A+ will be down for a version upgrade on Tuesday 03.01.2023 at 9-12.

**This course has already ended.**

« Programming Project (/elec-a7100/202… Example Exam » (/elec-a7100/2021-sum…

ELEC-A7100 (/elec-a7100/2021-summer/)
/ Programming Project (/elec-a7100/2021-summer/programming-tasks/)
/ 1 Instructions & Task Assignment

# Instructions & Task Assignment¶

In the programming task you will create an entire program consisting of a number functions from scratch. The task is selected randomly for you, and you are required to submit your implementation below. Please read the guidelines and task descriptions carefully.

# Programming Task¶

Course Exercises

Implement a system that tracks the exercise points for a course. The course has six exercise rounds, for which points are tracked separately. For each student, the system will store a student number, first name and last name, and the points for six exercise rounds as integers. For strings the system should store at least 20 characters, but you can decide yourself how to handle longer strings. You can assume that the student number is at most 6 characters long. The program should support arbitrarily large number of students.

System has the following functions:

- *A number lastname firstname* : **Add student**

  Adds a student by given studen number to the database. Initially the student has 0 points. For example: `A 234567 Opiskelija Osmo` *(1 p)*

- *U number round points* : **Update points**

  Sets the points for exercise round ”round”. You can assume that there are at most 6 exercise rounds. The points will be integers. If the given student number does not exist, an error message should be given. *(1 p)*

- *L* : **Print points**

  Prints the stored students, with their student number, last name and first name, along with points of each exercise round and total points. The students should be printed in the order of total points such that the student with most total points will be printed

first. This command is worth two points such that a functional output that contains all students will give one point, and if the order is correct, you will get a second point. *(max. 2 p)*

- *W filename* : **Save results**

  Writes the results to a file with given filename. *(1 p)*

- *O filename* : **Load results**

  Loads the results from file, and replaces the existing results in memory. *(1 p)*

- *Q* : **Exit program**

  Exits program and releases all allocated memory. This operation must be implemented so that you can find all possible memory leaks.

In addition to the above commands, there should be a working main function that repeatedly asks commands from user and acts accordingly.

The commands start with one capital letter and may be followed by some number of parameters, separated by space, depending on the command. Below is an example of possible input sequence:

```
A 234567 Opiskelija Osmo
A 111111 Ahkera Antti
U 234567 1 7
U 111111 1 14
U 111111 2 12
L
W pisteet
Q
```

As a result, Osmo Opiskelija will have altogether 7 points, and Antti Ahkera a total of 26 points. In the results table Antti Ahkera will be output first, and the the score database will be stored to file ”pisteet”.

Fighters

Implement a small role playing game, in which the game characters fight against each other. For each character on should define at least

- **Name** that identifies the character.
- **Hit points** (HP) that defines the health of the character. HP is reduced when the character gets damage in the fight. The character will die when HP reduces to zero.
- **Experience points** (Exp) that increases when the character hits its opponents in the fight. You can decide how the experience points increase (for example, by the amount of inflicted damage, with bonus on slaying, etc,). The experience points always start from zero on a new character.
- **Weapon** (i.e., a string with the weapoin name), and its **maximum damage**. In the simpliest form the weapon can always inflict the same amount of damage, but you can define also so that a hit causes a random amount of damage.

In the game one can create arbitrary many characters, that are being tracked by the game.

The program has the following functions:

- *A name HP weaponname weapondamage* : **Add character**

  Adds a character with given *name*. The character has initially *HP* hit points, and the given weapon, with maximum damage as indicated in "*weapondamage*". For example: `A Bilbo 25 Dagger 8` . *(1 p)*

- *H attacker target* : **Attack**

  Character "*attacker*" will attack towards character "*target*". As the result target's hit points will be decremented by the damage caused by the weapon (that can be randomized, if you wish). The outcome of attack must be printed to the user. For example: `H Bilbo Smeagol` . An example output could then be: `Bilbo attacked Smeagol with Dagger by 6 damage. Smeagol has 8 hit points remaining. Bilbo gained 6 experience points` *(1 p)*

- *L*: **List characters**

  Prints all characters added to the game, each on their own line. For each character all the above described properties must be printed (name, HP, Exp, Weapon). The characters should be ordered by their experience points: the character with most experience points will be printed first. However, the dead characters should be printed at the end of the list, regardless of their experience points. If output works, but the order is wrong, you will get one point. If the order is correct, you will get *max. 2 points*.

- *W filename* : **Save game**

  Saves all characters in the game (including the dead ones) to file named "filename". *(1 p)*

- *O filename* : **Load game**

  Loads characters from file name "filename", replacing the current setup in memory. *(1 p)*

- *Q* : **Exit program**

  Quits program and releases all allocated memory. This function is must be implemented in order to detect possible memory leaks.

In addition to the above commands, there should be a working main function that repeatedly asks commands from user and acts accordingly.

The commands start with one captial letter and may be followed by some number of parameters, separated by space, depending on the command. Below is an example of possible input sequence:

```
A Bilbo 25 Dagger 8
A Gimli 45 Axe 12
A Smaug 120 Fire 18
H Bilbo Smaug
H Gimli Smaug
H Smaug Gimli
L
W game
Q
```

With these commands three characters are created. First Bilbo and Gimli will attack Smaug, that will attach back towards Gimli. After this the game situation is printed, and saved to file named ”game”.

Game shop

Implement a simple system that holds record of games in a game shop, and their sales. For each game we store the name, price per unit and total sales of the title. For the name the program must be able to handle a string of at least 20 characters, but you can decide how to handle longer names than that. Prices and total sales are floating point numbers. The program must be able to store a arbitrary large number of games.

The system has the following operations:

- *A name price* : **Add game**

  Adds a new game to the database. Initially the total sales of the game is 0. For example: `A Testipeli 99.99` *(1 p)*

- *B name N* : **Buy game**

  Buy the game (as indicated by name) ”N” units, i.e., add the total sales by the amount of game price and bought units. If the given game is not found, or N is given incorrectly, an error message should be given. *(1 p)*

- *L* : **Print game information**

  Prints all the games in the shop, along with their name, price per unit and total sales. The games should be listed in the descending order based on their total sales, i.e., the best sold game is shown first. The prices must be output with two decimals. This part is worth two points such that correct output, even if in incorrect order gives one point, and if the ordering is correct, there will be second point. *(max. 2 p)*

- *W filename* : **Save games**

  Saves all game information to a file indicated by the filename. *(1 p)*

- *O filename* : **Load games**

  Load game information from the given file, replacing the existing database in memory. *(1 p)*

- *Q* : **Exit program**

Exits program and releases all allocated memory. This operation must be implemented so that you can find all possible memory leaks.

In addition to the above commands, there should be a working main function that repeatedly asks commands from user and acts accordingly.

The commands start with one capital letter and may be followed by some number of parameters, separated by space, depending on the command. Below is an example of possible input sequence:

```
A Tetris 9.99
A Wormgame 4.50
B Tetris 1
B Wormgame 5
L
B Tetris 3
L
W pelit
Q
```

As a result of these commands, total sales of Tetris should be 39.96 EUR, and Wormgame sales should be 22.50.

Olympics

Implement a system that tracks the olympic medals of different nationalities. You can enter different countries to the system, and maintain the medal counts for them. The system should support arbitrarily large number of different nations. The database will consist of names of the nations (strings), and integers that represent gold-, silver- and bronze medals for each nation. For the name of the nation arbitrarily long names should be supported.

The program has the following functions:

- *A nation*: **Add nation**

  Add "nation" to the database. Initially each nation has no medals. For example: `A Finland` *(1 p)*.

- *M nation gold silver bronze* : **Update medals**

  Adds the given amount of medals to the given country. For example `M Finland 0 1 1` will add one silver and one bronze medal to Finland, in addition to existing medals. If the nation was not yet added using the A - command, an error message will be given. The medal counts are signed integers, and because of possible doping cases also negative adjustments need to be allowed, which means that the total count of the particular medal will be reduced. *(1 p)*

- *L* : **Output medal table**

Prints the current medal table, i.e. all nations in the memory along with their medal counts: first gold, then silver, and finally bronze, each nation on a separate line. The nations should be listed in order such as first the nation with most gold medals will be output. In cases where there are equal number of gold medals, the number of silver medals counts. If also the silver counts are equal, the number of bronze medals counts. This command is worth of two points such that if you output all nations and medal counts correctly, but in wrong order, you will get one point. If, in addition, also the order is correct, you will get a second point. *(max. 2 p)*

- *W filename* : **Save table**

  Writes the medal table to a file with given filename. *(1 p)*

- *O filename* : **Load table**

  Loads the medal table from file, and replaces the existing table in memory. *(1 p)*

- *Q* : **Exit program**

  Exits program and releases all allocated memory. This operation must be implemented so that you can find all possible memory leaks.

In addition to the above commands, there should be a working main function that repeatedly asks commands from user and acts accordingly.

The commands start with one captial letter and may be followed by some number of parameters, separated by space, depending on the command. Below is an example of possible input sequence:

```
A China
A Finland
M China 2 1 1
M Finland 0 0 1
M China 1 3 1
M China -1 0 0
L
W medals
Q
```

After this, China has 2 gold medals, 4 silver, and 2 bronze, but Finland has only one bronze. When the table is printed, China should be shown before Finland. The table will be written to file ”medals”.

Rally

Implement a results system for a rally race that consists of several special stages. For each driver, the last name is stored, along with his team. In addition, the system maintains the overall time for each driver. Time consists of hours, minutes, and seconds. You can decide how the time is stored in memory, but it should be possible to store an arbitrarily large number of drivers. You can assume that the last name identifies the driver. For strings, at least 20 characters should be supported, but but you can decide how to handle longer strings.

The program has the following functions:

- *A lastname team* : **Add driver**

  Adds driver named "lastname" to the database, who is representing "team". Initially, the total time will be 0 seconds. For example: `A Kankkunen Renault` *(1 p)*

- *U lastname hours minutes seconds* : **Update total time**

  Adds the given time to drivers total time, for example after a completed special stage. For example: `U Kankkunen 0 52 16` will add 52 minutes and 16 seconds to the total time of driver Kankkunen. If the given driver was not yet added to the database, an error message should be printed. *(1 p)*

- *L*: **Print results**

  Prints the current results of the race, i.e., all drivers along with their teams and total times, each on a separate line. The drivers should be printed fastest first, i.e., the driver with smallest total time will be printed first. This command is worth of two points such that a correct output of all drivers (and other data) will provide one point, even if the order is wrong. If the order is correct, you will get another point. Remember that an hour has at most 60 minutes, and a minute has at most 60 seconds *(max. 2 p)*

- *W filename*: **Save results**

  Writes the current database to file with given filename. *(1 p)*

- *O filename*: **Load results**

  Loads the database from file, and replaces the existing results. *(1 p)*

- *Q* : **Exit program**

  Exits program and releases all allocated memory. This operation must be implemented so that you can find all possible memory leaks.

In addition to the above commands, there should be a working main function that repeatedly asks commands from user and acts accordingly.

The commands start with one capital letter and may be followed by some number of parameters, separated by space, depending on the command. Below is an example of possible input sequence:

```
A Kankkunen Renault
A Latvala Volkswagen
U Kankkunen 0 52 16
U Latvala 1 01 20
U Kankkunen 0 49 50
U Latvala 0 47 15
L
W tulokset
Q
```

As a result, Kankkunen will have total time of 1 hour, 42 minutes and 6 seconds, while Latvala has 1 hour, 48 minutes and 35 seconds. Kankkunen will be printed before Latvala in the results table. The table will be written to file "tulokset".

Scheduling

Implement a system for scheduling that can be used to schedule one-hour meeting times. There should be only one reservation per hour. It should also be possible to cancel meetings and print them according to time. The meeting should be presented with one string description and at least 20 characters should be supported, but you can decide how to handle longer strings. You can assume that there are no white spaces in the string. The string for the meeting doesn't have to unique. The system can only handle one year at the time so you don't need to keep record of the year.

The program has the following functions:

- *A description month day hour*: **Add entry**

  Add an entry with the given parameters to the scheduling system. Check that month, day and hour are reasonable. You can assume that every month would have 31 days. If the given time is already reserved, error message should be printed. For example, `A Haircut 3 26 14` would reserve a slot for haircutting on 26.3. at 14. The system should use 24-hour clock. *(1 p)*

- *D month day hour*: **Delete entry**

  Delete an entry with the given parameters. If there is no entry with the given parameters, error message should be printed. For example, `D 3 26 14` would delete the entry added above. *(1 p)*

- *L* : **Output calendar**

  Prints the current entries according to scheduled time. The earliest entry should the printed first and the latest last. For every entry, description and time should be printed in the following format: `description XX.YY. at ZZ` where XX is the day, YY is the month and ZZ is the time. You will get full (2) points if the entries are printed right in the correct order, if the order isn't correct but the printing is otherwise right, you will get one point. *(max 2 p)*

- *W filename* : **Save calendar**

  Writes the calendar to a file with the given filename. *(1 p)*

- *O filename* : **Load calendar**

  Loads the calendar from file, and replaces the existing calendar in memory. *(1 p)*

- *Q* : **Exit program**

  Exits program and releases all allocated memory.

  **This operation must be implemented so that you can find all possible memory leaks.**

In addition to the above commands, there should be a working main function that repeatedly asks commands from user and acts accordingly.

The commands start with one capital letter and may be followed by some number of parameters, separated by space, depending on the command. Below is an example of possible input sequence:

```
A Haircut 3 26 14
A C-lecture 3 27 12
A C-exercise 3 27 14
A Math_lecture 3 26 14
D 3 26 14
A Math_lecture 3 26 14
L
W calendar
Q
```

After this, the calendar should save the first three entries since the first "Math_lecture" shouldn't be successful because the time is already reserved in previous entry. After the entry is deleted with the command "D", adding the "Math_lecture" should be successful. After these, the calendar is printed and saved to the file "calendar".

⚠ The deadline for the assignment has passed (Monday, 30 August 2021, 19:59).

You cannot submit this assignment

The deadline for the assignment has passed (Monday, 30 August 2021, 19:59).

Show anyway

## Assigned Project

Refresh the page to see your assignment.

**project.c**

Choose File No file chosen

**project.h**

Choose File No file chosen

Submit

# Evaluation¶

**The task will be evaluated on score range from 0 to 10.**

Important

1. The program will be compiled with the following gcc command line options:

```
-c -std=c99 -g -Wall -Wextra -Wno-missing-field-initializers
```

2. If compilation gives errors that prevent creating an executable, or the given command line format is not exactly followed, the assignment cannot be accepted, and you will get no points. A good strategy is to proceed on functionality at a time and test it before starting to implement the next function.
3. The program will be evaluated and tested on this A+ server. **However, you should implement and test your program in your development environment, and only submit the ready program in the end.**
4. Your submission will be also checked for memory leaks using `valgrind` using the the following options:

   ```
   --trace-children=yes --leak-check=full --show-leak-kinds=all
   ```

5. We will evaluate you submission manually, so you will not get any automatic grade with the submission, but you can observe test results of your code once you submit your code.

**Note:** The test input in A+ system will operate as a file that ends with end-of-file mark `EOF`. Be prepared to handle `EOF` in your program when you parse the user input. End of file may occur in any case also in normal use of the program.

# Implementation and Submission¶

Implement the program in single .c file. You can also use one header file for your definitions, but you can include them also in the .c file and omit the header file.

**You can design your program in any way you want.** However, grade is affected by readability of you code, which mainly depends on the programming style you use.

1. Design your implementation in a clear structure. i. Divide the implementation into functional blocks, and implement them as functions.
    2. Select appropriate variables that are required to implement the state machine of the project.
2. Apply a consistent naming convention for your variables and functions.
    1. Functions should be named so that they imply their operation.

       For example, if you have a function to allocate an instance of *user* structure and initialize its members, you can name the function as `allocate_user(...)`.
    2. Variables should be named to imply their purpose.

       For example, if you declare a variable to hold the size of a statically allocated array, you can name it as `size_t arraySize = sizeof(array);`.

Submission

- **You should submit your implementation in the submission box above.** The submission box will be active till the end of the course, but submissions later than the deadline will be penalized according to the rules described below.
- **You can see the deadline on the submission box.**

- **You can submit your implementation more than once, but only the latest submission will be evaluated.** If you have submissions both before and later than the deadline, only the latest will be evaluated and penalized accordingly.

# Grading¶

**Max: 10 points.**

- Each functionality that is implemented correctly: 1 point, in L-command max. 2 points (total max. 6 points)
- Working main function and command parsing: 2 points (required to pass)
- Appropriate programming style (distribution to functions, appropriate indentation, appropriate naming, comments as needed): 2 points
- Compiler gives warnings: max. **-2 points**
- There are valgrind errors: max. **-2 points**

Late Submissions

If you submit your work late, the points will be reduced as follows:

- 0 - 24 hours late: -1 point
- 24 - 48 hours late: -3 points
- 48 hours or more late: you will get max. 5 points regardless of what features have been implemented (i.e., a badly late submission will get your work accepted with bad points).

# Guidelines¶

1. If you have a header file for your definitions, you must name it as `project.h`, and submit it along with your project file. It might help you if you name your project implementation file as `project.c`.
2. To facilitate testing, the aforementioned **command syntax must be precisely followed**.
3. After every command, the system must give an output that tells whether the command was successful, or whether its execution failed.
    1. Each function must give a clear output that either confirms successful operation, or an error message as needed.
        - You can see example outputs when you submit your project to A+ system. In particular, when a command completes successfully, example output is `SUCCESS` followed by a new line.
    2. Clearly erroneous commands must give on error message, and at least the program must not crash as a result of invalid command.
        - You can assume that names and other strings do not contain whitespaces (i.e., whitespaces can be assumed a command field separators).
4. You must use dynamic memory to implement the storage. A large static array is not acceptable. The dynamic memory allocation should be done based on the actual need: `malloc(1000000)` is **not acceptable**.
5. You can assume a maximum length for user input line. For example, 1000 characters is sufficient.

# Getting Started¶

1. You can use *Visual Studio Code* (or any other editor) to write your project.
2. Learn how to compile and link in your development environment. * Since your submission will be tested using `gcc`, it is recommended to use `gcc` compiler. * Use the compiler flags given above when compiling your code.
3. Start by writing a `main` function since it is called automatically when executation of your code starts.

```
int main(void) {
    // The magic starts here
}
```

4. Make an infinite `while` loop inside your `main` function. Your application should terminate only when it receives `quit` or `exit` command.
5. Read a line from user (e.g. using `fgets` from `stdin`)
6. Resolve command code either using `sscanf` or get just first character on an input line.
7. Use `if-else` or `switch` branches for processing each user command.
8. Process command parameters using `sscanf`.
9. Have a dedicated function for each command, and name the functions appropriately.
10. Implement and test one function at a time. **Compile and Test often.** Move to the next function only after previous works.
11. Check your executable using `valgrind` to see whether your code treats memory correctly.

# Hints¶

- Command handling functions are covered in **Module 4 Strings** and **Module 9 I/O Streams**.
- You might simplify command handling using function pointers covered in **Module 10 Advanced Features** and structured data types and multi-dimensional arrays covered in **Module 7 Multidimensional Arrays**.
- File reading and writing are covered in **Module 9 I/O Streams**.
- The database related oprations can be implemened using `linked-list` covered in **Module 6 Structured Data Types**.

# How to interpret the submission feedback?¶

When you press the `Submit` button after choosing your project files, the A+ system performs the following operations:

1. **The submitted source code `.c` file is compiled using the specfied `gcc` compiler flags.**

- If there are compiler errors, the process ends at this point and the compiler output is shown.
- Otherwise, the compiler output is checked for warnings. If there are warnings, the specified penalty is marked.

2. **A list of normal and stress tests are created for the assigned project.**
   - Each test assumes that your code follows the specified command syntax precisely.
   - **All commands are composed of a single (upper case) character and whitespace seperated variable number of arguments.**
   - The test list always ends with `W <filename>` command followed by `Q` command.
   - *The saved file* `<filename>` *can be binary or text file*.
   - **However, if it is binary, the system cannot convert its content to human readable form. Thus, you are strongly discourage to implement** `W` (Save) **command using binary files.**
   - **Beware that, if** `W` **command uses text files, so should the** `O` (Load) **command.**

3. **The compiled executable is run as an independent process**.

4. **Each entry of the created test commands list is input to the running process sequentially.**
   - It is important to understand that the system assumes your program reads the input as a line of `C arg1 arg2 ...` where `C` is the command, and `arg` represents the proceding arguments of the command. If the command does not have an argument, then `C` should be followed by `\n` which corresponds to hitting the `<Enter>` key.
   - It is for the best to **avoid user query strings** such as `Please enter a command:` or similar.
   - The system assumes that the printed output is `US-ASCII` encoded. **You should not use non-ASCII characters**.

5. **After each command, the system waits for** `stdout` **strings to be printed.**
   - As mentioned above, when there is an error, the program should indicate that error as a `stdout` print.
   - When the command execution completes successfully, you are encouraged to print `SUCCESS\n` on the screen so that the user is informed and the evaluation system can capture the output associated with the current command being tested.
   - If the executable does not print anything on the `stdout` for more than 2 seconds, **that test is marked as** `timed out` . This mark implies that manual code-inspection based evaluation will be applied. Such tests have **yellow-like background color** to indicate that when evaluating your project, a closer look is required. **This usually increases the chance of incorrect evaluation**.
   - If the entered command or associated arguments causes the executable to crash, that test is marked as `incomplete` . Such tests have **red background** color to indicate that **your implementation has serious problems and likely to fail the associated command evaluation.**.
   - **Otherwise, the acquired output from the** `stdout` **is compared with the example output string.**

- If these two strings exactly match, the result of the test is assigned as `True`. Such tests have **green background color** to show that everything works as expected.
- If the acquired string and example output strings do not match, the result of the test is assigned as `Unevaluated`. Such tests have **yellow-like background color** to indicate that when evaluating your project for that test case, a closer look is required.

6. **After running all the tests, the compiled executable is executed using `valgrind` with the stated arguments.**
   - The previously created test list is input in the same order.
   - This time the outputs are not acquired, but the valgrind messages are stored.
   - If there are `valgrind` errors, the stated penalty is marked.

7. **Finally, the output file input to the last `W <filename>` command is read.**
   - If the string content of the file can be successfully read, it is shown as the `Acquired Output` box of the feedback page.
   - As a reference, an example output is also shown on the left of it. The example output is the output one would expect if the program correctly handles the commands.
   - The output is not evaluated by the system, and just aims at showing the expected end result of the test list.
   - **The red color of the Output header bar does not indicate a failure**.

« Programming Project (/elec-a7100/202… Example Exam » (/elec-a7100/2021-sum…