Code Review #3 (for Team Dosa)

- The “Types” enum should be converted to an enum **class** for type safety / to make the code harder to misuse. Right now, someone less familiar with the codebase could do something like “1==Types::DOUBLE” and true would be returned, even though 1 is an int and not a double. Conversion to an enum class is trivial and would prevent misuse such as this.
- “Value” and “Variable / var” seem to refer to the same thing but seem to be used interchangeably even in similar contexts (e.g. selectVariable and selectPlayerVal which both have the same return type). In addition, a variable or value could be literally anything. From what I see in this sample, I can’t figure out what a variable is actually supposed to represent. More clear naming that somewhat describes what a variable is used for would make this code easier to read.
- All the functions in the namespace human_input make use of std::cout and std::cin which only the person running the server can interact with, which isn’t what this code seems to be trying to accomplish (acknowledged in a comment). **Suggestion:** If you are using Nick’s networking code, I ***highly*** recommend looking at the **Connection** and **Message** classes (defined in chatserver.h I think) and **understanding how they are used in chatserver.cpp**. These classes are ***very*** helpful in both receiving input from users and sending prompts or updates to users from a game.
- The numerous functions in the namespace human_input all have a similar purpose -- directly taking human input. As a result, there is a lot of unnecessary repeated code (i.e. the timer, std::format, taking input and output, taking in many of the same parameters and doing similar things with them) which will not be fun to refactor later if changes need to be made (and they do – see above). A change in one function means changing the two others.
- Directly sending and receiving user input seem more like a networking responsibility that should be more loosely coupled from the game. Right now it’s pretty tightly coupled to the game and changing how prompts, choices, players, choices etc.. are implemented in the game also affects the basic sending / receiving of input (in all the namepsace human_input functions).

Code Review 3

Point 1: A weakness in the following code is the use of Stringly typed code, this is used for quick-fixes and can hurt code maintainability down the line.

```
55  void
56  GameState::updateVariables(std::string varName, Value val) {
57      std::list<Value> vals = variables.select(varName);
58      if (!vals.empty()) {
59          variables.update(varName, val);
60      } else {
61          // Weakness: avoid using strings like so, a better alternative is to use a try/catch block ---------------- Point 1
62          std::cout << "Please initialize variable first\n";
63      }
64  }
65
```

Point 2: The strength of the following code blocks is their separation of the updating of the player and the updating of the audience. As they both update separate entities. Another one related to the following code base is that it follows the principle of keeping it simple as well as keeping the states separate.

```
69  // Strength: I think you should stick to different methods since they both update separate entities, A strength to Keep It Simple
70  // and Keeping the states separate
71  // which makes the understanding of the given code better as well as modification of that particular method can easily be done without interferring
72  // with the state. ------------------------------------------------------------------------------------------ Point 2
73  void
74  GameState::updatePlayer(std::string playerName, std::string varName, Value val) {
75      if (perPlayerMap.find(playerName) != perPlayerMap.end()) {
76          GameVarList varList = perPlayerMap.at(playerName);
77          varList.update(varName, val);
78          perPlayerMap[playerName] = varList;
79      }
80  }
81
82  void
83  GameState::updateAudience(std::string audienceName, std::string varName, Value val) {
84      if (perAudienceMap.find(audienceName) != perAudienceMap.end()) {
85          GameVarList varList = perAudienceMap.at(audienceName);
86          varList.update(varName, val);
87          perAudienceMap[audienceName] = varList;
88      }
89  }
```

Point 3: Another strength that I have found is the good use of auto and STL functions. The auto type gives the flexibility of the types to be passed in making the code more modular and flexible regarding the change of the data type. On the other hand a good use of the std::this_thread::sleep_for can be observed which comes under the very convenient and quick way of using the STL function.

```
161  // strength: good use of auto and STL functions. ---------------------------------------------------------------------- Point 3
162  void Timer::setTimeout(auto function, int delay) {
163      this->clear = false;
164      std::thread t([=]() {
165          if(this->clear) return;
166          std::this_thread::sleep_for(std::chrono::milliseconds(delay));
167          if(this->clear) return;
168          function();
169      });
170      t.detach();
171  }
```

Point 4: Another design weakness in the following function is the use of for each loop where a lambda from the STL would have made the code look more concise and efficient.

```
namespace human_input{
    template<class P>
    void input_choice(const P& player, std::string_view prompt, std::unordered_map<std::string, std::string>& choices, std::string& respond, long timeout = -1)
        if (timeout > 0) {
            Timer t = Timer();
            t.setTimeout([&]() {
                auto s = std::format(prompt, player.getName());

                std::cout << s << std::endl;

                // Alternative Design: Try using lamdas from the STL that would be more clean and efficient ------------------ Point 4
                for (auto i : choices) {
                    std::cout << i->first << std::endl;
                }

                std::cout << ">";
                std::cin >> respond;
            }, timeout);
        }
```

Point 5: The following code's weakness is its initialization of timeout to a -1. The reason I say this is because it is potentially dangerous to initialize a variable in the function argument. An alternate way to achieve this goal is to initialize it in the function.

```
template<class P>
// weakness: avoid initializing timeout in the arguments bracket, this confuses the reader and can be potentially dangerous. An alternative way
// to achieve this goal is to initialize it in the function. ------------------------------------------------------------- Point 5
void input_text(const P& player, std::string_view prompt, std::string& respond, long timeout = -1){
    if (timeout > 0) {
        Timer t = Timer();
        t.setTimeout([&]() {
            auto s = std::format(prompt, player.getName());

            std::cout << s << std::endl;

            std::cout << ">";
            std::cin >> respond;
        }, timeout);
    }
}
```

GameState

- `GameState::GameState(Config &c)`: Don't know how the architecture is, but assuming GameState is just a class for rule execution, don't think GameState should have to know about the config.
- Maybe GameState is doing double duty by managing Rules and Variables? Split into separate classes?
- line 24: std::list is sort of weird. Only advantage over vector is std::list allows insertion everywhere in constant time which I dunno is useful in this case. Might as well use std::vector.
- `GameState::initVariable`: "playerName" argument here is sort of weird, if it's blank it sets a normal variable, otherwise sets a player variable. Should split into Variable/Player/Audience functions like select and update
- line 45: `map::at()` returns a reference so you can change the varlist in place:

```
auto& varList = perPlayerMap.at(playerName);
varList.append(newMap);
```

- line 61: Should throw an exception instead of printing
- It would probably be better to combine `initXXX` and `updateXXX` into one function `setXXX`, easier to use
- If you don't want to spam out `initVariable`, `initPlayerVariable`, `initAudienceVariable`, and you wanted to be fancy, you could probably do tag dispatch:

```
class GameState {
    struct VAR_NORMAL {};
    struct VAR_PLAYER {};
    struct VAR_AUDIENCE {};
};

void GameState::initVariable(GameState::VAR_NORMAL, std::string varName, Value
val) {
    // ...
}

void GameState::initVariable(GameState::VAR_PLAYER, std::string varName, Value
val) {
    // ...
}

state.initVariable(GameState::VAR_NORMAL, "greeting", "hi");
state.initVariable(GameState::VAR_AUDIENCE, "audience_variable", "hi");
```

(something like that at least)

VariableTypes

- Switch `VariableTypes::Types` to `enum class`? Looks safe to do so

- `getTypes` wouldn't work for float, long, unsigned, either. In general I don't think there is a clean way to go from a type to an enum value, you'll have to explicitly specify what type you want.

HumanInput

- `setTimeout` doesn't do what the code thinks it does, it'll run the timeout then call rather than running the function then ending it at a timeout. I don't think this timer library will work, there's no way for it to forcefully end running a function after a certain time.

What my group is planning on doing is a "request handling system". `input_choice` will generate a request for some user input with a request ID. The server will get user input and fulfill the request using the request ID. Then the interpreter knows to start running the rule again.

couts do NOT replace test code: From this code snippet only, it seems like tests are conducted via couts?
Ex. lines 59 - 62

```
for (auto &player : player_list) {
      std::cout << "Name: " << player.getName() << ", Score: "
                << player.getScore()<< std::endl;
}
```

Here, you are using a loop to check whether the list had been sorted the correct way or not. Print statements are not the best way to test your code because they aren't testing your code; you will just be looking at the terminal for some sort of sanity check. Instead, using the googletests will prove to be a better option. These tests will end up acting as documentation as well, whereas with your print statements, you will have to delete them.

Unnecessary implementations, should be using std functions. The title says it all but you are implementing a lot of loops and functions that you don't need to be implementing. For example, Lines 284 - 294:

```
template<class T> bool CustomGameList<T>::contains(T item) const
noexcept {
    for (const auto& e : data) {
        if (item == e) {
            return true;
        }
    }
    return false;
}
```

You can replace the for each loop and if statement combo with std::find or std::find_if
https://en.cppreference.com/w/cpp/algorithm/find
https://stackoverflow.com/questions/13394000/c-find-if-lambda
If you do replace your long functions with these, you can shorten your code, increase readability and minimize any errors that you might make programming it. Moreover, writing at a higher level allows you to focus at a higher level, rather than having to worry about some low level implementation.

Most of your methods in `ListRules.cpp` can and should be replaced using the std::functions.

String typed consts: For JSON parsing, it made sense to have string constants because that's the data you were presented with. So this is better than just using raw strings in your code. However, when there is no JSON parsing and you are instead interpreting and processing data, you should be using a different data type, such as enums or a struct.

Inconsistencies in naming: The code alternates between camelCase and doing this: function_name.

```
Line 38: void global_message(std::string_view value, const T& winner)
```

`Line 86: void updateValues(const T& value)`

The use of **templates** allows for more versatile and reusable code and your group has taken advantage of that so good for you.

Good formatting: your code is very readable and there are minimal indenting issues -- those that exist, I'm assuming are due to copy and pasting issues. Very consistent and easy to read code.

Confusing TODO line 215: I usually give suggestions when teams put ToDos but yours doesn't make much sense. Why would you want all the attributes to be the same? Seems like an unnecessary step.

Code Review:

1. The initVariable() is confusing to look at. The parameter name “varName” is too generic and I can’t tell what it’s referring to. Might just need some good comment clarification or better naming. It looks like it is initializing the game variables, adding it to either the player or the generic variables. Some comments and spacing work help with the readability

2. I like the GameState’s push and pop design to iterate the rules. I’m guessing a main game class will do the actual iteration. But I’m wondering if it’s a good idea to have this design within the GameState itself. Ideally, a gameState class should only be keeping track of the state (retrieving variable or updating it). Perhaps the design should use separate the rules from the gameState, and use something like an AST to visit the rules.

3. While I am not 100% sure how VariableTypes class will integrate with the rest of the code, I imagine you are trying to use multiple types and find which type you care currently using. If you are using boost::variant, there is a helpful apply_visitor function which can take in just in the parameter as different types and return the Types::someType . If not, you might just want to use the visitor pattern similarly.

4. It’s not clear why human_input functions are templated for &player. Players should be it’s own class. If you are distinguishing between a player and an audience then that seem redundant.

5. The human_input namespace has a very clean structure and readability. The spacing really helps understand the structure for each function. I liked that it was placed under a namespace instead of leaving it as a bunch of open functions. Keeps the calls organized.