Juego Solitario

Introducción

El objetivo de este laboratorio es que pongas en práctica el uso de algunas estructuras de datos de interés que nos provee el Lenguaje de Programación C++, además del uso de estructuras secuenciales, particularmente las listas enlazadas.

Para la solución de este laboratorio debes hacer revisión del manejo de tipos enumerados, especialmente los “scoped enums” (`enum class`) y el tipo `tuple` de C++. También debes hacer la revisión de la documentación del tipo `list` de la STL.

El problema que resolverás aquí es un juego de cartas solitario. Debes programar algunas funciones que resolverán pequeños subproblemas para llevar a cabo un juego. Finalmente terminarás programando un jugador automático.

El juego requiere una baraja (mazo de cartas o naipes) a la cual, por simplicidad, llamaremos “lista de cartas”; también se requiere un valor objetivo (un valor numérico). Existe también una pila de cartas, inicialmente vacía, a la cual llamaremos “cartas retenidas”[1] . Un jugador puede hacer dos tipos de jugadas: “Draw”, lo cual significa remover la primera carta de la lista de cartas y añadirla en el tope de las cartas retenidas. Y la otra jugada es “Discard”, lo cual significa eliminar una carta de las cartas retenidas. El juego tiene dos formas de terminar, una es cuando el jugador decide no hacer más jugadas y la otra es cuando la suma de los valores de las cartas retenidas es mayor que el objetivo.

La meta es terminar el juego con una baja puntuación (0 es la mejor). La puntuación del juego funciona de la siguiente manera: sea ***sum*** la sumatoria de los valores de las cartas retenidas, si ***sum*** es mayor que el objetivo, entonces la puntuación preliminar es tres veces (***sum−objetivo***) (nota que es una resta), en caso contrario, la puntuación preliminar será (***objetivo−sum***). La puntuación definitiva será la puntuación preliminar, a menos que todas las cartas retenidas sean del mismo color, en cuyo caso, la puntuación definitiva será la puntuación preliminar dividida por 2, redondeado al entero inferior, es decir, el resultado de la división entera.

La práctica

Para la realización de este trabajo, se te provee un archivo llamado `defs.hpp`, el cual contiene la definición de las abstracciones que utilizarás para solucionar el problema. No modifiques

[1] El término “retenidas” refiere a que son las cartas que mantendremos para el conteo de nuestra puntuación.

este archivo, pero estúdialo bien. Las abstracciones que se te proveen en el archivo son las siguientes:

- **`Suit`** es un enumerado que representa un palo (también conocido como pinta) de la baraja. Sus posibles valores son: `Spades` (pica), `Clubs` (trebol), `Diamonds` (diamante) y `Hearts` (corazón).
- **`Rank`** es un enumerado que representa el valor de una carta en un palo. Sus posibles valores son: `Ace`, `Two`, `Three`, `Four`, `Five`, `Six`, `Seven`, `Eight`, `Nine`, `Ten`, `Jack`, `Queen`, `King`.
- **`Color`** es un enumerado que representa el color de una carta. Sus posibles valores son `Red` y `Black`. Hay una opción `NonColor` por razones de pruebas, ninguna carta tiene ese valor de color.
- **`Card`** es la representación de una carta. Una carta es un par ordenado de tipo (`Suit`, `Rank`). Este par ordenado lo representamos con el tipo `tuple` de la biblioteca estándar de C++.
- **`card_value_t`** es un alias de **`unsigned int`** utilizado para representar los valores numéricos de las cartas.
- **`Movement`** es la representación de una jugada abstracta, está representada como una clase virtual pura (interfaz). Esta interfaz tiene 3 operacione, las cuales son:
    - `bool has_card const noexcept;` Retorna `true` si la jugada contiene una carta asociada y `false` en caso contrario.
    - const Card& get_card() const; Este método retorna (de existir) una referencia constante a la carta asociada al movimiento. Si la jugada no tiene una carta asociada, arroja una excepción.
    - `bool operator == (const Movement& m) const noexcept;` El cual retorna true si dos jugadas son iguales y false en caso contrario.
- **`Draw`** y **`Discard`** son especializaciones de `Movement`. `Draw` no tiene carta asociada, por lo que la implementación de `has_card()` retorna false, el método `get_card()` arroja una excepción de tipo `std::logic_error` si es invocado y `operator ==` siempre retorna `true` debido a que siempre son iguales. `Discard` tiene una carta asociada, por lo cual tiene un único constructor paramétrico que recibe como parámetro la carta asociada. La implementación de `has_card()` retorna `true`, el método `get_card()` retorna una referencia constante a la carta asociada y `operator ==` retorna `true` si las cartas asociadas son iguales y `false` en caso contrario.
- `MovementPtr` es un alias que se le da al tipo `std::shared ptr<Movement>`. Una lista de jugadas almacenará objetos de ese tipo para que se permita la copia a otras listas de ser necesario y se auto destruyan al liberar las listas.
- `make_draw()` y `make_discard(c)` son funciones auxiliares que nos retornan objetos de tipo `MovementPtr` con apuntadores a instancias de `Draw` y `Discard` respectivamente.

También se te provee un archivo llamado `test.cpp` el cual contiene pruebas unitarias básicas de cada una de las rutinas que se piden. Hay una única prueba de ejemplo por cada rutina. Modifica este archivo a tu gusto añadiendo más casos de prueba que cubran todos las posibilidades según las reglas establecidas para cada una de las rutinas. También se provee un `Makefile` para compilar tus pruebas.

Finalmente, se te provee el archivo denominado `solitaire.hpp` el cual contiene la plantilla de las rutinas que debes programar. El archivo contiene la implementación de las operaciones "vacías", éstas retornan valores sin sentido. Tu trabajo es programarlas para que retornen los valores correctos. Las rutinas que vas a resolver son las siguientes:

1. Calentamiento. Programa:
    a. `card_color` la cual recibe una carta y retorna su color.
    b. `card_value` la cual recibe una carta y retorna su valor numérico. Este valor numérico es el que cuenta para el cálculo de la puntuación y debe cumplir las siguientes reglas: Una carta con `Rank::Ace` vale 11 puntos, las cartas `Rank::Jack`, `Rank::Queen` y `Rank::King` valen 10 puntos y cualquier otra carta vale su propio número; por ejemplo la carta `Rank::Three` vale 3 puntos.
2. Rutinas auxiliares del juego. Programa:
    a. `remove_card` la cual recibe como parámetros una lista de cartas `cs` y una carta `c` y debe retornar una nueva lista de cartas casi igual a `cs` pero que no contenga la carta `c`. El orden de la lista no debe ser alterado. En caso de que la carta `c` no esté en `cs`, se debe arrojar `IllegalMovement`.
    b. `all_same_color` la cual recibe como parámetro una lista de cartas y retorna `true` si todas las cartas de la lista tienen el mismo color y `false` en caso contrario.
    c. `sum_cards` la cual recibe una lista de cartas y retorna la sumatoria del valor numérico de todas las cartas.
    d. `score` la cual recibe como parámetros una lista de cartas y un objetivo y calcula la puntuación siguiendo las reglas descritas en la sección 1.
3. Problemas de mayor reto. Programa:
    a. `officiate` la cual recibe una lista de cartas (la baraja inicial de juego) `cs`, una lista de jugadas (las que el jugador hace en cada instante) `ms` y un objetivo. La rutina debe retornar la puntuación obtenida al finalizar el juego. Cada jugada debe ejecutarse en el orden en el cual vienen en la lista. A continuación se te da una descripción de cómo debe operar el juego:
        i. El juego comienza con una lista de cartas retenidas vacía, llamémosla hs.
        ii. El juego termina si no hay más jugadas en la lista. (El jugador decidió parar debido a la lista de jugadas vacía).
        iii. Si el jugador descarta alguna carta `c`, el juego continúa con una lista de cartas retenidas que ya no contiene `c`. Es decir, la jugada (Discard, c)

debe eliminar a `c` de `hs`. En caso de que `c` no se encuentre allí, debe arrojarse la excepción `IllegalMovement`.

iv. Si la jugada es `Draw` y `cs` está vacía, entonces el juego termina. En caso contrario, si al efectuar esta jugada, la sumatoria de los valores en `hs` excede al objetivo, el juego termina; nota que el juego termina luego de haber efectuado la jugada `Draw`. En caso contrario, el juego continúa con un `hs` más grande y un `cs` más pequeño.

NOTA IMPORTANTE: No agregues ninguna directiva de preprocesador en el archivo solitaire.hpp. Nada de #include, #define, etc.