Exercises — libzork

version **#1.0**

Yet Another Kind of Assistants 2024 <assistants@tickets.assistants.epita.fr>

Copyright

This document is for internal use at EPITA (website) only.

Contents

*https://intra.forge.epita.fr

Obligations

*Obligations are **fundamental** rules shared by all subjects. They are non-negotiable and to not apply them means to face sanctions. Therefore, do not hesitate to ask for explanations if you do not understand one of these rules.*

Obligation #0: **Cheating**, as well as sharing source code, tests, test tools or coding-style correction tools is **strictly forbidden** and penalized by not being graded, being flagged as a cheater and reported to the academic staff.

Obligation #1: The coding-style needs to be respected at all times.

Obligation #2: If you do not submit your work before the deadline, it will not be graded.

Obligation #3: Anything that is not **explicitly** allowed is **disallowed**.

Obligation #4: All your files must be encoded in ASCII or UTF-8 without BOM.

Obligation #5: **Global variables** are forbidden, unless they are **explicitly** authorized.

Obligation #6: When examples demonstrate the use of an output format, you must follow it scrupulously.

Obligation #7: Your submission repository must be **clean**. Except for special cases, which (if any) are **explicitely** mentioned in this document, an *unclean* repository may contain:

- binary files;[1]
- files with inappropriate privileges;
- forbidden files: `*~`, `*.swp`, `*.o`, `*.a`, `*.so`, `*.class`, `*.log`, `*.core`, etc.;
- a file tree that does not follow our specifications.

[1] If an executable file is required, please provide its sources **only**. We will compile it ourselves.

1 Introduction

File Tree

```
libzork/
├─ CMakeLists.txt
├─ include/
│  └─ libzork/
│     ├─ runner/
│     │  ├─ choice.hh
│     │  ├─ html.hh
│     │  ├─ interactive.hh
│     │  ├─ network.hh
│     │  ├─ runner.hh
│     │  └─ smart.hh
│     ├─ store/
│     │  ├─ network-store.hh
│     │  └─ store.hh
│     ├─ story/
│     │  ├─ node.hh
│     │  └─ story.hh
│     └─ variables/
│        ├─ action.hh
│        └─ condition.hh
└─ src/
   └─ *
```

All files in the file tree are to submit. Be careful, you **must** have a root `libzork/` directory.

1.1 Project overview

Libzork is a project that will allow you to practice the C++ knowledge that you learned during the past week. You will have to implement a branching story game engine that will let users follow interactive stories in which they make their own choices.

A branching story is a kind of story in which the reader makes decisions that have an impact on the plot. They usually use the first person so as to include the reader as the story's main character.

You will have to implement a static library that lets users implement and run such a story.

1.2 Project structure

The directory `include/` contains all exported header files. You will need to edit those header files to add you own variables and functions. All source files (`.cc`) should be contained in the directory `src/`.

You will use the build system CMake. There **must** be a `CMakeLists.txt` in your root directory and in the subdirectories containing code.

When executed from your root directory, the following commands **must** produce a dynamic library `libzork.so` in the directory `build`:

```
42sh$ cmake -B build
42sh$ cmake --build build --target libzork
```

Some header files are already provided. If needed, you are **only** allowed to – and encouraged to – **add** data members and member functions. You **must not** change the interface of the provided member functions. You may add any header and source files you wish.

This project is structured in three parts:

- The basic story implementation, that will barely allow you to pass.
- Additional features that include the variables and command line story runners.
- Two different and independant **completion paths**. You can chose either of them to get all points:
  - An 'Advanced Runner' path in which you write a new story runner and add builtin commands.
  - A 'Network' path that will allow you to make your runner interact with a RESTful API.

2 Let's write a story

2.1 Basic representation

A story is represented by a YAML file and a set of script files. The YAML file for a simple story looks like this:

```yaml
title: My awesome story
scripts-path: ./scripts
story:
  - name: welcome
    script: welcome.txt
    choices:
      - text: Explore the cave
        target: cave
      - text: Go in the forest
        target: forest
  - name: cave
    script: cave.txt
    choices: []
  - name: forest
    script: forest.txt
    choices: []
```

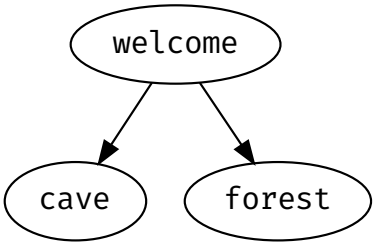
Let's look at this file step by step.

- `title` simply represents the story's title.
- `scripts-path` is a path to the directories containing the script files. The path can be either absolute or relative to the directory containing the YAML file. For instance, here is a possible file structure:

```
.
+-- script
|   +-- welcome.txt
|   +-- cave.txt
|   +-- forest.txt
+-- story.yml
```

- `story` is the description of the story graph. It is a list of nodes, each of which is a YAML object with the following keys:
  - `name` is an arbitrary label associated to the node. It is used by other nodes to reference it.
  - `script` is a path to the file containing the text associated to the node. This path is relative to `scripts-path`.
  - `choices` is a list describing all possible choices from the node. Each choice is another YAML object with the following keys:
    * `text` is the choice description.
    * `target` is the label of the node to which the choice leads.

Here is the graph representing the story described above:



If a node has an empty list of choices, it is a terminal node. Terminal nodes are the ends of the story.

2.2 Variables

The format described above allows users to create static stories. It is however also possible to change the choices available from a node dynamically, depending on previous decisions.

Variables are initialized via the `variables` key. It is a list of YAML objects describing the initial state of all variables. Each object has a `name` and `value` keys. Here is an example of variables definition:

```yaml
variables:
  - name: xp
    value: 0
  - name: health
    value: 100
```

In this story, the player starts with 0 xp and 100 points of health. Variable values are always 32 bits integers.

Once variables are declared, it is possible to use them as conditions for a choice to be available. The `conditions` key of a choice is a list of YAML object containing the following keys:

- `name` is the name of the variable.
- `value` is the value to compare the variable to.
- `comparison` is the comparison method to use. It must be one of:
  - *equal* ($=$)
  - *greater* ($>$)
  - *lower* ($<$)
  - *greater_equal* ($\geq$)
  - *lower_equal* ($\leq$)

A choice is available only if all its conditions are met.

Similarly, it is possible to change the value of a variable if a specific choice is made. The `actions` key of a choice is a list of YAML object containing the following keys:

- `name` is the name of the variable.
- `value` is the value used in the operation.
- `operation` is the operation to apply. It must be one of:
  - *assign*, which assigns `value` to the variable.
  - *add*, which adds `value` to the variable.
  - *sub*, which substracts `value` from the variable.

Here is how `conditions` and `actions` can be used:

```
[...]
  choices:
    - target: goblins
      text: Attack the goblins
      conditions:
        - name: health
          comparison: greater
          value: 20
      actions:
        - name: health
          operation: sub
          value: 10
        - name: xp
          operation: add
          value: 5
[...]
```

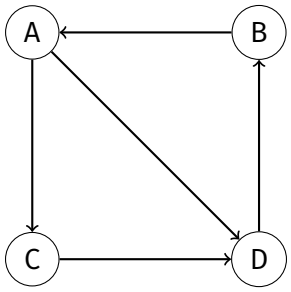
In this example, players can attack the goblins only if they have at least 20 points of health. If they chose to do so, they lose 10 points of health but earn 5 points of xp.

3 Story implementation

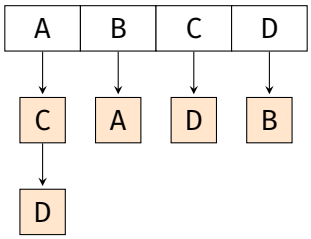
In this section, you will implement the story graph. This will be essential for the next steps so make sure to **thoroughly** test your code.

3.1 Graph representation

A commonly used representation for directed graphs is the adjacency list. It is a collection of lists, each containing the set of successors of a vertex. Consider the following graph:

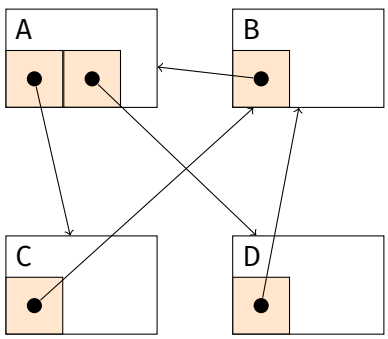


An easy way to implement an adjacency list is to assign a number to each of the vertices. The adjacency list is then an array indexed by vertices in which each cell is a linked list of the successors. For the above graph, the adjacency list could look like this (we use letters instead of numbers for clarity):



This implementation is easy to understand and relatively memory efficient. However, arcs are not explicitly represented and do not carry information. In the case of a story, an arc is a possible choice so we need to store its name and target, and optionally its conditions and actions.

A more flexible graph representation consists in storing the lists of outcoming arcs in objects representing the vertices. Each arc is an object in and of itself and stores a reference to its end vertex. This representation could look like this:



The orange squares represent arc objects and can contain any amount of additionnal information, beyond the end vertex. It is now time for you to implement this.

3.2 `Node` class

The file `include/libzork/story/node.hh` contains a declaration of the `Node` class. You must implement all member functions in the provided public interface.

Be careful!

Notice that there is *no* provided class to represent a choice, corresponding to an arc in the graph. Therefore, you **must** implement such a class. If you create a new file, do not forget to add its definition in your `CMakeLists.txt`.

The constructor is declared as:

```
Node(const std::string& name, const fs::path& script_path);
```

`script_path` is a path to the file containing the text of the node. At this point you should already have **private** data members.

Once the constructor is implemented, implement getters for the node name and text:

```
const std::string& get_name() const;
const std::string& get_text() const;
```

Then, `make_choice` returns the end vertex of the n-th choice (starting from 0). It must return `nullptr` if the index is out of bounds.

```
const Node* make_choice(std::size_t index);
```

You may and **must** add a **private** data member to represent the choices available from a node. You may use any STL container you wish. However, bear in mind that as per the C++ core guidelines, the containers that should be used most of the time are `std::array` and `std::vector`.

You will also need to list all available choices. Implement `list_choices` that returns the texts of all available choices in the order they are declared in the YAML story description:

```
std::vector<std::string> list_choices(check_conditions = true) const;
```

Tips

The optional parameter `check_conditions` will be used in the next section, you can ignore it for now.

Finally, the member function `add_choice` is used to add choices:

```
void add_choice(const Node* other, const std::string& text,
                const std::vector<Condition>& conditions = {},
                const std::vector<Action>& actions = {});
```

The optional parameters `conditions` and `actions` are used for variables too and should be ignored for now.

At this, point, you should be able to compile and run the following code, assuming the file `hello.txt` contains `Hello!` and `world.txt` contains `Hello, world!`.

```
story::Node hello("hello_node", "hello.txt");
story::Node world("world_node", "world.txt");

hello.add_choice(&hello, "self reference");
hello.add_choice(&world, "<random text>");

const auto also_world = hello.make_choice(1);
std::cout << hello.get_name() << ": " << hello.get_text() << "\n";
std::cout << also_world->get_name() << ": " << also_world->get_text()
          << "\n";
for (const std::string& choice : hello.list_choices())
    std::cout << hello.get_name() << " -> " << choice << "\n";
```

This code produces the following output:

```
hello_node: Hello!
world_node: Hello, world!
hello_node -> self reference
hello_node -> <random text>
```

3.3 `Store` class

The class `Store` will allow you to encapsulate the dynamic data of your program in a single class and make your implementation more flexible. It is declared in `include/libzork/store/store.hh`. For now, the only dynamic data is the current node of the graph. Implement an accessor and a mutator:

```
virtual const story::Node* get_active_node() const;
virtual void set_active_node(const story::Node* node);
```

Of course, you need to store the node in a private data member. If the node has not been set, `get_active_node` returns `nullptr`.

3.4 `Story` class

The story class is responsible for reading the YAML file and creating the graph. You can find its declaration in `include/libzork/story/story.hh`. To parse the YAML file, you are **strongly** advised to use the library yaml-cpp.

Be careful!

Because of the testing environment, you **must** use the command `find_library` instead of `find_package` to include the `yaml-cpp` library in your `CMakeLists.txt`.

The `Story` constructor takes a path to a specification file as parameter and parses it:

```
Story(const std::filestystem::path& path);
```

Once again, you must implement your own private data members. However, you **must** use the provided `nodes_` of type `std::vector<std::unique_ptr<Node>>` to store the nodes, as well as `store_` of type `std::unique_ptr<Store>` for the store associated to the story.

Going further...

The data member `nodes_` contains elements `std::unique_ptr<Node>` instead of simply `Node`. This is because we need to use raw pointers on the story. Storing pointers to elements in the vector could result in undefined behavior if the vector gets reallocated.

Notice that the copy constructor and copy assignment operator are deleted. Copying a story would indeed mean copying all of its data members, including `std::unique_ptr`, which is non-copyable.

Once this is done, implement accessors for the title and the current node and a mutator for the latter:

```
std::string get_title() const;
const Node* get_current() const;
void set_current(const Node* node);
```

Initially, the current node is the first one defined in the YAML file.

Finally, implement the non-member overload of `operator<<`:

```
std::ostream& operator<<(std::ostream& os, const Story& story);
```

Tips

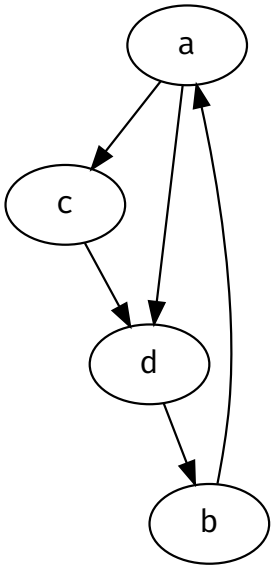
You might want to declare the << operator as a **friend** function if it needs to access private data members of the `Story` class.

This prints in the dot format. For example, the graph depicted in the section *Graph representation* has the following dot representation:

```
digraph story {
  "a" -> {"c" "d"};
  "b" -> "a";
  "c" -> "d";
  "d" -> "b";
}
```

You don't have to generate the same exact dot file. However, it must produce the same result with `graphviz`[1]. Here is the output of `dot -Tpng out.dot`:

[1] We use `dot -Tplain` to test this part

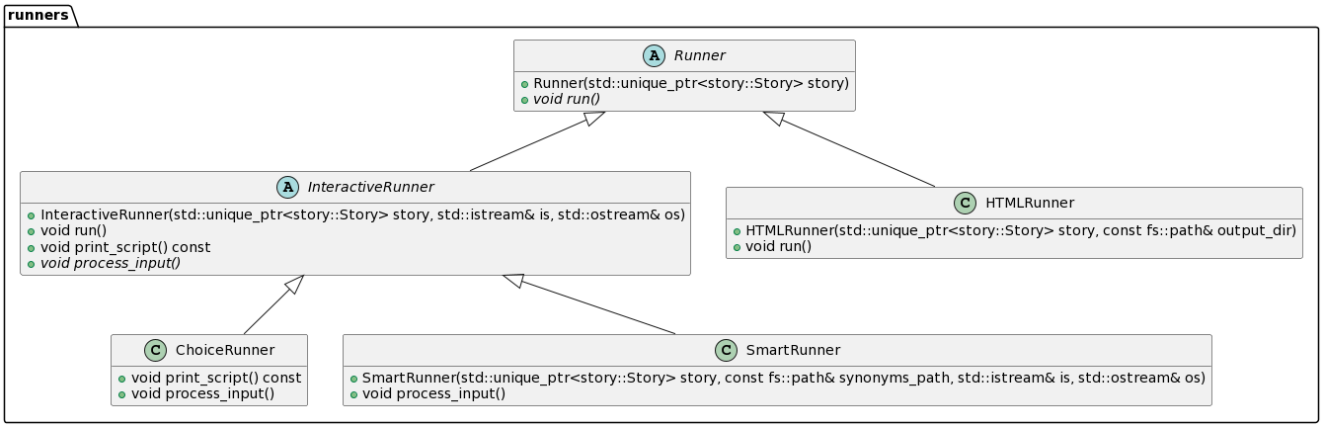

a
c
d
b

4 Runners

Tips

The provided files contain a directory `zorkxplorer/`. You can link it with your library to test it. After each feature, you can uncomment the corresponding macro in `zorkxplorer/src/main.cc`.

In this section, you use your story implementation to implement runners! Runners allow you to actually play the branching game. You will have to implement different types of runners, each with their own specificities which will be explained in their sections. Here is an UML diagram describing the class hierarchy for the runners:



The `HTMLRunner` class is described in the 'Advanced runner' validation path. It is present in this diagram to help you understand the benefits of having a base `Runner` class. All runners have a common interface that can be used to write generic code as in `zorkxplorer`.

All runners are derived from the base class `Runner` defined in `include/libzork/runner/runner.hh` that has a pure virtual member function `run`. Implement its constructor:

```
Runner(std::unique_ptr<story::Story> story);
```

Tips

This constructor should do nothing but initialize a **protected** data member that will be accessed by subclasses.

Be careful!

The copy constructor and copy assignment operator of `std::unique_ptr` are deleted.

4.1 Interactive runner

An interactive runner allows interactions with the user. It works roughly as a REPL[1]. For each node:

1. The text associated to the node is printed on an output stream (*print*)
2. A user input is requested (*read*)
3. The user input is parsed to chose the next node (*eval*)

The file `include/libzork/runner/interactive.hh` defines the abstract class `InteractiveRunner`. This is the base class for interactive runners.

Implement the constructor for the class `InteractiveRunner`, defined in `include/libzork/runner/interactive.hh`:

```
InteractiveRunner(std::unique_ptr<story::Story> story,
                  std::istream& is = std::cin,
                  std::ostream& os = std::cout);
```

It takes an input and an output stream, used to interact with the user during the execution of the `run` function. By default, the streams used are the standard input and output.

Implement the protected member function `print_script` which displays the script of the current story node followed be a newline on an output stream.

```
virtual void print_script() const;
```

The function `process_input` is responsible for steps 2 and 3: it reads from the runner's input stream and applies the choice to the story. If the choice is invalid, it raises an exception containing the error message to be printed.

```
virtual void process_input() = 0;
```

Tips

`InteractiveRunner::process_input` is a pure virtual function. This means that you don't have to implement it.

The thrown exceptions **must** be derived from `std::runtime_error`. This behavior will be tested.

Then, override the function `run` from the base class `Runner`:

```
void run() override;
```

You must call `print_script` and `process_input` in this function with the previously given input and output streams. *Before* calling `process_input`, write '> ' on the output stream (*without* a newline) to let the user know that an input is expected. Whenever `process_input` throws an error message, display it and call the function again. `run` returns when a node with an empty list of choices is reached. After the loop, do not forget to call `print_script` one last time.

[1] Read-eval-print loop

4.1.1 Choice Runner

The class `ChoiceRunner` in `include/libzork/runner/choice.hh` represents a runner that presents the user with all available choices and asks them to chose between them.

Going further...

Notice the line `'using InteractiveRunner::InteractiveRunner'`. This using-declaration makes the constructor from the base class visible in the derived class.

The choices must be printed in the order they are defined in the YAML file and associated to a number starting from 1. If the conditions for a node to be chosen are not met (see *next section*), the choice is not even shown.

Consider the following example node:

```yaml
...
- name: forest
  script: forest.txt
  choices:
    - text: Go deeper into the forest
      target: deep-forest
    - text: Climb the tree
      target: tree
      conditions:
        - name: inventory
          comparison: lower
          value: 10
    - text: Go to clearing
      target: clearing
...
```

where `forest.txt` contains the following text[2], where `$` represents a newline:

```
This is a path winding through a dimly lit forest. The path heads north-south
here. One particularly large tree with some low branches stands at the edge of
the path. A nearby path leads into a clearing to the west.$
```

If we **ignore variables**, the expected output from `print_string` is:

```
This is a path winding through a dimly lit forest. The path heads north-south
here. One particularly large tree with some low branches stands at the edge of
the path. A nearby path leads into a clearing to the west.$
$
1. Go deeper into the forest$
2. Climb the tree$
3. Go to the clearing$
$
```

The script is printed, followed by a newline and the list of choices. In this example, we ignore variables so all three choices are printed. Notice the extra newline after the list of choices.

[2] This example is directly inspired from Zork

Going further...

After implementing the variables and assuming that the value for the variable `inventory` is strictly greater than 10, `print_string` must have the following output:

```
This is a path winding through a dimly lit forest. The path heads north-south
here. One particularly large tree with some low branches stands at the edge of
the path. A nearby path leads into a clearing to the west.$
$
1. Go deeper into the forest$
2. Go to the clearing$
$
```

If the value of `inventory` had been lower than 10, all three choices would have been displayed.

Override `print_string` to add the choices. You **must** reuse the function from the base class ; this behaviour *will* be tested.

Then, override `process_input` to let the user chose their fate:

```
void process_input() override;
```

The expected behavior of `process_input` is specified above. In the case of `ChoiceRunner`, the input is a integer between 1 and n, where n is the number of available choices.

Tips

You can uncomment the line `#define CHOICE_RUNNER` in `zorkxplorer/src/main.cc` and run the executable like this:

```
42sh$ ./zorkxplorer --story path/to/story.yml
```

Example

Here is an example run using the `ChoiceRunner`[3]:

```
Hello and welcome to this amazing game. You see a village in the distance.

1. Go to the village
2. Stay here

> 1
You are in what appears to be a fair. Despite the hustle and bustle, you
notice an old man who has been staring at you for a while

1. Go talk to him
2. Run away

> 3
Please input an integer between 1 and 2
>
```

[3] In this example, script files end with a newline character

4.1.2 Smart Runner

The goal of the smart runner is to let the user type their choice without knowing the available options. To do this, you compute a sentence similarity between the user input and the available options.

The first step is to convert the input string to lowercase. Then, split the sentence in words. Use the following separators `" ,;!?'-.:"`. The surrounding quotes are not included in the list and the first character is a whitespace. Finally, remove duplicate words.

For example, for the following sentence:

```
To be, or not to be: that is the question.
```

The preprocessing produces the set of words `{to, be, or, not, that, is, question}`. This step is called *tokenization*.

Then, you have to compare the tokens in the user input with the choices using the provided synonyms dataset. The dataset is a YAML file with the following format:

```yaml
- word: word1
  synonyms: [synonym1, synonym2, ...]
- word: word2
  synonyms: [...]
...
```

This files provides a list of synonyms for thousands of words in the english language. Using this information, you can count the number of synonyms of words entered by the user present in each of the available choices. Words that are not in the dataset are ignored. The choice with the most synonyms is then selected. If no choice has synonyms in common with the user input, display an error message and wait for another input.

Be careful!

Notice that words are not present in their own list of synonyms. However, a word is a synonym of itself.

For example, assume that the following choices are available :

1. `Attack the ogre`
2. `Go around the ogre`
3. `Go back to a safe place`

The sentence `Assault the monster!` has two synonyms in common with the first choice (`assault` $\leftrightarrow$ `attack`, `ogre` $\leftrightarrow$ `monster`), only one with the second and none with the last one. The first one is therefore the closest in meaning.

Tips

You can assume that the 'is a synonym of' relation is symmetric, meaning that if a word w_1 is a synonym of w_2, then w_2 is a synonym of w_1. Stricly speaking, this is not the case for all words in the dataset but those cases will not be tested.

Going further...

The given dataset is a subset of the synonyms sets present in Wordnet. Wordnet also has other relations between words such as hyponymy (`X is a kind of Y`). There are better methods based in this relation to find word similarity but this is good enough for our case.

Implement the constructor class `SmartRunner` in `include/libzork/runner/runner.hh`:

```
SmartRunner(std::unique_ptr<story::Story> story,
            const fs::path& synonyms_path, std::istream& is = std::cin,
            std::ostream& os = std::cout);
```

The constructor is responsible for parsing the dataset and storing the synonyms in an appropriate container.

Then, implement the private function `count_synonyms`, which returns the number of synonyms in common between two sentences:

```
int count_synonyms(const std::string& left, const std::string& right) const;
```

Finally, implement `process_input`:

```
void process_input() override;
```

Tips

You can uncomment the line `#define SMART_RUNNER` in `zorkxplorer/src/main.cc` and run the executable like this:

```
42sh$ ./zorkxplorer --story path/to/story.yml --smart path/to/synonyms.yml
```

Example

Here is an example run using the `SmartRunner`:

```
Hello and welcome to this amazing game. You see a village in the distance.

> visit the village
You are in what appears to be a fair. Despite the hustle and bustle, you
notice an old man who has been staring at you for a while.

> call my mom
I beg your pardon?
> escape
Terrified, you flee without looking back. As you stop to catch your breath,
you notice that you are out of the village.

>
```

5 Variables

Now that you have a working implementation for static stories, it's time to add variables!

5.1 Storing variables

Variable values are part of the dynamic data, so they must also be stored in the `Store` class.

Implement the following accessors and mutators:

```
virtual bool has_variable(const std::string& name) const;
virtual int get_variable(const std::string& name) const;
virtual void set_variable(const std::string& name, int value);
```

`get_variable` throws a custom exception if the variable isn't defined. This exception must be derived from `std::runtime_error` ; this behavior will be tested.

You now have a way to store and access variable values, it's time to implement conditions and actions. The directory `include/libzork/variables` contains header files for classes representing them.

5.2 Conditions

The class `Condition` in `condition.hh` represents a comparison between a variable and an integer. It has a nested enum class `Condition::Type` containing all comparison types.

Implement its constructor:

```
Condition(const Store& store, const std::string& variable,
          const std::string& comparison, int value);
```

where:

- `store` is an instance of `Store` in which to lookup the variable
- `variable` is the name of the variable
- `comparison` is a lowercase string representing the comparison type. If the string is not valid, you must raise an exception
- `value` is the integer to compare the variable to

Then, implement the member function `apply` that compares the variable from the store to the integer and returns a boolean:

```
bool apply() const;
```

5.3 Actions

The class `Action` in `action.hh` is similar to `Condition`. It also has a nested enum class `Action::Type` containing all action types.

The constructor of the class `Action` resembles the one of `Condition`:

```
Action(Store& store, const std::string& variable,
       const std::string& action, int value);
```

You must also implement the member function `apply` that applies the action to the variable:

```
void apply();
```

This function doesn't return a value but must modify the value of the variable.

5.4 Implementing variables

Now that you have a way to easily represent conditions and actions, you are able to parse variables from the YAML file. Do not hesitate to read *Section 2.2* again.

The first step should be to change the constructor of the class `Story`. Use the values in the node `variables` to set the initial values for the variables.

For each choice, if a node `conditions` or `actions` isn't defined, it defaults to an empty list. Otherwise, you must parse them according to the format defined above. Rewrite `story::Node::add_choice` to store conditions and actions properly.

Do not forget to update `story::Node::list_choices` to filter out choices whose associated conditions are not met.

Finally, when calling `story::Node::make_choice`, filter out the choices whose associated condition is false while keeping the initial order. You must also apply all associated actions to the branch.

6 'Advanced runner' path

This section present one of the two possible validation paths. You will have to extend the features of the smart runner and add a new story runner that can be played in the web browser.

6.1 Builtin commands

You will now have to add builtin commands in the **smart runner**. Builtin commands are case insensitive and evaluated *only* if no match has been found.

You must handle the following commands in `process_input`:

Command	Description
`quit`	Exit the game. You **must** exit through an exception derived from `std::exception` but **not** `std::runtime_error`.
`brief`	Print the message of the active node by calling `print_script`.
`jump`	Print a funny error message.[1]
`shout`	Print '`Aaaarrrrgggghhhh!`' followed by a newline.

Be careful!

Builtin commands are handled in `process_input` and all messages must be printed on the output stream passed to the `SmartRunner`.

After a builtin command has been called, `process_input` is called again. Do not forget to print > before calling `process_input`.

Here is an example of a game using those commands:

```
Greetings and salutations esteemed newcomer, it is with great delight and
fervor that we extend to you a most cordial welcome to this illustrious game.

> jump
Do you expect me to applaud?
> shout
Aaaarrrrgggghhhh!
> brief
Greetings and salutations esteemed newcomer, it is with great delight and
fervor that we extend to you a most cordial welcome to this illustrious game.

> quit
```

If the user types a valid builtin command but one of the available choices matches, the choice takes precedence.

[1] The original zork game prints messages such as 'whee!', 'do you expect me to applaud?', 'are you proud of yourself?'.

6.2 Undo/redo

You will now have to implement a way to undo or redo a transition. The `undo` operation simply consists in restoring a previous state of the game. The `redo` operation gives a way to revert an `undo`. It can only be performed if:

1. It has a corresponding `undo`.
2. No new choices have been made in the game after the associated `undo`.

Be careful!

For the undo and redo operations, transitions must be seen as **atomic**. This means that when undoing the transition from node A to B, you must also undo all changes that occured in the variables. A good way to do this is to save the state each time the current node changes.

Implement the two **public** functions `undo` and `redo` in the class `Store`:

```
bool undo();
bool redo();
```

The two functions return `true` if the operation succeeds, else `false`.

Tips

You are advised to create a **private** class `Store::State` containing the variables and the current node. This will allow you to easily stack states.

Now, implement the two builtins `undo` and `redo` in the **smart runner**. If the operation is impossible, print a message on the output stream. If the operation is possible, execute it and do **not** call `process_input` as you would do with the other builtins but instead print the text of the new node.

Here is an example of how those two commands are used:

```
Welcome the the game! You see evil goblins in front of you and a red door on your left.
> go forward
You can attack the goblins or step back.
> attack the goblins
You aren't strong enough. The goblins are about to beat you but you can't escape anymore.
> undo
You can attack the goblins or step back.
> undo
Welcome to the game! You see evil goblins in front of you and a red door on your left.
> undo
Nothing to undo.
> redo
You can attack the goblins or step back.
> undo
Welcome to the game! You see evil goblins in front of you and a red door on your left.
> open the door
The door opens onto a large flower field.
> redo
Nothing to redo.
>
```

6.3 Save/Restore

In this section, you implement a way to save a user's progression in the story so that they can restore it later. To do that you will have to save the progress in a YAML file with the following format:

```
active-node: <name of the current node>
variables:
  <variable-1>: <value-1>
  ...
  <variable-n>: <value-n>
```

The name of the current node is its name as written in the YAML file describing the story. For example, if the current node is `cave` and the variables are `xp` and `health`, set to respectively 7 and 11, here is the expected file:

```
active-node: cave
variables:
  xp: 7
  health: 11
```

Implement the **public** member function `save` in the class `Store` which writes the current state of the game into an output stream:

```
void save(std::ostream& os) const;
```

If there is no active node, set it to a null value such as `null` or `~`.

Tips

The library `yaml-cpp` also offers you the possibility to easily emit YAML.

Implement the function `restore` that loads a state from a backup file into the store. If the undo/redo features is implemented, `restore` also resets the undo history.

```
void restore(std::istream& is, const story::Story& story);
```

Tips

`restore` takes a `story::Story` as a parameter since only the name of the active node is present in the backup file. You need to extract the node from the story based on its name.

Now, add the two builtin commands `save` and `restore` in the **smart runner**. Those builtins are parametrized and expect the path to a backup file as an parameter:

```
> save /path/to/backup.yml
> restore /path/to/backup.yml
```

Note that the path can contain any character except a `NUL` (`\0`) byte. The case where the path given to `restore` doesn't exist or points to an ill-formated file won't be tested and you can assume that the path given to `save` is writeable.

Similarily to `undo` and `redo`, the `restore` command does **not** call `process_input`. Here is an example of how `save` and `restore` can be used:

`Session 1:`

```
You are in front of a set of two open doors

> take the left door
You are now in the meeting room

> save backup.yml
> quit
```

Session 2:

```
You are in front of a set of two open doors

> restore backup.yml
You are now in the meeting room

>
```

6.4 Static runner

The goal of this part is to be able to follow a story in a web browser. For each node, an HTML file is generated as follows:

```
<html>
  <body>
  <p>[Node script]</p>
  <ol>
    <li><a href="[target1.html]">[First choice]</a></li>
    <li><a href="[target2.html]">[Second choice]</a></li>
  </ol>
  </body>
</html>
```

The whitespaces and newlines do not matter, only the elements and their contents are tested. Here is an explanation of each required tag:

- `<html>...</html>` is the root tag which encloses the whole document
- `<body>...</body>` contains the body of the document
- `<p>...</p>` represents a paragraph
- `<ol>...</ol>` represents an ordered list
- `<li>...</li>` represents an element of a list
- `<a>...</a>` represents a link, the attribute `href` being the resource it points to. In our case, it is the relative path of the target node file

The class `HTMLRunner` in `include/libzork/html.hh` represents such a runner. Implement its constructor:

```
HTMLRunner(std::unique_ptr<story::Story> story, const fs::path& output_dir);
```

Since it derives from `Runner`, implement the method `run` that generates the HTML files in `output_dir`:

```
void run() override;
```

If the directory doesn't exist, create it.

Tips

You can uncomment the line `#define HTML_RUNNER` in `zorkxplorer/src/main.cc` and run the executable like this:

```
42sh$ ./zorkxplorer --story path/to/story.yml --html directory/
```

Going further

Now that you have static HTML files, you may add variables using the javascript tag and the local storage. When the first page is loaded for the first time, you need to clear the local storage and initialize the variables. By adding ids to the the `<li>` tags, you may hide choices conditionally. To execute actions, add the attribute `onclick` to the `<a>` tags.

If you wish to make your pages prettier, you may also add style using CSS.

Be careful!

`HTMLRunner` won't be tested with variables so those features are bonuses. Be careful not to change the basic required behavior.

7 'Network' path

This sections presents one of the two possible validation paths. The ultimate goal is to be able to use a remote server to fetch and store all data about the game, from the inital YAML file to the current state.

You are allowed to use the library cpp-httplib to make HTTP requests. You can find documentation on how to integrate it with CMake in its CMakeLists.txt

We provide an HTTP server with all the required endpoints. You can find a description of each one in the associated sections below. You can also use the endpoint `GET /swagger` of the given server to get the full API documentation.

7.1 Persistent storage

In this section, you derive the base class `Store` to persist the game state on a remote server.

You will need to use the following endpoints:

- `GET /stories/{story}/active-node?login=<login>`

  Get the active node in the story `{story}`.

- `PUT /stories/{story}/active-node?login=<login>`

  Set the active node in the story `{story}`.

- `GET /stories/{story}/variables/{variable}?login=<login>`

  Get the value of the variable with the name `{variable}`.

- `PUT /stories/{story}/variables/{variables}?login=<login>`

  Set the value of the variable with the name `{variable}`.

The query parameter `login` is used to handle multiple users on a single server.

Tips

The full description of the API including the response codes can be found at the endpoint `GET /swagger` on the provided server.

The file `include/store/network-store.hh` declares the class `NetworkStore` that is derived from `Store`. Implemenet its constructor:

```
NetworkStore(const std::string& url, const std::string& username, const story::Story& story);
```

where:

- `url` is the URL of the server (e.g. `http://localhost:8080`).
- `username` is a string that uniquely identifies the user (e.g. `xavier.login`).
- `story` is the story the store is attached to.

Implement the following overriden functions to use the REST API:

```cpp
const story::Node* get_active_node() const override;
void set_active_node(const story::Node* node) override;

bool has_variable(const std::string& name) const override;
int get_variable(const std::string& name) const override;
void set_variable(const std::string& name, int value) override;
```

Finally, add a new constructor to the class `Story` to use a `NetworkStore` instead of a regular `Store`:

```cpp
Story(const fs::path& path, const std::string& store_url, const std::string& username);
```

Congratulations! You have implemented a persistent storage for your stories!

Tips

You can uncomment the line `#define NETWORK_STORE` in `zorkxplorer/src/main.cc` and run the executable like this:

```
42sh$ ./zorkxplorer --story path/to/story.yml \
                    --state-url http://localhost:8080 \
                    --username xavier.login
```

7.2 Network story

As you have surely already understood, the data related to the story are separated into two categories: static and dynamic data. The `Store` contains all dynamic data (the current node and the variable) while the `Story` contains static data (the description of the story). You just made it possible to handle dynamic data by using the network. It's time to do the same for static data.

You will have to use the following endpoints:

- `GET /stories/{story}`

  Get the raw content of the YAML describing the story `{story}`.

- `GET /stories/{story}/scripts/{scriptName}`

  Get the the script of the node `{node}` of the story `{story}`.

Be careful!

The last endpoint uses the name of the node and **not** the name of the script.

Implement the following constructor:

```cpp
Story(const std::string& url, const std::string& title);
```

Tips

You will need to add a constructor to the `Node` class. You **must not** write the story description in a temporary file.

This constructor fetches the YAML file describing the story from the server instead of the filesystem. `url` is the URL of a REST API exposing the two endpoints presented above and `title` the title of the

story to fetch.

Be careful!

This constructor initializes a local store and **not** a `NetworkStore`.

Since the data is not read from the filesystem anymore, some keys are *optional* in the description of the story. For example, this would be a valid response from `GET /stories/example`:

```
title: example
story:
  - name: welcome
    choices:
      - text: Explore the cave
        target: cave
      - text: Go in the forest
        target: forest
  - name: cave
    choices: []
  - name: forest
    choices: []
```

Finally, implement this constructor:

```
Story(const std::string& url, const std::string& title,
      const std::string& store_url, const std::string& username);
```

It is analogous to the constructor that you implemented in the previous part. It uses a `NetworkStore` for the dynamic data.

Tips

You can uncomment the line `#define NETWORK_STORY` in `zorkxplorer/src/main.cc` and run the executable like this:

```
42sh$ ./zorkxplorer --story-url http://localhost:8080 --title story-title
```

Be careful, you need to first add a story to the server using the following endpoints:

- `PUT /stories/{story}`
- `PUT /stories/{story}/scripts/{scriptName}`

Note that you can combine the network story with the network store.

7.3 Network runner

You now have a way to store remotely both static and dynamic data. In this final part, you will implement a runner that allows the user to chose a story among those available on a remote server. It will then behave like a smart runner and play the story normally.

For example, assume that the following stories are available on the server: `stanley-parable`, `zork-i`, `zork-ii` and `example`. Here is the expected output, where `$` represents a newline:

```
1. stanley-parable$
2. zork-i$
3. zork-ii$
4. example$
$
> 4$
Welcome to this game. Do you prefer to go forward or backward?$
> forward$
You enter a small room in which the fireplace is still burning.$
>
```

Notice that after the choice of the story has been made, it behaves **exactly** like a smart runner. The streams used are the standard input and output. In the event of an invalid input, print an error message on the standard output and retry.

To list the stories, you have to use the endpoint `GET /stories` which is exposed by the same API that is used in the previous section. The response format is a list of lines, each containing the name of a story. For the example above, the following would be returned:

```
stanley-parable$
zork-i$
zork-ii$
example
```

The file `include/libzork/runner/network.hh` declares the class `NetworkRunner`, which is derived from `Runner`. Implement its consructor:

```
NetworkRunner(const std::string& story_url, const fs::path& synonyms,
              const std::string& store_url, const std::string& username);
```

where:

- `story_url` is the URL of a server exposing routes to get static data.
- `synonyms` is the path to a YAML file containing synonyms. It is used when running the story.
- `store_url` is the URL of a server exposing routes to handle dynamic data.
- `username` is a string that uniquely identifies a user.

Then, implement the function `run`:

```
void run() override;
```

Tips

You can uncomment the line `#define NETWORK_RUNNER` in `zorkxplorer/src/main.cc` and run the executable like this:

```
42sh$ ./zorkxplorer --remote-runner http://localhost:8080 \
                    --smart path/to/synonyms.yml \
                    --state-url http://localhost:8080 \
                    --username xavier.login
```

You mean it's working? For real this time?