Laboratorio 1 de Programación 3

Alejandro Mujica

- **Este laboratorio es evaluado y la calificación que obtengas cuenta para tu calificación definitiva.**
- **Fechas válidas de entrega: desde el 03/03/2018 hasta el 05/03/2018.**
- **El rango de fechas de entrega no será cambiado bajo ninguna circunstancia. Toma tus precauciones al respecto.**
- **Tu entrega consiste de un archivo llamado `solitaire.H`, Por favor, al inicio de este archivo, en comentario, pon tu número de cédula, nombre y apellido.**

1. Introducción

El objetivo de este laboratorio es que aprehendas 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 el tratamiento como clases (`enum class`) y el tipo `tuple` de C++. También debes hacer la revisión de la documentación del tipo `SLList` de la biblioteca DeSIGNAR.

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

Para jugar el juego (valga la redundancia) se requiere una baraja (mazo de cartas o naipes) a la cual, por simplicidad, llamaremos "lista de cartas"; también se requiere un 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: "Sacar", 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 "Descartar", 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

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

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.

2. Laboratorio

Para la realización de este trabajo, se te provee un archivo llamado `defs.H`, el cual contiene la definición de las abstracciones que utilizarás para solucionar el problema. **No modifiques 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.
- `CardExcept` Es otro enumerado de un único valor posible (`IllegalMove`) el cual representa un tipo para arrojar como excepción de manera personalizada sin incluir mensajes.
- `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++`.
- `Move` es la representación de una jugada abstracta, está representada como una clase virtual pura (interfaz). Esta interfaz tiene 3 operaciones virtuales puras, las cuales son:
  - `explicit operator bool()` Este es un operador de casting explícito a bool. Esto permite que un objeto de tipo `Move` pueda ser utilizado como predicado en un "statement" `if`. El resultado dice si el movimiento tiene una carta asociada (`true`) o no (`false`).
  - `card()` Este método retorna (de existir) una referencia constante a la carta asociada al movimiento.
  - `operator ==` El cual retorna `true` si dos movimientos son iguales y `false` en caso contrario.

- `Draw` y `Discard` son especializaciones de `Move`. `Draw` representa al movimiento "Sacar", este movimiento no tiene carta asociada, por lo que la implementación de `operator bool()` retorna `false`, el método `card()` arroja una excepción de tipo `std::logic_error` si es invocado y `operator ==` siempre retorna `true` debido a que siempre son iguales. `Discard` representa el movimiento "Descartar", este movimiento tiene una carta asociada, por lo cual tiene un único constructor paramétrico que recibe como parámetro la carta asociada. La implementación de `operator bool()` retorna `true`, el método `card()` retorna la referencia constante a la carta asociada y `operator ==` retorna `true` si las cartas asociadas son iguales y `false` en caso contrario.

- `MovePtr` es un alias que se le da al tipo `std::shared_ptr<Move>`. Una lista de movimientos almacenará objetos de ese tipo para que se permita la copia a otras listas de ser necesario y se autodestruyan al liberar las listas.

- `create_draw_move()` y `create_discard_move(c)` son funciones auxiliares que nos retornan objetos de tipo MovePtr con apuntadores a instancias de `Move` y `Discard` respectivamente.

También se te provee un archivo llamado `test.C` el cual contiene pruebas básicas (mediante asertos) 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.

Tienes un `Makefile` para compilar tus pruebas al cual debes ajustarle la ruta de instalación de la biblioteca DeSIGNAR y el compilador.

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

1. Calentamiento. Programa:

   a) `card_color` la cual recibe una carta y retorna su color. Esta rutina se puede instrumentar en 3 líneas cuando mucho. Una versión bastante compacta lo hace en una línea.

   b) `card_value` la cual recibe una carta y retorna su valor númerico. 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. Esta rutina puede ser instrumentada en 6 líneas.

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 la excepción `IllegalMove`. Una posible solución sale en un máximo de 3 líneas mediante el uso de una de las operaciones funcionales vistas en clase.

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. Nuevamente, esta rutina tiene una posible solución con una de las rutinas funcionales vistas en clase. Se puede resolver en 3 líneas.

c) `sum_cards` la cual recibe una lista de cartas y retorna la sumatoria del valor numérico de todas las cartas. Esta rutina se puede resolver en una sola línea con una de las operaciones funcionales vistas en clase.

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:

- El juego comienza con una lista de cartas retenidas vacía, llamémosla `hs`.
- El juego termina si no hay más jugadas en la lista. (El jugador decidió parar debido a la lista de jugadas vacía).
- 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 `IllegalMove`.
- Si el jugador "saca" (la jugada es `Draw`) y `cs` ya 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.

Esta operación se puede resolver en menos de 15 líneas.

b) `careful_player` la cual recibe como parámetros una lista de cartas (el mazo original) `cs` y un objetivo `goal` y retorna una lista de jugadas `ms` tal que al ejecutar `officiate(cs, ms, goal)`, tenga el siguiente comportamiento:

- El valor de las cartas retenidas (la sumatoria de `hs`) nunca excede el valor del objetivo.
- Si se alcanza una puntuación de 0, entonces no se deberán añadir más movimientos.

- Si el objetivo excede el valor de la sumatoria de `hs` en 10 o más, entonces la jugada debe ser `Draw`. Ten en cuenta que si `cs` queda vacío, deberías agregar una jugada `Draw` para que `officiate` termine el juego.
- Si luego de hacer una hipotética jugada `Draw`, la sumatoria de `hs` se mantuviese menor o igual que la meta, entonces se debe hacer esa jugada. En caso contrario, se descarta la carta del tope de `hs`. Nota, que este punto requiere que evalúes un cálculo a futuro, es decir, aún no has hecho la jugada y debes observar qué ocurriría en caso de hacerla.

Esta rutina se puede resolver en menos de 20 líneas.

3. Evaluación

La fecha de entrega de este laboratorio es desde el 03/03/2018 hasta el 05/03/2018.

Tienes permitido enviar tu práctica máximo una vez por día. Es decir, desde el día de inicio hasta el día final de la práctica tienes disponibles 3 intentos. Éstos no son acumulativos. Si un día no envías, perdiste ese intento. Si por ejemplo llegas al segundo día de la práctica y no enviaste nada el primer día, entonces te quedarían solamente 2 intentos disponibles.

Para evaluarte debes enviar el archivo `solitaire.H` a la dirección:

`alejandro.j.mujic4@gmail.com`

El "subject" debe ser **exactamente** el texto **"PR3-LAB-01"** sin las comillas. Si fallas con el subject entonces probablemente tu laboratorio no será evaluado, pues el mecanismo automatizado de filtrado no podrá detectar tu trabajo. No comprimas el archivo y no envíes nada adicional a éste.

El único contenido que debe aparecer en el correo es tu número de cédula y tu nombre separados por espacio como se muestra en el siguiente ejemplo:

```
V01XXXXXXX Alejandro Mujica
```

Por favor, no uses otros medios para enviar la solución ni escribas otros comentarios adicionales en el email.

Atención: si tu programa no compila, entonces el evaluador no compila. Si una de tus rutinas se cae, entonces el evaluador se cae. Si una de tus rutinas cae en un lazo infinito o demora demasiado, entonces el evaluador cae en un lazo infinito o se demora demasiado. Por esa razón, en todos estos casos será imposible darte una nota, lo que simplemente se traduce en que tienes cero en el intento en el cual ocurra una de las circunstancias mencionadas.

No envíes programas que no compilan o se caen. Hacen perder los tiempos de red, de cpu, el tuyo y el mío. Si estás al tanto de que una rutina se te cae y no la puedes corregir, entonces trata de aislar la falla y disparar una excepción cuando ésta ocurra. Si no logras implementar una rutina, entonces haz que dé un valor de retorno el cual, aunque estará incorrecto y no se te evaluará, le permitirá al evaluador proseguir con otras rutinas. De este modo no tumbarás al evaluador y eventualmente éste podría darte nota para algunos casos (o todos si tienes suerte). Si no logras aislar una falla, entonces deja la rutina tal como te fue dada, pero asegúrate de que dé un valor de retorno. De este modo, otras rutinas podrán ser evaluadas.

4. Recomendaciones

1. Lee enteramente este enunciado antes de proceder al diseño e implementación. Asegúrate de comprender bien tu diseño.

   Haz un boceto de tu estructura de datos y cómo esperas utilizarla. Por cada rutina, plantea qué es lo que vas hacer, cómo vas a resolver el problema. Esta es una situación que amerita una estrategia de diseño y desarrollo.

2. La distribución del laboratorio contiene un pequeño test. No asumas que tu implementación es correcta por el hecho de pasar el test. Construye tus casos de prueba, verifica condiciones frontera y manejo de alta escala.

3. Este es un problema en el cual los refinamientos sucesivos son aconsejables. La recomendación general es que obtengas una correcta versión operativa lo más simplemente posible. Luego, si lo prefieres, puedes optar por mejorar el rendimiento.

4. Ten cuidado con el manejo de memoria. Asegúrate de no dejar "leaks" en caso de que utilices memoria dinámica. Todo `new` que hagas debe tener su `delete` en algún lugar. valgrind **y** DDD **son buenos amigos**.

5. Usa el foro para plantear tus dudas de comprensión. Pero de ninguna manera compartas código, pues es considerado **plagio**.

6. No incluyas headers en tu archivo `solitaire.H` (algo como `# include ...`), pues puedes hacer fallar la compilación. Si requieres un header especial, el cual piensas no estaría dentro del evaluador, exprésalo en el foro para así incluirlo.