



# RUSH C++ — libzork

---

version #1.0



# Copyright

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

Copyright © 2023-2024 Assistants [<assistants@tickets.assistants.epita.fr>](mailto:assistants@tickets.assistants.epita.fr)

**The use of this document must abide by the following rules:**

- ▷ You downloaded it from the assistants' intranet.\*
- ▷ This document is strictly personal and must **not** be passed onto some-one else.
- ▷ Non-compliance with these rules can lead to severe sanctions.

## Contents

<b>1</b>	<b>Project overview</b>	<b>5</b>
<b>2</b>	<b>Project structure</b>	<b>5</b>
<b>3</b>	<b>Let's write a story</b>	<b>7</b>
3.1	Basic representation . . . . .	7
3.2	Variables . . . . .	8
<b>4</b>	<b>Story implementation</b>	<b>10</b>
4.1	Graph representation . . . . .	10
4.2	Node class . . . . .	11
4.3	Store class . . . . .	12
4.4	Story class . . . . .	12
<b>5</b>	<b>Runners</b>	<b>14</b>
5.1	Interactive runner . . . . .	15
5.1.1	Choice Runner . . . . .	16
5.1.2	Smart Runner . . . . .	18
<b>6</b>	<b>Variables</b>	<b>21</b>
6.1	Storing variables . . . . .	21
6.2	Conditions . . . . .	21
6.3	Actions . . . . .	22
6.4	Implementing variables . . . . .	22
<b>7</b>	<b>Advanced runner</b>	<b>23</b>
7.1	Builtin commands . . . . .	23
7.2	Undo/redo . . . . .	24
7.3	Save/Restore . . . . .	25
7.4	HTML runner . . . . .	27

\*<https://intra.forge.epita.fr>

## File Tree

```
libzork/
├── CMakeLists.txt  (to submit)
├── include/
│   └── libzork/
│       ├── exceptions.hh
│       ├── runner/
│       │   ├── choice.hh
│       │   ├── html.hh
│       │   ├── interactive.hh
│       │   ├── runner.hh
│       │   └── smart.hh
│       ├── store/
│       │   └── store.hh
│       ├── story/
│       │   ├── node.hh
│       │   └── story.hh
│       └── vars/
│           ├── action.hh
│           └── condition.hh
├── src/
│   └── **/
│       └── *  (to submit)
└── tests/
    └── **/
        └── *  (to submit)
```

## 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;<sup>1</sup>
- files with inappropriate privileges;
- forbidden files: `*~`, `*.swp`, `*.o`, `*.a`, `*.so`, `*.class`, `*.log`, `*.core`, etc.;
- a file tree that does not follow our specifications.

---

<sup>1</sup>If an executable file is required, please provide its sources **only**. We will compile it ourselves.

# 1 Project overview

Libzork is a project that will allow you to practice the C++ knowledge that you gathered 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 to include the reader as the story's main character.

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

You will have to compile with these flags `-Wall -Wextra -Werror -pedantic -std=c++20 -Wold-style-cast`

# 2 Project structure

## Be careful!

Be careful, you **must** have a root `libzork/` directory.

The `include/` directory contains all exported header files. You **must not** edit those header files as they represent the public API that should stay the same no matter what and will be overwritten by the moulinette.

The `src/` directory contains all private headers and all sources (`*.cc`). You are free to modify anything in this directory as long as you provide correct behaviour from the public API.

We use a special kind of private implementation for the classes to write in this project. Instead of the usual class declaration in the header file and definition in the source file, we use a public header with an interface (a class with only pure virtual methods) and a private header with the implementation of this class. This pattern enables us to hide the implementation details from the user and to provide a stable public API.

For example the `libzork::story::Node` interface is declared in the public header `include/libzork/story/node.hh` and implemented in the private header `src/story/node_impl.hh` in the class `libzork::story::NodeImpl`. In addition to the class declaration, the public header also contains the factory method `libzork::story::make_node` that creates a unique pointer to a node. This function is the only way to create a `Node` object from the public API.

You **must not** modify the *public* headers and you **should** add any method or attribute you need in the *private* headers. You can also add and remove *private* headers and sources as you please.

You **may** also assume that no other implementation than yours will be used in the tests. Thus a valid `Node*` is always castable to a `NodeImpl*` for example.

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

### Be careful!

For this project, the target name and the library name are the same. Normally, creating a target `libzork` will build the library `liblibzork.so`. We advise you to look at the documentation of [set\\_target\\_properties](#) and the target property **OUTPUT\_NAME** so that your target `libzork` builds the library `libzork.so`.

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.
- The advanced runner in which you write a new story runner and add builtin commands.

### Tips

You can put your tests in the `test/` directory. This directory does not need to be submitted, but you're allowed to.

## 3 Let's write a story

### 3.1 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:

```
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: []
```

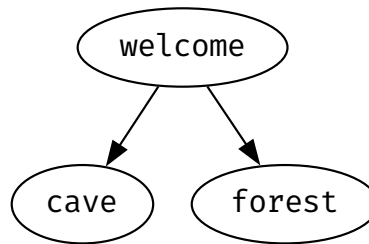
Let us look at this file step by step.

- `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. Here is an example of a possible file structure:

```
.
+-- scripts
|   +-- 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.

### 3.2 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:

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

In this story, the player starts with 0 experience points and 100 health points. 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` (`>=`)
  - `lower_equal` (`<=`)

A choice is available only if it has no conditions or 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 subtracts `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
[...]
```

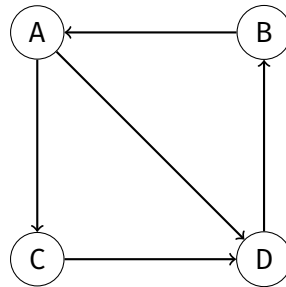
In this example, players can attack the goblins only if they have at least 20 health points. If they chose to do so, they lose 10 health points but earn 5 experience points.

## 4 Story implementation

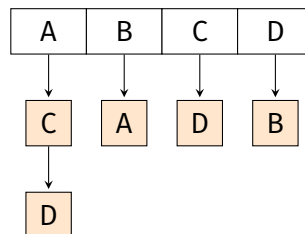
In this section, you will implement the story graph. This will be essential for the next steps so make sure to **thoroughly** test your code.

### 4.1 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:

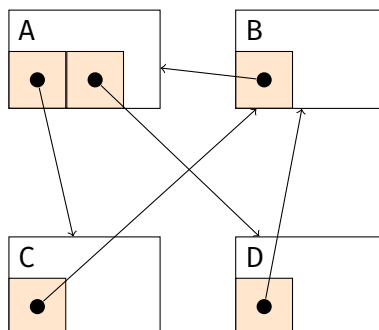


An easy way to implement an adjacency list is to assign a number to each of the vertices. The adjacency list is then an array indexed by vertices in which each cell is a linked list of the successors. For the above graph, the adjacency list could look like this (we use letters instead of numbers for 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 outgoing 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 additional information, beyond the end vertex. It is now time for you to implement this.

## 4.2 Node class

The file `include/libzork/story/node.hh` contains a declaration of the `Node` interface. You **must** implement it in `src/`. A `NodeImpl` class is provided in `src/story/node_impl.hh` for this purpose.

### 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 factory function is declared as:

```
std::unique_ptr<Node> make_node(const std::string& name, const fs::path& script_path);
```

`script_path` is a path to the file containing the text of the node. The behavior when `script_path` is an invalid path or its content has an invalid format will not be tested. However, we advise you to handle these cases for debugging purposes. At this point you should already have **private** data members.

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

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

Then, implement `make_choice` that 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(bool 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,
               std::vector<std::unique_ptr<vars::Condition>> conditions = {},
               std::vector<std::unique_ptr<vars::Action>> operations = {}) = 0;
```

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!`.

```
auto hello = libzork::story::make_node("hello_node", "hello.txt");
auto world = libzork::story::make_node("world_node", "world.txt");

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

const auto also_world = hello->make_choice(1);
std::cout << hello->get_name() << ": " << hello->get_text() << "\n"
          << 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>
```

### 4.3 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:

```
const story::Node* get_active_node() const;
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` must return `nullptr`.

### 4.4 Story class

The `story` class is responsible for reading the YAML file and creating the graph. You can find its interface 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` factory function takes a path to a specification file as parameter and parses it:

```
std::unique_ptr<Story> make_story(const fs::path& path);
```

The behavior when *path* is an invalid path or when its content has an invalid format will not be tested. However, we advise you to handle these cases for debugging purposes. Once again, you must extend this interface in your own class. You **should** have at least a list of nodes and a store as your attributes.

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);
const store::Store* get_store() const;
```

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

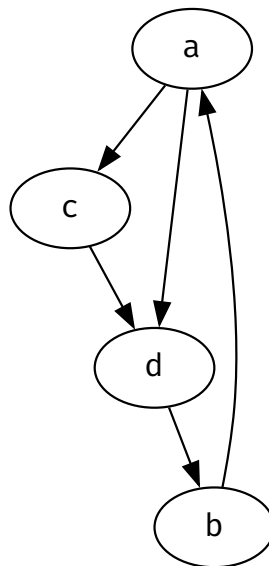
Finally, implement the `display` method:

```
std::ostream& display(std::ostream& os) const;
```

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`<sup>1</sup>. Here is the output of `dot -Tpng out.dot`:



---

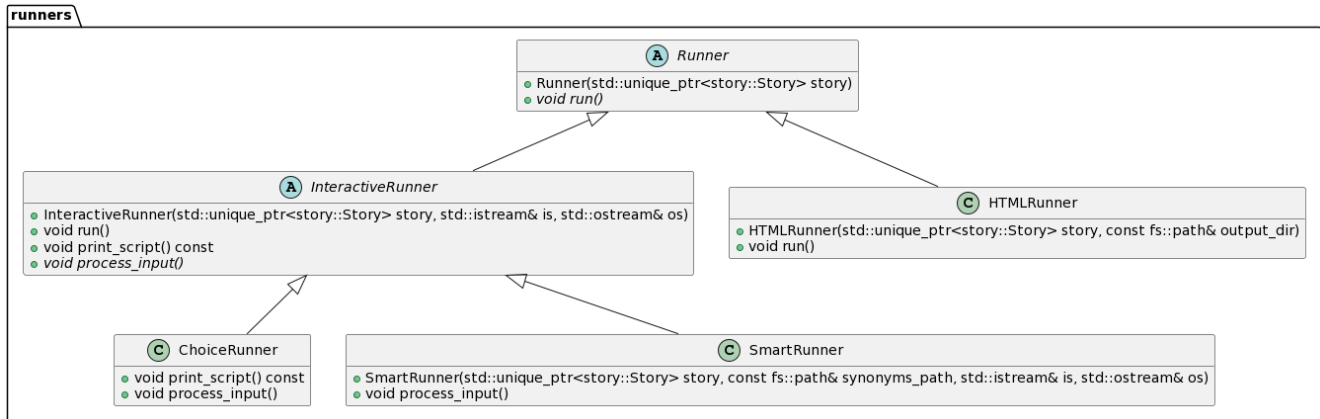
<sup>1</sup> We use `dot -Tplain` to test this part

## 5 Runners

### Tips

The provided files contain a directory `zorkexplorer/`. You can link it with your library to test it.

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 `zorkexplorer`.

All runners are derived from the base abstract class `Runner` defined in `include/libzork/runner/runner.hh` that has a pure virtual member function `run`. This class and later `InteractiveRunner` do not follow the private implementation idiom as they are just collections of utility functions (and thus not pure). 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.

## 5.1 Interactive runner

An interactive runner allows interactions with the user. It works roughly as a REPL<sup>1</sup>. 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 choose the next node (*eval*)

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

The constructor for the class `InteractiveRunner` is defined in `include/libzork/runner/interactive.hh`.

### Be careful!

It is already implemented in the provided files. You do **NOT** have to implement it, nor modify it. The reason behind this is that since references cannot be null, and they have no default constructor, they need to be explicitly instantiated in the constructor. Hence, if the constructor implementation had not been given, your code would not have compiled when implementing the basics section.

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

This constructor 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 by a newline on an output stream.

```
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 have to implement it in its derived classes.

The thrown exceptions **must** be a `RunnerInterrupt` and have a non-empty error message. This behavior will be tested.

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

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

---

<sup>1</sup> Read-eval-print loop

You **should** call `print_script` and `process_input` in this function with the previously given input and output streams. *Before* calling `process_input` you **must** 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.

### 5.1.1 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 choose 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. Take a look at [Inheriting constructors](#) for further explanation.

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 should not appear.

Consider the following example node:

```
...
- 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<sup>2</sup>, 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_script` 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.\$  
\$

(continues on next page)

---

<sup>2</sup> This example is directly inspired from Zork



(continued from previous page)

```
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.

### Going further...

After implementing the variables and assuming that the value for the variable `inventory` is strictly greater than 10, `print_script` 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_script` to add the choices. You **should** reuse the function from the base class.

Then, override `process_input` to let the user choose 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 run `zorkxplorer` like this to test your implementation:

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

### Example

Here is an example run using the `ChoiceRunner`<sup>3</sup>:

```
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
```

(continues on next page)

---

<sup>3</sup> In this example, script files end with a newline character

```
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
```

```
>
```

### 5.1.2 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, the, 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:

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

This file provides a list of synonyms for thousands of words in the English language. Using this information, for each word in each choice, you can count the number of synonyms with the tokens. Words that are not in the dataset are ignored. The choice that has the most synonyms with the user input is then selected (or the first of them if there are multiple choices with the most synonyms). If no choice has synonyms in common with the 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  $w_1$ . Strictly 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 factory function of the class `SmartRunner` in `include/libzork/runner/smart.hh`:

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

`synonyms_path` is a path to the file containing the synonyms. The behavior when `synonyms_path` is an invalid path or its content has an invalid format will not be tested. However, we advise you to handle these cases for debugging purposes.

The factory 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 unique 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();
```

### Tips

You can run the smart runner in `zorkexplorer` like this:

```
42sh$ ./zorkexplorer --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.  
  
>
```

## 6 Variables

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

### 6.1 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:

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

`get_variable` throws a custom exception if the variable is not 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 is time to implement conditions and actions. The directory `include/libzork/vars/` contains header files for classes representing them.

### 6.2 Conditions

The class `Condition` in `condition.hh` represents a comparison between a variable and an integer.

Implement its factory function:

```
std::unique_ptr<Condition> make_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 for 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;
```

## 6.3 Actions

The class `Action` in `action.hh` is similar to `Condition`.

The factory function of the class `Action` resembles the one of `Condition`:

```
std::unique_ptr<Action> make_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 returns nothing but it must modify the value of the variable.

## 6.4 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 the *section on variables* again.

The first step should be to change the factory function 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` is not defined, their default value is 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.

## 7 Advanced runner

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.

### 7.1 Builtin commands

You will now have to add builtin commands in the **smart runner**. Builtin commands are case insensitive and evaluated *only* if the input string starts with a dot.

You must handle the following commands in `process_input`:

Command	Description
<code>.quit</code>	Exit the game. You <b>must</b> exit through a <code>RunnerQuit</code> exception.
<code>.brief</code>	Do nothing, let the <code>run</code> method print the usual message.
<code>.jump</code>	Throws a <code>RunnerInterrupt</code> exception with a funny error message. <sup>1</sup>
<code>.shout</code>	Throws a <code>RunnerInterrupt</code> exception with the following message: <code>'Aaaarrrrrgggghhh!'</code> .

You may ask yourself why you have to throw a `RunnerInterrupt` for `.jump` and `.shout`. If you remember the step 3 of the REPL described in the *Interactive runner* section<Interactive runner>, an input of the user that does not choose the next node is considered as an invalid choice. Therefore, those builtins **must** be handled accordingly. In the implementation, the builtin `.brief` is considered as a valid choice that stays on the current node.

Feel free to add any other builtin command you want.

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
Aaaarrrrrgggghhh!
> .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.

---

<sup>1</sup> The original zork game prints messages such as 'wheel', 'do you expect me to applaud?', 'are you proud of yourself?'.

## 7.2 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 state 1 to 2, you must undo the modification of the node but also undo all changes that occurred in the variables. A good way to do this is to save the state each time the current state changes.

Implement the two functions `undo` and `redo` in your implementation of `Store` by inheriting from the class `UndoStore` in addition to `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 `StoreImpl : State` containing the variables and the current node. This will allow you to easily stack states.

Implement the `new_state` function that **must** set up a new state. It means that if the current state is state 1, the current state after calling `new_state()` would be state 2. `new_state`.

```
void new_state();
```

Now, implement the two builtins `.undo` and `.redo` in the **smart runner**. If the operation is impossible, throw a `RunnerInterrupt` exception with a descriptive error message.

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
```

(continues on next page)



```

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.
>

```

### 7.3 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 so, 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 respectively to 7 and 11, here is the expected file:

```

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

```

Implement the member function `save` in your implementation of `Store` by inheriting from the class `SaveStore` in addition to `Store`. It shall write 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 parameterized and expect the path to a backup file as an parameter:

```
> .save /path/to/backup.yml
Saved the game to /path/to/backup.yml
> .restore /path/to/backup.yml
Hello world!
>
```

The case where the path given to `restore` points to an ill-formatted file will not be tested. If you cannot open the file however, throw an exception derived from `RunnerInterrupt` with a descriptive error message.

## Be careful!

Keep in mind that the `save` builtin does not change the next node but `restore` does. See explanation in the *builtin commands*<*Builtin commands*> for a better understanding.

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
Saved the game to 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

>
```

## 7.4 HTML 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 path of the target node file. The path **must** be absolute for debugging purpose.

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

```
std::unique_ptr<HTMLRunner> make_html_runner(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();
```

If the directory does not exist, create it.

### Tips

You can run the html runner in `zorkexplorer` like this:

```
42sh$ ./zorkexplorer --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 `<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.

*Success is granted to those who seek the light and fight the shadows.*