Project Topics
for the Course
Programming in C++

Petr Gajdoš, Jakub Beránek and Vít Doleží

February 5, 2021

1 Game Rules and Project Evaluation

All the points below must be met:

1. C++11 and above is accepted. C++17 is a recommended version (e.g. auto_ptr is removed, ...)
2. All project must be unique from the perspective of source code. The same or very similar solutions will be excluded from the evaluation and authors will be not classified.
3. The source codes must be written with respect to versatility, high performance, and minimal memory consumption; e.g. no wasted lines of codes, using generic classes (templates), using suitable data types/structures, avoiding memory leaks, using exceptions, etc. All these aspects will be evaluated.
4. Students have to provide sample data with relevant description. This data will be used in an application verification process.
5. The limitations of the application must be clearly defined, if any.
6. The solution must be consulted with the lecturer at least once during the semester.
7. All projects must be submitted in a form agreed with the lecturer until the end of organized education of the semester. The lecturer may change this deadline.

Code purity, application performance, and memory requirements will be included in the project evaluation.

2 Project 1

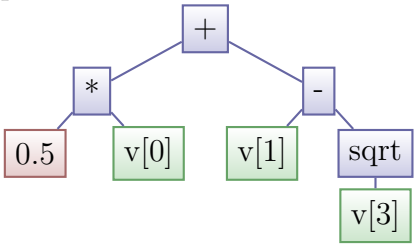
The main goal of this project consists in an implementation of a tree structure and related functionalities. A tree will represent some mathematical formula, that can be later evaluated. Result type must be a generic type T. It is evident, that depths of trees can vary depending on the number of operators. Next, a tree can contain certain types of nodes: constants, operators (functions), variables. Generally, all variables can be represented by a vector (array) v in N - dimensional space; i.e. v_i, where $i \in \langle 0 \dots N-1 \rangle$. This is important because input data for the evaluation will be represented by a set of such vectors. Input data for the tree construction is represented by a mathematical formula in a form of string.

2.1 An illustrative example

1. Let's have an input string $(0.5 * v[0]) + (v[1] - sqrt(v[3]))$ in the file *formula.txt*. Moreover, vector dimension N (the number of elements = length on input vectors) must be also defined.
   Important note: Some of vector elements may be omitted in the formula; e.g. $v[2]$ in this example.

   formula.txt:
   N 4
   $(0.5 * v[0]) + (v[1] - sqrt(v[3]))$

2. This file (*formula.txt*) must be parsed and a related tree structure must be created.



3. Les's have a set of K input vectors v_{k_i} written in some text file *inputData.txt*, where $k \in \langle 0 \dots K-1 \rangle$, and $i \in \langle 0 \dots N-1 \rangle$.

   inputData.txt:
   K 5
   N 4
   1.0 2.0 3.0 4.0
   4.0 5.0 6.0 7.0
   1.2 2.3 3.4 5.6
   12312 3213 54654 321
   9 8 7 4

4. Finally, the tree can be evaluated for each input vector v_k and all results will be stored in an output file *output.txt*.

2.2 Expected features of the solution

It is evident, that some features (methods) must be implemented to achieve required functionality.

- Input parser for mathematical formulas. Here you can simplify your solution with predefined set of accepted operators, mathematical functions. But the set must contain at least the following operators: +, -, *, /, %, exp, sqrt, sin, cos. Beware of using brackets.
- Input parser for data vectors.
- In case the tree can not be evaluated, the result is "undefined", all exceptions must be handled, e.g. division by zero, out of range exception, numeric limits, ...
- The tree structure will be a generic type with some constrains on this type. The given type is the type of expected results, i.e. "string" will never be the used type.
- Tree traversing (iteration) must be implemented.
- To visualize the tree structure, some print method must be implemented. Here, LaTeX package called "tikz" is recommended. Then the output is represented be a simple LaTeX text document. See https://texample.net/tikz/examples/tree/ for more details.
- Random tree generator. All parameters must be set in a JSON file, e.g. depth of tree, number of nodes, probabilities of node types, dimension of vectors, etc.
- Serialization and deserialization methods.

3 Rubik's Cube (RC)

The main goal of this project is to implement a suitable data structure that can represent Rubik's cubes with variable edge length, for example a cube 3x3x3, 4x4x4, 10x10x10, etc. Moreover, additional methods must be implemented, e.g. random shuffle with saving the sequence of rotations in a defined form, rotations defined by some proposed language, serialization and deserialization of the current state, simple visualization, checking the cube assembly, etc.

This is a recommended language used to describe transformations of cubes with respect to a variable edge length:

- 6 characters define all faces of the cube: F - Front, B - Back, R - Right, L - left, U - Upper, D - Down (B for bottom is already used).
- 1 character defines the rotation: ', e.g. F means a rotation of the front fact in the clockwise direction, whereas F' in the counter-clockwise direction.
- A numeric range X-Y defines one-based indices of faces to be rotated, e.g.: Let's have a RC of edge length equals to 3 (3x3x3 RC). Then 1-1R represents clockwise rotations of the first right face, 1-2R represents clockwise rotation of the first right face, and the second face from right.
  In case of 3x3x3 RC, the following is valid:
  - 1-1R = 3-3L'
  - 3-3R = 1-1L'
  - 1-2R = 2-3L'
  - 1-1R = 1R = R (some kind of simplification)
  - 2-2R = 2R = 2L' = 2-2L'
  - ...
- A number $i \in \langle 1, 2, 3 \rangle$ represents the number of 90°-rotations; e.g. 1R2 = rotate the first right face by 180° clockwise.
  In case of 3x3x3 RC, the following is valid:
  - 1R1 = 1R (some kind of simplification)
  - 1R2 = 1R'2 (inverse rotations)
  - 1R1 = 1R = 1R'3 (inverse rotations)
  - ...

If R face is rotated, colors on Front, Upper, Back, and Down faces are changed!!!

3.1 An illustrative example

Let's have an input file *initState.txt* that defines the initial state of the 3x3x3 RC. This file contains the edge length N an a list of initial rotations that must be done to set a completely assembled cube into an initial state. This is some kind of user defined random shuffle.

initState.txt:
N 4
1-2R'

U
1-2D2'
F3'
B1
...

3.2 Expected features of the solution

It is evident, that some features (methods) must be implemented to achieve required functionality. The set of features can be extended after discussion with lecturer.

- Suitable data storage of the Rubik's Cube.
- Suitable data storage for transformations (rotations). Users can move back in the history of transformations, and create new branches from the certain point of history.
- Parser of the input file. This parser will be also used to process user commands in a real-time. Moreover, is must be possible to parse commands in a simplified form as well (see the notes on simplification above).
- CLI: You should define your commands to be used in real-time interaction with users, e.g. *savecube*, *savehistory*, *loadcube*, *set ¡some rotation¿*, *createbranch*, *plot*, ...
- To visualize the cube, some print method must be implemented. Here, LaTeX package called "tikz" is recommended for a simple 2D visualization. Here, the cube can be unfolded into a plane. Then the output is represented be a simple LaTeX text document. See https://texample.net/tikz/examples/tree/ for more details. Advanced user can visualize the cube using e.g. SDL (2D or "pseudo"-3D). There are no limits. In case of SDL visualization, export of the cube in PNG format must be provided.
- Load and Save methods for the cube and its transformation history.
- Error handling: You have to handle all errors. A log file with error details should be created.

4 Key-value database

The goal of this project is to build a simple key-value database (a "Redis clone"). The database will store various data types (integers, strings, lists, sets, hash maps) and later look them up via a string key. It will also support basic queries, such as selection, update/append, removal, increment, aggregation queries (sum, average), etc. on items with a specific key or items with a given key prefix. It also has to be able to support disk (de)serialization so that you can save the database state and restore it later.
The database will offer both a programmable API (used for automatic testing) and a command line interface (CLI), which will allow the user to store and load data using a simple query language.

4.1 Illustrative example

The format of the API, of the CLI language and the implemented database features are to be discussed with your teacher. This example is purely illustrative.

4.1.1 CLI usage

```
$ ./kv-database --load-from database.bin
PUT age.alena 24
GET age.alena
25
PUT age.marek 18
PUT age.martin 26
SUM age
68
SAVE database.bin
```

4.1.2 API usage

```
auto database = Database::load_from("database.bin");
database.put("age.alena", 24);

int age = database.get<int>("age.alena");
assert(age == 24);

database.put("age.marek", 18);
database.put("age.martin", 26);

int sum = database.sum<int>("age");
assert(sum == 68);

database.save("database.bin");
```

4.2 Expected features of the solution

The solution will have to implement (at least) the following features:

- **Data storage**. The database will be able to store several data types indexed by a string key. It has to be able to store at least numbers, strings, lists and sets. As a bonus feature, you can also implement hash maps. You will need to design data structures to

hold the data in memory (the format is up to you). The solution should be reasonably efficient. You can e.g. simplify things by storing integers as strings, but you shouldn't store lists/sets as e.g. comma separated lists and parse them out on each operation.

- **Queries and operations**. The database will implement simple queries on the stored data, such as adding an item to a set, incrementing a number, calculating the sum or average of all numbers with a given key prefix, etc. At the very least, you have to implement put, get, remove and increment operations and sum and average queries.
- **Binary serialization and deserialization**. The database will be able to store its contents into a single file on a disk and also restore its content from a file. The format of the file is up to you, but it should be efficient (e.g. prefer binary serialization, not ASCII). You can also use a library for this, e.g. Protocol Buffers. The database should also be able to export its contents into a human readable format (e.g. JSON), but it doesn't need to know how to import data from this format.
- **CLI interface**. There will be a CLI interface available, which will expose all of the available operations of the database. It will parse simple user queries and print results. The CLI interface should be built on top of a programmable API. In other words, the database should be usable programmatically even without the CLI.
- **Error handling**. The database will implement proper error handling, both inside the API (e.g. attempt to fetch a missing key) and in the CLI (wrong input, non-existent database file path, etc.).

5 SDL graphics demo or a game

The goal of this project is to create a "visually interesting" application using a 2D graphical library or a game engine (such as SDL or SFML). The application should be reasonably complex, handling complex memory and window management, object hierarchies and user input. Crucially, it should produce interesting and complex visual output (using e.g. shaders).
This project is quite open-ended, discuss specific details with your teacher.

5.1 Illustrative example

Implement a "space station attack" game.

- **Tiles** The game will take place on various levels, which will be defined by the player using a (XML/JSON/ASCII) text file. Each level will be represented with a 2D grid of tiles, with different types (empty tile, wall, platform). Each tile can be drawn either using a bitmap image or using a procedural shader.
- **Gun placement** The player will be able to place various types of guns on tiles that allow it (e.g. platforms) using a drag'n'drop system. Each gun will have a different resource cost, damage, appearance and special effects (such as a freezing gun).
- **Enemy ships** In specific intervals, enemy ships will appear on the tile and they will try to reach the command base of the player. The guns placed by the player will shoot at the ships to prevent them from reaching it. Defeating a ship will award resources to the player. There will be several ship categories, with distinct speed, appearances and special behaviours. The ships will have to navigate the level by selecting the shortest path available to reach the goal (using graph algorithms, e.g. BFS or Dijkstra's algorithm).
- **Menu** There will be a menu at the beginning of the game, where the player can select the level and start a game. The game will also be able to save and load its state to the disk.

6 Paint

The main goal of this project is to make simple image editing/batch convert application. You cannot use any image library for working with images, all image editing, loading and saving functions have to be implemented by you.
You have to support 2 image formats, it is recommended to use lossless formats or their lossless variants as they are easier to work with e.g.: BMP, TGA, TIFF, PNG.
There will be 3 applications. First application will be used from CLI, you will write commands directly to the command line. Second application will take commands from file, it can be used for editing/converting multiple images at once. Third application will read commands from file and generate image based on those commands.

6.1 Command (functions) to implement

- required parameter will be passed right after the command separated by space
- optional parameter will be passed after the required parameters inside of curly brackets {} , values inside {} can have different order, only some of the optional parameters can be used
- option %|PX means whether to use relative (percentage) OR absolute (pixel) values
- you can use some library like OpenCV to visualize the changes on screen (optional, do this only if you want)

- **LOAD ./folder/*.bmp**- load images in folder or one image
- **SAVE ./folder/*.bmp**- where the loaded image(s) will be saved and in which format
- **COLOR r g b**- set global color for your next commands
- **Line** - draw line between two points
  - LINE %|PX x1 y1 x2 y2 {width: number, color: {r: number, g: number, b: number}}
  - width - sets width of line, default=1
  - color - sets color of line, this color will be used instead of global settings
- **Circle** - defined by its center point and radius
  - CIRCLE %|PX x1 y1 radius {fill: bool, fill-color: {r: number, g: number, b: number}, border-color: {r: number, g: number, b: number}, border-width: number}
  - fill - whether the inside of the circle will be filled or not
  - fill-color - sets fill color, this color will be used instead of global settings
  - border-color - sets border color, this color will be used instead of global settings
  - border-width - sets width of border line, default=0
- **Bucket** - sets color to the area with the same color (Flood fill)
  - BUCKET %|PX x1 y1 {color: {r: number, g: number, b: number}
  - color - sets color of the fill, this color will be used instead of global settings
- **Resize** - resize the entire image

  - RESIZE %|PX width height - with % use relative size

- **Rotation** - rotate image by +-90 degrees
  - ROTATE CLOCK - rotate image clockwise
  - ROTATE COUNTERCLOCK - rotate image counter clockwise
- **Invert colors**
  - INVERTCOLORS
- **Convert to grayscale**
  - GRAYSCALE
- **Crop** - define crop by top left corner and bottom rigth corner
  - CROP %|PX x1 y1 x2 y2 - with % use relative coordinates
- **UNDO** - undo previous command (can be used multiple times)
- **REDO** - redo previously undone commands (can be used multiple times)

6.2 Batch edit/convert

This application will have 1 arguments, input file with commands.

File input example *exampleFileBatchConvert.txt*:
LOAD ./images/
GRAYSCALE
RESIZE 71
COLOR 0 255 0
LINE % 0 0 100 100
COLOR 255 0 0
LINE % 0 100 100 0
SAVE ./images-converted/*.bmp

Commands in this file will load all supported images from folder *./images/*, it will convert them to grayscale, it will draw line from top left corner to right bottom corner of the image in green color, then it will draw line from bottom left corner to right top corner of the image in red color, then it will be resized to 71% size of original image. Finally they will be all saved as BMP (one of the supported formats) into *./images-converted* folder.

6.3 Generate image based on commands

This application will have 3 arguments, input file with commands, output resolution, output filename. It will generate image based on commands in file.

File input example *drawCzFlagCommands.txt*:
LINE % 0 100 50 50 {color: {r: 255, g: 0, b:, 0}}
LINE % 0 0 50 50 {color: {r: 255, g: 255, b:, 255}}
LINE % 50 50 100 50 {color: {r: 255, g: 255, b:, 255}}
BUCKET % 95 95 {color: {r: 255, g: 0, b:, 0}}

BUCKET % 95 5 {color: {r: 255, g: 255, b:, 255}}
BUCKET % 5 50 {color: {r: 0, g: 0, b:, 255}}

Usage example: *commandsToImage.exe drawCzFlagCommands.txt 3000x2000 czFlag.bmp*
This should draw flag of Czech republic to file czFlag.bmp with resolution 3000x2000.