# # Design Rationale

## ## Arg Parsing

I decided to create a templated cli::Option<T> class which has a constructor with the format: Option(const std::string\_view name, const T defaultVal). This Option uses a templated cli::parse<T> function to attempt to parse a T from an iterator over the arguments. cli::parse<T> is specialized for each parse-able type. This allows new options to be added with very little overhead. Due to using templates, each cli::Option<T> cannot be stored as the same type, such as in a vector, which means that each cli::Option<T> needs to be added to the loop manually.

Enabled strategies are parsed into a format resembling a union of a double and enumeration for `RNDX.XX` and all other strategies respectively. I used the top bit to store whether the rest of the `uint64\_t` stored a double or the enumeration to reduce the memory footprint.

## ## Strategies

All strategies inherit from the virtual `Strategy` class. This allows each strategy to be used interchangeably via polymorphism. Each strategy has the `getChoice` method which is called to get whether the strategy would like to cooperate or defect this round, and afterwards the `giveResult` method will be called to tell the strategy what payoff they got, and what their opponent chose.

I could have stored game state separately to each strategy and only used one function for a strategy, which would have allowed me to avoid the overhead of polymorphism and heap allocated objects by using a lambda function for each strategy. I also could have used a tagged union (such as `std::variant`) for the strategies which would have likely had less overhead than polymorphism while allowing for a similar amount of flexibility.

## ## Operator Overloads

Most classes and enumerations have an overload of the `<<` operator to work with an `std::ostream` to allow easy formatting. The `RGBColor` class can be used to set the colour for proceeding outputs.

`Choice & Choice` will result in the corresponding `Payoff`.

`!Choice` will flip the choice.

## ## Multithreaded Repeats

Each `Bracket` runs R number of `Game`s which each run N number of rounds. The `Bracket` collects the R repeats into an amount of chunks equalling the number of hardware threads, and then dispatches each chunk on separate threads. Each thread stores the completed games in a local vector, and then moves the games to the main vector upon completion to reduce the amount of time spent waiting on a locked mutex.

## ## Testing

My project has many project configurations setup with the “Command Arguments” field set appropriately. This allows for each test to be run quickly without the risk of arguments being subtly changed accidentally. The results of these tests can be written out as either a text or csv file to be analysed later, or fed into Excel to create graphs.

## ## Results

Each `Game` stores a vector of the results for the rounds between the two strategies it contains. These results are retrieved through a getter in `Tournament` to be processed after the Tournament is completed. This allows for flexibility in how the results are used (such as being repurposed for fitness in evolution) but increases the memory requirement.