
Trabajo Práctico final de Programación

Orientada a Objetos en C++

Simulacion de epidemias.
Programación orientada a objetos en C++, Instituto Balseiro,
2do. cuatrimestre 2020.

Denise Stefanía Cammarota.

15 de diciembre de 2020

Enunciado del problema.

Objetivos.

El problema propuesto es una adaptación del presentado en: https://aktemur.github.io/cs534/project_epidemic.html. El objetivo principal es simular, a partir de unas reglas muy simples, la propagación de una enfermedad contagiosa (por ejemplo, causada por un virus).

En esta simulación, se considera al mundo como una grilla de $N \times M$ países, donde N es el número de filas y M es el número de columnas. Cada país tiene una cantidad determinada e ilimitada de personas. Existe un virus que infecta a la población. Cuando una persona es infectada con el virus, entra en un período de incubación tras el cual se enferma. Las personas infectadas y/o enfermas pueden contagiar a otros. Adicionalmente, las personas tienen la capacidad de moverse a países vecinos, eligiendo países sin personas enfermas. De esta manera, el virus puede ser introducido en otros países.

Reglas.

A continuación se presentan las reglas generales a seguir por la simulación:

1. Todos los países, excepto los que se encontrasen en los bordes del mundo, tienen 4 vecinos: al sur, al norte, al este y al oeste.
2. Cada paso o actualización de la simulación corresponde a un día transcurrido en el mundo.
3. Hay varios estados de salud posibles para las personas:
 - Una persona sana o , que es susceptible a ser infectada con el virus.
 - Una persona infectada con el virus. Luego de un período de incubación de 6 días, esta se transforma en una persona enferma. Durante esta incubación, puede contagiar a otros. Por esto, estas personas podrían ser clasificadas como infectious/infecciosas.
 - Una persona enferma/visiblemente infectada, que puede transmitir el virus a otros. Estas personas se mueren con una probabilidad del 25 % tras 8 días de enfermedad. A los 10 días, si han sobrevivido, se convierten en inmunes. Estas personas pueden ser clasificadas como infectious/infecciosas y además visibly infectious/visiblemente infecciosas.
 - Una persona muerta.

- Una persona inmune, quien no puede contagiarse. Tras 2 días de ser inmune, una persona vuelve a ser sana y puede volver a ser infectada por el virus.
4. Al comienzo de la simulación, hay P personas en el mundo, de las cuales un $X\%$ están infectadas con el virus. La distribución inicial de personas y de personas infectadas en el mundo es aleatoria.
 5. Las personas permanecen en un país entre 5 y 10 días. Los días de permanencia son asignados aleatoriamente.
 6. Transcurridos los días correspondientes, una persona se mueve a un país vecino sin personas enfermas/visiblemente infecciosas. De no existir tal país, no se mueve. De existir mas de un país vecino sin personas visiblemente infectadas, se cualquiera de ellos con igual probabilidad.
 7. Si una persona sana mueve a un país con personas infecciosas, se contagia con una probabilidad del 40%. Así, por mas de que haya elegido un país sin personas visiblemente infecciosas o enfermas, puede contagiarse, ya que hay personas en el periodo de incubación que pueden contagiarla.

Parámetros de la simulación.

La inicialización de los parámetros de la simulación se implementa en el `main.cpp`. Allí, se le pide al usuario que ingrese 5 parámetros: filas, columnas, cantidad de personas en el mundo, porcentaje de personas infectadas y días que quiere que pasen en el mundo.

El resto de los parámetros (como el periodo de incubación, la probabilidad de infectarse, la cantidad de días que dura la inmunidad, etc) son fijos y el usuario no puede cambiarlos.

Impresión por consola.

Luego de ingresados todos los parámetros correspondientes, transcurren los pasos en la simulación. Al finalizar un paso, se imprime el numero de día correspondiente y las estadísticas de salud relevantes para cada país, y para el mundo. Estas incluyen personas sanas, infectadas, enfermas/visiblemente infectadas, muertas e inmunes. Un ejemplo de la vista en la consola se muestra en la Figura 1.

```
Beginning of the Simulation
The Word is an rows x columns grid with a fixed number of people, a percentage of which is infected with a disease.
Please enter the following parameters:
Rows (integer number): 2
Columns (integer number): 2
Number of people (integer number): 100
Percentage of infected people (number between 0 and 1): 0.5
Days to pass in the simulation (integer number): 10
-----
Day 0
1x1: healthy: 12 immune:0 dead: 0 infected: 13 sick: 0
1x2: healthy: 10 immune:0 dead: 0 infected: 14 sick: 0
2x1: healthy: 11 immune:0 dead: 0 infected: 14 sick: 0
2x2: healthy: 17 immune:0 dead: 0 infected: 9 sick: 0
Word (sum): healthy: 50 immune:0 dead: 0 infected: 50 sick: 0
-----
```

Figura 1: Vista de la consola, ingreso de parámetros y estadísticas de salud.

Diseño e Implementación.

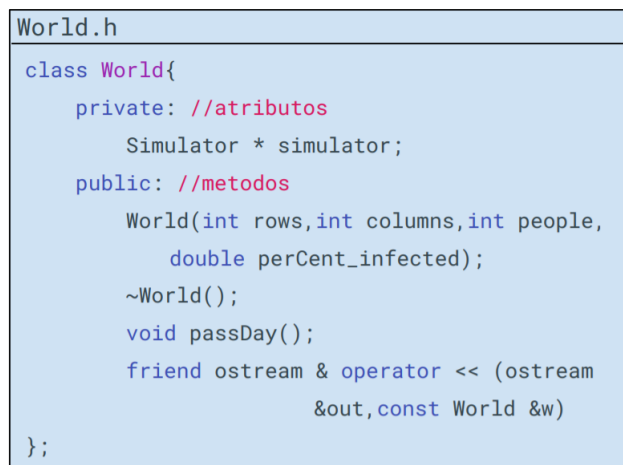
A continuación se discuten brevemente las clases implementadas para resolver el problema, al igual que sus métodos y atributos.

Se han creado en total 12 clases diferentes. Por un lado, tenemos las clases `World`, `Simulator`, `SimulationParameters`, `Human`, `Country`, `HealthState` y `HealthStats`, que no son heredadas de ninguna clase padre. Por otro lado, las clases `Healthy`, `Immune`, `Dead`, `Sick` y `Infected` son clases que heredan de `HealthState`.

Si tenemos una clase de nombre `my_class`, su definición se hace en el archivo `my_class.h` y la implementación se encuentra en el archivo `my_class.cpp`.

Clase World.

Esta clase representa el mundo en donde se lleva a cabo la simulación. En la Figura 2 se presenta un esquema de los atributos y métodos de la misma.

El diagrama muestra el código de la clase World.h en un editor de texto con un fondo azul claro. El código está escrito en C++ y define la estructura de la clase World, incluyendo atributos privados, métodos públicos, un constructor, un destructor, un método para pasar un día y un operador de salida de flujo amigable. El código es:

```
World.h
class World{
private: //atributos
    Simulator * simulator;
public: //metodos
    World(int rows,int columns,int people,
          double perCent_infected);
    ~World();
    void passDay();
    friend ostream & operator << (ostream
                                   &out,const World &w)
};
```

Figura 2: Esquema de la clase World.

Atributos.

Un objeto de esta clase tiene un puntero a `Simulator`, que se encarga de llevar a cabo la simulación propiamente dicha.

Métodos.

Los dos primeros métodos son el constructor y el destructor. El constructor de esta clase recibe como argumentos el número de filas `rows`, el número de columnas `columns`, la cantidad de personas `people` y el porcentaje de infectados `perCent_infected`. Aquí se verifican que los números recibidos sean válidos (que el número de filas y columnas sean enteros mayores a 0, y que el porcentaje de infectados sea un número entre 0 y 1). En caso de que no hayan errores, se crea `simulator` y se llaman a métodos de `Simulator` que permiten crear el estado inicial del mundo.

El método `passDay` llama a un método `passDay` de `Simulator` que permite realizar un paso de la simulación.

Se sobrecarga el operador `<<` para imprimir el estado del mundo. Esto se logra llamando al mismo operador, sobrecargado para punteros de la clase `Simulator`. Así, se puede imprimir por consola la cantidad de personas sanas, muertas, inmunes, infectadas y muertas en cada paso de la simulación.

Clase Simulator.

Esta clase representa a la simulación propiamente dicha, que se lleva a cabo dentro de un mundo determinado. Un esquema de la misma se presenta en la Figura 3

```
Simulator.h
class Simulator{
    //atributos
    vector<Country*> list_countries;
    int days_passed;
    int rows;
    int columns;
public: //metodos
    Simulator(int r,int c);
    void populate(int peop,double pcinf);
    ~Simulator();
    void passDay();
    int getDaysPassed();
    void westNeighbourAdd(Country * c, int index);
    void eastNeighbourAdd(Country * c, int index);
    void northNeighbourAdd(Country * c, int index);
    void southNeighbourAdd(Country * c, int index);
    friend ostream & operator << (ostream &out, const
                                   Simulator * sim);
};
```

Figura 3: Esquema de la clase Simulator.

Atributos.

Un objeto de esta clase tiene en primer lugar un vector de punteros a `Country`, que representan los países del mundo. También, tiene el número de días que han pasado `days_passed`, el número de columnas `columns` y el de filas `rows`.

Métodos.

Se tienen los métodos constructor y destructor como en todas las clases. El constructor recibe las filas `r` y columnas `c` como argumentos. El constructor asigna estas cantidades a los atributos `rows` y `columns`. Además, inicializa los días en 0, aloca dinámicamente punteros correspondientes a todos los países y asigna los vecinos correspondientes. Como nombre de cada país, se asigna como tal un `string` de número de fila del país x número de columna del país.

El método `populate` aloca dinámicamente los humanos, los asigna a un país e infecta al porcentaje correspondiente de la población. Tanto la designación del país como del estado de salud (sana o enferma) de una persona son asignados al azar. Finalmente, inicializa las estadísticas de salud de un país (cantidad de personas con cada estado de salud).

El método `passDay` corresponde a pasar un día en la simulación. Llama a métodos de la clase `Country` que procesan las llegadas y las salidas de un país, y actualiza las estadísticas de salud.

El método `getDaysPassed` simplemente devuelve el número de días que han pasado, es decir, el valor de la variable `days_passed`.

Los métodos `northNeighbourAdd`, `southNeighbourAdd`, `eastNeighbourAdd` y `westNeighbourAdd` son llamados por el constructor de esta clase, ya que permite asignar los vecinos a un país.

También, se sobrecarga el operador `<<` para imprimir los estados de salud de los países y del mundo en un día determinado.

Clase `SimulationParameters`.

Esta clase representa los parámetros fijos de la simulación. Un esquema del archivo `SimulatorRules.h` donde se la define se presenta en la Figura 4.

```
SimulatorRules.h
class SimulationParameters{
//atributos
private:
    int daysMaxStay = 10;
    int daysMinStay = 5;
    int daysUntilSick = 6;
    int daysUntilDeadChance = 8;
    int daysUntilImmune = 10;
    int daysUntilHealthy = 2;
    //probabilidades de cosas
    double probToTransmitVirus = 0.4;
    double probToDie = 0.25;
//metodos
public:
    SimulationParameters();
    ~SimulationParameters();
    int getMaxStayDays();
    int getMinStayDays();
    int getUntilSickDays();
    int getUntilDeadChanceDays();
    int getUntilImmuneDays();
    int getUntilHealthyDays();
    bool infectionDiceThrow();
    bool travelDiceThrow();
    bool dieDiceThrow();
};
extern SimulationParameters g_simpars;
```

Figura 4: Esquema de la clase `SimulationParameters`

Se crea una instancia `g_simpars` global, que se utiliza en todo el programa que permite

obtener todos los valores de parámetros en el programa.

Atributos.

Los atributos de esta clase corresponden a parámetros como probabilidades y días, explicados anteriormente en el enunciado.

Tenemos `daysMaxStay = 10` y `daysMinStay = 5` que son los días máximos y mínimos de permanencia en un país, respectivamente.

Por otra parte, `daysUntilSick = 6` corresponde a los días de incubación, es decir, la cantidad de días que una persona esta infectada, pero no enferma. La variable `daysUntilDeadChance = 8` es la cantidad de días hasta que una persona puede morir por la enfermedad. Además, `daysUntilImmune = 10` es la cantidad de días tras la cual una persona infectada, si no ha muerto, se transforma en inmune. Finalmente, `daysUntilHealthy = 2` es la cantidad de días que dura la inmunidad.

Las probabilidades son `probToTransmitVirus = 0.4` y `probToDie = 0.25`. La primera es la probabilidad de infectarse, al ingresar a un país con personas infectadas o enfermas. La segunda es la probabilidad de morir.

Métodos.

Tenemos el constructor y el destructor, ambos por default.

Los métodos cuyo nombre comienza con `get` devuelven los valores de los parámetros que indican en su nombre. Por ejemplo, `getMaxStayDays` devuelve el valor de `daysMaxStay`.

El método `infectionDiceThrow` devuelve `true` o `false` con una probabilidad determinada, correspondiente a si una persona efectivamente se infecta o no al llegar a un país. Lo mismo hace el método `dieDiceThrow`, que devuelve `true` o `false` correspondiendo a si una persona se muere o no tras 8 días de estar enferma.

Clase Human.

Esta clase corresponde a una persona dentro del mundo de la simulación. Un esquema se presenta en la Figura 5.

```

Human.h
class Human{
private: //atributos
    static int total_humans;
    int id_persona;
    int days_until_move;
    HealthState * health;
    Country * country;
public: //metodos
    Human(Country * c);
    ~Human();
    int get_id();
    void Gen_MoveDays();
    void Become_Healthy();
    void Become_Infected();
    void Become_Sick();
    void Become_Dead();
    void Become_Immune();
    bool isHealthy();
    bool isInfected();
    bool isSick();
    bool isImmune();
    bool isDead();
    bool isInfectious();
    bool isVisiblelyInfectious();
    Country * selectDestination();
    void moving(Country * dest_country);
    void passDay();
    Country * get_country();
};

```

Figura 5: Esquema de la clase Human.

Atributos.

Tenemos como atributos `total_humans` que cuenta la cantidad de humanos creados y `id_persona` que es el número de persona creada. Además, `days_until_move` son los días que le quedan a una persona antes de mudarse de país. El estado de salud de una persona esta dado por el puntero a `HealthState` llamado `health`. Finalmente, `country` hace referencia al país en el que esta una persona.

Métodos.

Los primeros dos métodos son el constructor y el destructor. El constructor recibe la variable `c` un puntero a `Country` que corresponde al país en el que se encuentra una persona. Procede a asignarle un id y el país `c` a la persona. Además, añade a la persona al país utilizando métodos de la clase `Country`. Genera también los días hasta mudarse de una persona, y asigna un estado de salud sano a la persona creada, allocating dinámicamente el puntero `health`.

El método `get_id` devuelve el id de una persona.

El método `Gen_MoveDays` genera los días hasta mudarse de una persona.

Los métodos que cuyos nombres comienzan con `Become_` transforman el estado de salud de una persona como su nombre lo indica. Por ejemplo, el método `Become_Immune` cambia el estado de salud de una persona y la transforma en inmune.

Por otra parte, los métodos que comienzan con el prefijo `is` y siguen con un estado de salud devuelven `true` si la persona se encuentra en ese estado y `false` en caso contrario. Por ejemplo, el método `isImmune` devuelve `true` si la persona es inmune y `false` en caso contrario. Se implementaron, además de para los cinco estados de salud posibles (sano,

enfermo, infectado, inmune y muerto), para `infectious/infeccioso` (personas enfermas e infectadas) y para `visibly infectious/visiblemente infeccioso` (personas enfermas). Esto fue así ya que en un principio se tenía la idea de incorporar personas muertas a `visibly infectious/visiblemente infeccioso` para que las personas evitaran países con muchas personas muertas, por ejemplo.

El método `selectDestination` elige el país de destino de una persona al mudarse. En principio, todos los países vecinos podrían ser posibles países de destino. Luego, como se detalló en las reglas, se filtran los países que no tienen personas visiblemente infecciosas y se elige uno de ellos para moverse al azar. Devuelve un puntero al país de destino, y, en caso de no moverse, devuelve un puntero nulo.

El método `moving` hace efectivamente el movimiento de una persona a un país de destino. Recibe un puntero a este país de destino como argumento. Primero, Cambia el argumento `country` del humano por este nuevo país, y luego llama a un método de `Country` que agrega a esta persona a las llegadas al país. Luego, si el país de destino tiene personas infecciosas, una persona se infecta con la probabilidad del 40% asignada anteriormente. Para determinar esto, se llama al método `infectionDiceThrow` de `SimulationParameters`.

El método `passDay` decrementa el número de días `days_until_move` en caso de corresponder. Si `days_until_move = 0`, entonces llama a `GenMoveDays` para generar un nuevo valor no nulo para esta variable, luego llama a `selectDestination` para generar un país de destino y, con el mismo como argumento, llama a `moving` para mudar a la persona. Llama adicionalmente a un método `passDay` de `health`, que se encarga de sumar días a la cantidad de días que una persona se encuentra en un estado de salud y actuar en consecuencia (si han pasado 8 días, por ejemplo, determinar si la persona fallece).

Finalmente, `get_country` permite devolver un puntero al país en el que se encuentra una persona.

Clase Country.

Esta clase pretende representar un país en el mundo. Un esquema de la misma se presenta en la Figura 6.

```

Country.h
class Country{
private: //atributos
    static int total_countries;
    int id_country;
    string country_name;
    HealthStats country_stats;
    vector <Country*> country_neighbours;
    vector <Human*> country_people;
    vector <Human*> country_arrivals;
public: //metodos
    Country(string s);
    ~Country();
    int get_id();
    string get_name();
    void addHuman(Human * h);
    void removeHuman(Human * h);
    void addNeighbour(Country * c);
    void moveHuman(Human * h);
    bool hasVisiblyInfectious();
    bool hasInfectious();
    vector <Country *> get_countryneighbours();
    vector <Human *> get_countryresidents();
    void UpdateHealthStats();
    int get_healthypeople();
    int get_infectedpeople();
    int get_sickpeople();
    int get_immunepeople();
    int get_deadpeople();
    int get_infectiouspeople();
    int get_visiblyinfectiouspeople();
    HealthStats get_countrystats();
    void runHealthActions();
    void processMoves();
};

```

Figura 6: Esquema de la clase Country.

Atributos.

Tenemos como atributos **total_countries** que indica la cantidad de países en el mundo creados. La variable **id_country** corresponde a un id del país, correspondiente al orden en el que fue creado. El nombre del país esta dado por la variable tipo **string** de nombre **country_name**. Adicionalmente, se tienen las estadísticas de salud de un país **healthstats**, que corresponden a una instancia de la clase **HealthStats**. Cada país tiene además dos vectores de punteros a instancias de **Country**: **country_people** y **country_arrivals** que corresponden respectivamente a las personas que residen en un país, y a las personas que llegan al mismo. Finalmente, el vector de punteros de tipo **Country** llamado **country_neighbours** donde se almacenan los países vecinos.

Métodos.

Tenemos el constructor y el destructor. El constructor recibe un **string** a asignar como nombre del país, asigna el id correspondiente al numero de país creado e inicializa las estadísticas **healthstats** de salud del mismo.

Los métodos **get_id** y **get_name** retornan respectivamente el id y el nombre de un país.

Los métodos `addHuman` y `removeHuman` reciben como argumento un puntero a *Human*, y se encargan respectivamente de añadir y remover el mismo del vector `country_people`.

El método `addCountry` recibe un puntero a un objeto `Country`, y lo añade al vector `country_neighbours`. Es decir, permite asignar vecinos a un país.

`moveHuman` recibe un puntero a *Human* y lo añade al vector `country_arrivals`, es decir, a la lista de llegadas a un país.

El método `hasInfectious` devuelve verdadero en caso de haber personas infecciosas en el país, y falso en caso contrario. El método `hasVisiblyInfectious` hace algo análogo, pero para la existencia de personas visiblemente infecciosas. Ambos llaman a otros métodos de esta clase para contabilizar la cantidad de personas en ambos estados de salud.

Los métodos `get_countryneighbours` y `get_countryresidents`, como sus nombres lo indican, retornan respectivamente el vector de países vecinos y el vector de residentes de un país.

Todos los métodos cuyo nombre tiene el prefijo `get_` seguidos por un estado de salud recorren el vector de residentes y retornan la cantidad de personas cuya salud esta en un determinado estado. Por ejemplo, `get_healthypeople` retornara la cantidad de personas sanas.

A partir de estas funciones, `UpdateHealthStats` actualiza el estado de salud de un país, llamando a métodos `set` de la instancia `healthstats` de la clase `HealthStats`.

`get_countrystats` retorna `healthstats` al ser llamado.

Los dos últimos métodos de esta clase, mencionados en secciones anteriores, son `runHealthActions` y `processMoves`. Ninguno de ellos recibe argumentos, y son llamados desde `passDay` de `Simulator`. El primero de ellos inicialmente llama a `passDay` de `Humans`. Luego de ello, determina que personas dejan el país y las remueve del vector de residentes. El segundo se encarga de procesar las llegadas, es decir, añade a la gente que esta en `country_arrivals` al vector `country_people`, y hace un clear del vector de llegadas, dejándolo listo para el próximo paso de la simulación.

Clase `HeathState` y derivadas.

La clase `HealthState` pretende representar el estado de salud de una persona. La estructura básica de la misma se presenta en la Figura 7. De ella, heredan diferentes clases que representan estados de salud propiamente dichos de una persona, como sano o inmune.

```

HealthState.h
class HealthState{
public: //metodos
    virtual void passDay(Human * h);
    virtual bool isHealthy();
    virtual bool isInfected();
    virtual bool isSick();
    virtual bool isImmune();
    virtual bool isDead();
    virtual bool isInfectious();
    virtual bool isVisiblelyInfectious();
};

```

Figura 7: Esquema de la clase HealthState.

Cuenta con varios métodos, todos de los cuales son virtuales y de los cuales se hacen los **override** correspondientes en cada una de las clases derivadas.

El primer método es **passDay** que recibe un puntero a **Human** y procesa el paso de un día en la salud de una persona como corresponda. Por defecto, no hace nada. Se hace un **override** en clases heredadas cuando el paso de los días afecte el estado de salud de una persona. Por ejemplo, el paso de los días puede hacer inmune a una persona. Sin embargo, el simple paso de los días no afecta a una persona sana, que puede contagiarse al entrar a un país con personas infecciosas.

El resto de los métodos, que comienzan con el prefijo **is** y siguen con el nombre de un estado de salud deberían devolver **true** si el estado de salud cumple con esa condición, y **false** en caso contrario. Por defecto, devuelven **false** y, para que cumplan su objetivo, se hace un **override** en las clases heredadas cuando corresponde que devuelvan **true**.

Clase Healthy.

Esta clase hereda de **HealthState** y representa a una persona sana. Un esquema de la definición de la misma se presenta en la Figura 8. Básicamente, se hace un **override** del método **isHealthy** para que devuelva **true**, al corresponder a una persona sana.

```

Healthy.h
class Healthy: public HealthState{
public:
    bool isHealthy() override;
};

```

Figura 8: Esquema de la clase Healthy heredada de HealthState.

Clase Dead.

Esta clase, también heredada de **HealthState**, representa a una persona muerta. Un esquema de su definición se presenta en la Figura 9. Únicamente, se hace **override** de **isDead** para que devuelva **true** en este caso.

```

Dead.h
class Dead: public HealthState{
public:
    bool isDead() override;
};

```

Figura 9: Esquema de la clase Dead heredada de HealthState.

Clase Infected.

Esta clase heredada de **HealthState** representa a una persona infectada, durante el periodo de incubación de la enfermedad. El esquema de esta clase se presenta en la Figura 10.

```

Infected.h
class Infected:public HealthState{
    int days_infected = 0;
public:
    bool isInfected() override;
    bool isInfectious() override;
    void passDay(Human * h) override;
};

```

Figura 10: Esquema de la clase Infected heredada de HealthState.

Tiene como único atributo **days_infected** que lleva la cuenta de los días que una persona esta infectada para actuar en base a ello. Como corresponde, se hace **override** de los métodos **isInfected** y **isInfectious** para que retornen **true**. Se hace lo mismo con **passDay**, que suma un día a **days_infected**, y enferma a una persona tras pasados los 6 días de incubacion.

Clase Sick.

Esta clase hereda de **HealthState** y representa a una persona enferma. El esquema correspondiente es el de la Figura 11.

```

Sick.h
class Sick:public HealthState{ //atributos
    int days_sick = 0;
public: //metodos
    bool isSick() override;
    bool isInfectious() override;
    bool isVisiblelyInfectious() override;
    void passDay(Human * h) override;
};

```

Figura 11: Esquema de la clase Sick heredada de HealthState.

Tiene como atributo la variable **days_sick** que lleva la cuenta de la cantidad de días que una persona lleva enferma.

Naturalmente, se debe hacer **override** de los métodos de la clase padre **isSick**, **isVisiblelyInfectious** y **isInfectious** para que devuelvan **true** en este caso. Además, se hace lo mismo con **passDay**. En este caso, este método se encarga de añadir un día a **days_sick**, determinar si una persona muere tras de estar enferma 8 días (llamando al método **dieDiceThrow** explicado anteriormente), o de convertirse en inmune tras 10 días.

Clase Immune.

Esta es la ultima clase heredada de `HealthState` y representa a una persona inmune. El esquema de su definición se visualiza en la Figura 12.

```
Immune.h
class Immune:public HealthState{ //atributos
    int days_immune = 0;
public: //metodos
    bool isImmune() override;
    void passDay(Human * h) override;
};
```

Figura 12: Esquema de la clase Immune heredada de HealthState.

Tiene como atributo la variable `days_immune` que lleva la cuenta de la cantidad de dias que una persona lleva siendo inmune a la enfermedad. Naturalmente, se debe hacer **override** del método `isImmune` para que devuelve `true`. Adicionalmente, se redefine el metodo `passDay` que añade un día a la variable `days_immune` y, tras dos días, hace que la persona vuelva a estar sana.

Clase HealthStats.

Esta ultima clase representa las estadísticas de salud de un país, es decir, el numero de personas sanas, infectadas, enfermas, muerta, inmunes, infecciosas y visiblemente infecciosas. Un esquema de su definición se presenta en la Figura 13.

```
HealthStats.h
class HealthStats{//atributos
    int healthy;
    int infected;
    int sick;
    int immune;
    int dead;
    int infectious;
    int visiblyInfectious;
public: //metodos
    int get_healthyCount();
    void set_healthyCount(int hc);
    int get_infectedCount();
    void set_infectedCount(int ic);
    int get_sickCount();
    void set_sickCount(int sc);
    int get_immuneCount();
    void set_immuneCount(int ic);
    int get_deadCount();
    void set_deadCount(int dc);
    int get_infectiousCount();
    void set_infectiousCount(int ic);
    int get_visiblyInfectiousCount();
    void set_visiblyInfectiousCount(int vic);
    friend ostream & operator << (ostream &out,
                                   const HealthStats hs);
};
```

Figura 13: Esquema de la clase HealthStats

Atributos

Los atributos son básicamente los números de personas en cada estado de salud, mas los números de personas infecciosas y visiblemente infecciosas. `healthy` es el numero de

personas sanas, `infected` de personas infectadas, `sick` de enfermas, `dead` de muertas, `immune` de inmunes, `infectious` de infecciosas y `visiblyinfetious` de visiblemente infecciosas.

Métodos

Los métodos con prefijo `get` y `set` permiten retornar el numero de personas y fijar el numero de personas en un estado, tal como sus nombres indican. Por ejemplo `get_healthyCount` retorna el valor de `healthy` y `set_healthyCount` asigna un valor que recibe como argumento a `healthy`. Estos métodos son llamados de `Country` para determinar si hay personas en un estado de salud, y para actualizar sus estadísticas de salud (desde el método `UpdateHealthStats`).

Finalmente, se sobrecarga el método `<<` para poder imprimir un objeto de tipo `HealthStats`. Este es llamado a la hora de imprimir el estado de salud de cada país al final de un día en la simulación.