

## Ejercitación Clases

La ejercitación consiste en completar clases y resolver problemas mediante la escritura de nuevas clases. En los directorios `src` y `tests` se pueden encontrar los archivos `Truco.cpp`, `Libreta.cpp`, `test_truco.cpp` y `test_libreta.cpp` de referencia.

Se recomienda resolver los ejercicios en orden. En CLion se encuentran disponibles los siguientes `targets`:

- `truco`: compila los tests la clase Truco
- `libreta`: compila los tests de la clase Libreta
- `ejN`: compila todos los tests hasta el ejercicio N inclusive ( $1 \leq N \leq 15$ ).

Los `targets` también pueden compilarse y ejecutarse sin usar CLion. Para ello:

1. En una consola pararse en el directorio raíz del proyecto. En este debería haber un archivo `CMakeLists.txt`.
2. Ejecutar el comando `$> mkdir build`. Esto generará el directorio `build`
3. Ejecutar el comando `$> cd build`. Esto hará que `build` sea el directorio actual.
4. Ejecutar el comando `$> cmake ..`. Esto generará el archivo `Makefile`
5. Ejecutar el comando `$> make TTT` donde `TTT` es uno de los `targets` mencionados anteriormente. Esto creará un ejecutable con el nombre del target en el directorio actual.
6. Ejecutar el comando `$> ./TTT` siendo `TTT` el nombre del target utilizado anteriormente. Esto correrá el ejecutable.

Una vez que se hacen modificaciones en el programa, no es necesario volver a ejecutar `cmake`, basta con hacer `make TTT && ./TTT` dentro del directorio `build`.

## Introducción clases

### Ejercicio 1

En el archivo `src/Geometria.cpp`, completar la clase Rectángulo.

## Ejercicio 2

Escribir una clase para representar un Elipse. Entre las varias representaciones de una Elipse, vamos a elegir la que nombra dos radios,  $a$  y  $b$ .



Escribir la clase completa en el archivo `src/Geometria.cpp`. Deberá tener los siguientes métodos públicos.

```
Elipse::Elipse(uint a, uint b)
uint Elipse::r_a();
uint Elipse::r_b();
float Elipse::area();
```

Definiendo  $\pi$  como:

```
float PI = 3.14
```

## Ejercicio 3

Completar la clase Cuadrado.

## Ejercicio 4

Escribir la clase Circulo, reutilizando la clase Elipse, utilizando la siguiente interfaz:

```
Circulo::Circulo(uint radio)
uint Circulo::radio();
float Circulo::area();
```

## Imprimir en pantalla

Vamos a buscar que todas las clases puedan imprimirse en pantalla. Para ello vamos a sobrescribir el operador de desplazamiento de bits («) aplicado a `ostreams`.

Esto implica agregar la definición del operador por fuera de la clase.

```
ostream& operator<<(ostream& os, Rectangulo r) {
    os << "Rect(" << r.alto() << ", " << r.ancho() << ")";
    return os;
}
```

Esto logra que podamos imprimir un Rectángulo con el siguiente resultado:

```
Rectangulo r(10, 15)
cout << r << endl; // Rect(10, 15)
```

## Ejercicio 5

Sobrecargar el operador « para la clase `Elipse` para lograr el siguiente resultado:

```
Elipse e(5, 7)
cout << e << endl; // Elipse(5, 7)
```

## Ejercicio 6

Hacer lo mismo para las clases `Cuadrado` y `Círculo` con las siguientes visualizaciones:

```
Cuadrado cuadrado(3)
Circulo circulo(10)
cout << cuadrado << endl; // Cuad(3)
cout << circulo << endl; // Circ(10)
```

## Recordatorios

Vamos a programar un pequeño sistema para manejar recordatorios. Nos interesa que el sistema de recordatorio permita agendar recordatorios para una fecha y una hora dada considerando solo el lapso de un año.

Vamos a escribir todo el código en el archivo `src/Recordatorios.cpp`.

### Ejercicio 7

Escribir una clase fecha con las siguientes funciones en la interfaz:

```
Fecha::Fecha(int mes, int dia);  
int Fecha::mes();  
int Fecha::dia();
```

Vamos a considerar que los meses empiezan en 1 (Enero) y terminan en 12 (Diciembre). También vamos a considerar que el primer día del mes es el 1.

### Ejercicio 8

Permitir que la fecha pueda imprimirse con el operador ‘«’ con el siguiente formato:

```
Fecha f(6, 10);  
cout << f << endl; // 10/6
```

### Ejercicio 9

Completar el operador de comparación (‘==’) para que devuelva ‘true’ si y solo si dos fechas tienen mismo mes y día.

### Ejercicio 10

Considerando la función provista `uint dias_en_mes(uint mes)` para conocer la cantidad de días de un mes (ignorando años bisiestos), escribir una operación en la clase Fecha que incremente en un día la fecha.

La función debe tener la siguiente aridad:

```
void Fecha::incrementar_dia();
```

### Ejercicio 11

Escribir una clase Horario que almacene la hora y minutos de un recordatorio. Sobreescibir además los métodos de impresión (<<) y comparación por igualdad (==).

Se espera la siguiente interfaz pública para la clase Horario.

```

Horario::Horario(uint hora, uint min);
uint Horario::hora();
uint Horario::min();

```

El impresión con el siguiente formato:

```

Horario h(10, 30);
cout << h << endl; // 10:30

```

## Ejercicio 12

Sobreescribir el operador de comparación por menor < en la clase Horario para poder comparar dos Horarios.

El operador < se sobreescribe con la siguiente aridad:

```

class Horario {
public:
    ...
    bool operator<(Horario h);
};

bool Horario::operator<(Horario h) {
    // Completar
}

```

## Ejercicio 13

Escribir una clase Recordatorio que almacene un mensaje (**string**), y la Fecha y Horario del recordatorio.

El recordatorio debe poder imprimirse con el siguiente formato:

```

Recordatorio r(Fecha(10, 6), Horario(9, 45), "Cumple March");
cout << r << endl; // Cumple March @ 10/6 9:45

```

## Ejercicio 14

Componer las tres clases en el sistema de recordatorio, una agenda. La agenda debe iniciarse con la fecha del día actual. Deben poder agregarse recordatorios. Además debe poder conocerse los recordatorios para el día actual. Finalmente, se debe poder incrementar el día en el sistema.

Se debe cumplir la siguiente interfaz:

```

class Agenda {
public:
    Agenda.Fecha fecha_inicial);
    void agregar_recordatorio(Recordatorio rec);
    void incrementar_dia();
    list<Recordatorio> recordatorios_de_hoy();
    Fecha hoy();
};

```

Para conocer los recordatorios del día se espera que se sobrescriba el operador << y se impriman los recordatorios del día con el siguiente formato:

```

Agenda a(Fecha(5, 10));
a.agregar_recordatorio(Recordatorio(Fecha(5, 10), Horario(9, 0), "Clase Algo2"));
a.agregar_recordatorio(Recordatorio(Fecha(5, 10), Horario(11, 0), "Labo Algo2"));
cout << a << endl;
// 10/5
// =====
// Clase Algo2 @ 10/5 9:0
// Labo Algo2 @ 10/5 11:0

```

Notar que se imprime primero la fecha actual, luego un separador de cinco = y finalmente un recordatorio por línea. Los recordatorios del día imprimirse ordenados por hora.

Por otra parte, la clase **Agenda** devuelve los recordatorios usando la clase **list**. Esta es otra implementación del TAD Secuencia. Como tal, tiene una interfaz similar a la de **vector**. Pueden ver más detalle acá: <https://es.cppreference.com/w/cpp/container/list>

## Juego

### Ejercicio 15

Vamos a implementar una parte de un videojuego. Las funcionalidades son las siguientes:

- El juego se desarrolla en un tablero grillado cuadrado de N casilleros (configurable).
- La posición (0, 0) es la esquina superior izquierda y la (N - 1, N - 1) es la inferior derecha.
- En el tablero se encuentra un jugador en una posición inicial (configurable).

- El jugador puede realizar 4 acciones que son moverse en alguna de las cuatro direcciones (arriba, abajo, derecha e izquierda).
- El jugador no puede moverse más allá de los límites del mapa.
- El juego también cuenta el paso del tiempo en forma de turnos.
- En el estado inicial, cada movimiento del jugador implica pasar al siguiente turno.
- En cualquier momento un jugador puede tomar una poción (cómo las consigue y cuantas tiene no se resuelve en este momento). Las pociones permiten que el jugador realice más movimientos en un solo turno. Las pociones no duran para siempre, sino que luego de una determinada cantidad de turnos desde que fue ingerida pierden efecto.
- Cada poción es distinta, variando en cuantos movimientos extra da por turno y cuantos turnos dura su efecto.
- Un jugador puede tener el efecto de más de una poción simultáneamente. En esa situación, puede moverse tantos movimientos en un turno como la suma de los movimientos extra por poción.

La clase Juego debe tener la siguiente interfaz pública:

```
using Pos = pair<int, int>;

char ARRIBA = "^";
char ABAJO = "v";
char DERECHA = "<";
char IZQUIERDA = ">";

Juego::Juego(uint casilleros, Pos pos_inicial);
Pos Juego::posicion_jugador();
uint Juego::turno_actual();
void Juego::mover_jugador(char dir);
void Juego::ingerir_pocion(uint movimientos, uint turnos);
```