Zoo Tycoon Design

Han Jiang, OSU

This design specified only some important features and design pattern I used in this project.

Animals

Apart from Class `Animal` as a base class, four Class `Camel`, `Otter`, `Monkey`, `Sloth` inherits class `Animal`.

They all share the same properties and methods as declared in Class `Animal`, except class `Monkey` has a `getBonus()` method specifically.

This inheritance implementation has a few merits:

1. all animals could be conveniently stored in a single array-like structure: like `vector<Animal*> animalList`.
2. can use prototype design pattern: `findAndClone()` is defined in `Animal class`, making it easier to instantiate an Animal of certain type.

class `Environment`

this class is designed as a singleton (meaning only a single object could be initiated in the game). As we only want a single environment to maintain time progressing.

I accomplished this by making the constructor private thus it can only be called inside the class's method. Also, a static environment pointer is defined inside the class and is initiated as null. By calling public and static method `getEnvironment()`, we can get the pointer pointed to the single instance (Environment).

```
// - Environment.h
class Environment {
private:
    Environment(){}
    static Environment* env;

public:
    static Environment* getEnvironment();
    // some irrelevant properties and methods are ignored here,a complete
version in Environment.h
};

// - Environment.cpp
Environment* Environment::getEnvironment() {
    if (env == nullptr) {
        env = new Environment();
    }
    return env;
}
Environment* Environment::env = nullptr;
// some methods are ignored here,a complete version in Environment.cpp
```

The responsibility is to maintain time progressing. This also is used in getAge() in animal class, as born time is stored in every animal, its age is computed by substraction of current time and born time.

Class `Zoo`

This class is also a singleton. As the game may contains only one player to buy animal. The signgleton implementation is the same as class `Environment`.

Five phases of a day

As specified, there are five phases in a day. In order not to share variables in between the 5 phases while the logic is still comlecated, and we may want some formatting traits in one phase, the **function type** is suitable to be as an argument:

```
void phase(function<void()> func) {
    string divider = "- - - - - - - - - - - - - - - - - - - - - -";
    cout << "\n" << divider << endl;
    func();
    cout << divider << endl;
}
```

`function` is a template-typed class, where `void()` means this function type returns void and receive no argument.

As we don't want a function to be used once after having arduously declared and defined the function. The **lambda function** is suitable, it is anonymous, and can be cast into function type.

For example:

```
// 1. day past
phase([&] () {
    Environment::getEnvironment()->oneDayPast();
    cout << "Good morning! This is a new day!" << endl;
    cout << "Day: " << Environment::getEnvironment()->getCurrent_time()
    << "\nRemain: " << player->getRemain()
    << "\nAdult amount: " << player->getAdultNum()
    << "\nBaby amount: " << player->getBabyNum()
    << endl;
});
```

Sometimes, we need to ask the user to choose from some options, we may first print a question, then the options, and read in the user's choice, if the user's choice is invalid, we might ask them agian until getting the valid input, after all this steps we can finally get the choice and do some job. This is a lot of steps and repeatedly steps of same pattern (question, option, read in, check input, do something corresponding to the input). This can be considerably abstracted as :

```
/**
 * This function makes a question-option prompts easier to implement in
order to save code.
 * Each option corresponds to a relative function given by solutions.
 * @reference stackoverflow.com/questions/15099707/how-to-get-position-of-
a-certain-element-in-strings-vector-to-use-it-as-an-inde
 * @param question the question you will give to users, eg. How many animal
do you want to buy
 * @param options the option vector in string, eg. {"0", "1", "2"}
 * @param solutions the function vector called corresponding to the option,
eg. { [](int pos) {} }
 * @return the position of the user's choice
 */
unsigned long promptor(const string& question,
                       const vector<string>& options,
                       vector <function<void(int)>> solutions) {
    cout << question + "?" << endl;
    string option;
    vector <string> characters = {"a", "b", "c", "d", "e", "f", "g", "h",
"i", "j"};
    for (int i = 0; i < options.size(); i++) {
        option += characters[i] + "." + options[i] + "\t";
    }

    cout << "\t" + option + "\nYour choice: " << endl;
    string choice;
    cin >> choice;
    unsigned long pos = find(characters.begin(), characters.end(), choice)
- characters.begin();
```

```
    while (pos >= options.size()) {
        // the entered choice is invalid
        cout << "Please enter between a to " + characters[options.size() -
1] << endl;
        cout << "\t" + option + "\nYour choice: ";
        cin >> choice;
        pos = find(characters.begin(), characters.end(), choice) -
characters.begin();
    }
    solutions[pos](pos);

    return pos;
}
```

The interesting part is also `vector <function<void(int)>> solutions`:

it is a vector of function corresponding to the vector of options. An alternate implementation would be using a map (map from the option to the solution), but this way the user might enter the solution separatedly. So I choose the formal one.