the following applies:
 - “This is my own work as defined in the Academic Ethics agreement.”
 - “This is my own work except for...”
- The overall state of your solution based on running on an STLinux server:
 - a) **Compilation:** Does it compile without errors or warnings?
 - b) **Test runs:** Does it produce the expected output with the small and large test inputs (see previous section)?
 - c) **Bugs:** Does your code have any bugs that you are aware of?

If your code works as expected, the report could be only a few lines long.

Your report needs to be accurate. For instance, reporting that everything works when it does not compile is offensive.

Submissions will be checked for collusion and plagiarism. It is better to submit an incomplete solution that is your own than a copy of someone else’s work.

6 Marking Scheme

Your submission will be marked on a 100-point scale. Substantial emphasis is placed on **WORKING** submissions, and you will note that a large fraction of the points is reserved for this aspect. Please note that successful BST implementation depends on the date ADT. In addition, it is to your advantage to ensure that whatever you submit compiles, links, and runs correctly, even if it is not a complete implementation of all the required specification.

The marking scheme is as follows.

Points	Description
10	Your report – accurately & honestly describes the state of your submission
20	<u>Date ADT</u> 6 for workable solution (looks like it should work) 2 if it successfully compiles 2 if it compiles with no warnings 6 if it works correctly (when tested with an unseen driver program) 4 if there are no memory leaks
70	<u>MemberList Skip List ADT</u> 30 for workable solution of the members (looks like it should work) 2 if it successfully compiles 2 if it compiles with no warnings 2 if it successfully links with server-monitor 2 if it links with no warnings 18 if it works correctly with small.txt and large.txt 4 if it works correctly with 10,000 entry unseen log file 4 if it works correctly with 100,000 entry unseen log file 6 if there are no memory leaks

The following points should be noted regarding the marking schemes.

- Your report needs to be accurate and honest. THIS WILL BE ONLY TESTED ON THE SCHOOL SERVERS. COMPIILING ON YOUR MACHINE WILL NOT BE CONSIDERED AS EVIDENCE. The 10 points associated with the report are probably the easiest 10 points you will earn, as long as you are honest.
- If your solution does not look workable, then the points associated with successful compilation and a lack of compilation errors are **not** available to you. This prevents you from handing in a stub implementation for each of the methods in each ADT and receiving points because they compile without errors but do nothing.
- The points associated with “workable solution” are the maximum number of points that can be awarded. If it is deemed that only part of the solution appears workable, then a portion of the points in that category will be awarded.