PrisonBreak

Alumno: Mario Corchero Jiménez

DNI: 09204648 W

Asignatura: **Laboratorio de Programación II**

Curso: **2008/2009**

Grupo de prácticas: 9

Profesor: Encarna Sosa Sánchez

Fecha: 9 de Junio de 2009

1.Manual de usuario

1.1. Introducción

Este software desarrollado consiste en una simulación de una prisión, en la cual se produce un intento de fuga, se controlan los sistemas de seguridad de las puertas, las rutas de vigilancia de los guardias, la posibilidad de que un preso robe llaves a los guardias o de que los guardias pierdan llaves. Además, la inicialización de la prisión se controla desde un fichero ajeno al programa, lo cual permite especificar el numero de presos, guardias y plantas que habrá en la prisión, junto con las características de los mismo con relativa facilidad. El programa incluye dos modos de ejecución, el normal y el modo de pruebas, el primero realiza una simulación basándose en el fichero de inicio y el segundo ejecutara el modo de pruebas para comprobar la funcionalidad del proyecto. Los guardias son soltados en la planta mas alta en la primera celda, mientras que los guardias serán colocados en la ultima celda de la planta indicada.

1.2. Guía de Instalación

La instalación del proyecto consiste simplemente en copiar el ejecutable a nuestro disco duro.

1.3. Interfaz de usuario

Para iniciar la ejecución del proyecto hemos de iniciarlo añadiendo los siguientes parámetros:

- [ruta con el fichero de inicio]: Se indica el fichero de inicio desde el cual se cargara la configuración, es importante seguir el siguiente formato:
  - Las lineas que comienzan con "--" serán consideradas anotaciones.
  - Todos los campos deben de separarse por "#"
  - El formato de la planta sera:
  PLANTA#<id>#<ancho>#<alto>#<entrada>#<salida>#<numero llaves>#<condición de altura en puerta>#
  Ejemplo: PLANTA#0#5#5#0#40#20#3
  - El formato de los guardias sera:
  GUARDIA#<nombre>#<marca>#<planta>#<turno en el que comienza a moverse>#
  Ejemplo: GUARDIAS#el_culebras#G#0#5#
  - El formato de los presos sera:
  PRESO#<nombre>#<marca>#<secuencia de elección de ruta>#<turno en el que comienza a moverse>#
  Ejemplo: PRESO#el_navaja#@#NSEO#4#

  Es importante que se siga el formato, que escriban las plantas en orden de identificador y los personajes en el orden del turno en el que se mueven, a su vez.
- TEST: (opcional) activa el modo de pruebas
- Nota: es recomendable usar tuberías para poder comprobar lo que obtendremos por pantalla, puesto que es mucha información y el terminal acabo borrando los primeros datos. Podemos dirigir la información con "> [ruta del archivo de salida]"
- Tras ejecutar el programa de forma exitosa, nos generará un fichero nombrado registro con la información del resultado de la simulación.
- Ejemplos: ./EC3 inicio.txt > salida.txt TEST
  ./EC3 configuracion.txt

1.4. Ejemplo de funcionamiento

El funcionamiento del programa es sencillo, aun así, proporcionamos el siguiente ejemplo:
Crear un fichero de inicio como este:
[inicio.txt]

```
----------------------------------------
--Fichero de Configuracion
--Plantas de la Prision
PLANTA#0#6#6#0#30#45#4#
PLANTA#1#10#10#0#99#45#4#
--Personajes de la Simulacion
PRESO#@#@#NESO#1#
PRESO#$#$#ESON#3#
PRESO#X#X#SENO#5#
GUARDIA#G#G#0#1#
GUARDIA#U#U#1#3#
----------------------------------------
```

Si tuviéramos el programa en la misma carpeta del fichero lo ejecutaríamos como:
./EC3 inicio.txt
De esta manera obtendremos un registro con el resultado de la simulación.
Si quisiéramos ejecutar las pruebas deberíamos de poner lo siguiente:
./EC3 inicio.txt TEST
De esta manera aparecería por pantalla el resultado de realizar las pruebas.

1.5. Lista de errores

Al ejecutar el programa podemos encontrar los siguientes errores:

-Si el archivo de inicio contiene las plantas indicadas en un orden incorrecto, nos aparecería un error, aunque seria controlado por el propio programa.

-Si el archivo de inicio no contiene los personajes indicados en orden de el turno en el que comienzan a moverse, la simulación no seria real, puesto que un preso no comenzara a moverse hasta que el que va delante se haya movido.

-Si el archivo de inicio contiene que un guardia patrulla una planta no existente, el programa informara del error y lo colocara en la ultima planta.

-Si el archivo de inicio contiene la ruta de los presos mal indicada, el programa terminará informando del error.

-Si el archivo de inicio contiene que una planta debe tener mas llaves de las que soporta (5 por cada celda, excepto en la entrada y la salida) informara de el error y creara solo el máximo de llaves permitidas.

-Si el archivo de inicio contiene que una planta tiene la puerta de entrada y/o salida mal colocada, informara del error y las colocara en una posición por defecto.

-Si el tamaño de una planta es incorrecto, el programa informara del error y le dará un tamaño estándar.

- Si no hay suficiente memoria para realizar la simulación el programa terminara.

- Si no se puede acceder correctamente al fichero de inicio el programa finalizara con un error de acceso.

- Si no se puede abrir correctamente el fichero el programa finalizara con un error de apertura.

-Si no se puede cerrar correctamente el fichero el programa finalizara con un error de cierre

2.Manual del Programador

2.1. Introducción

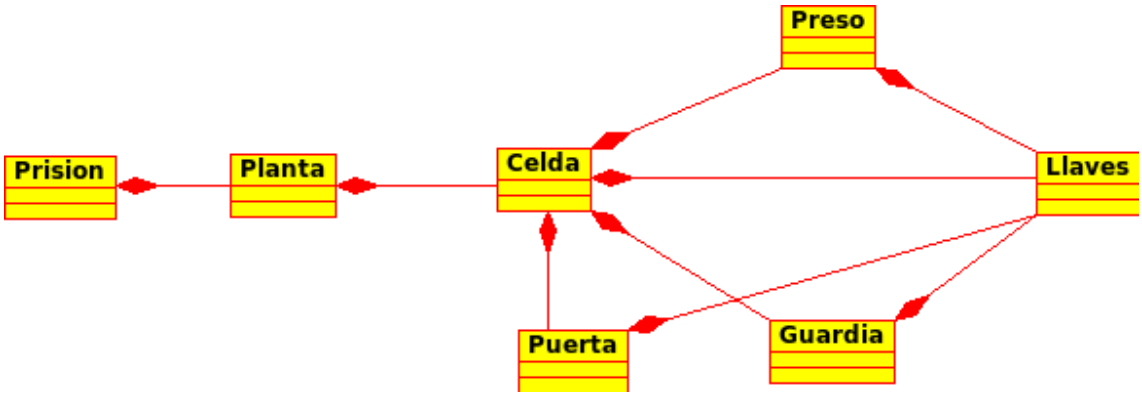
A continuación describimos el proceso de desarrollo del proyecto: Prison Break, Proyecto de Laboratorio de Programación 2 del curso académico 2008/2009, este software simula intentos de fuga masivos de prisiones a partir de unas especificaciones indicadas desde archivos. El resultado se obtiene a través de un fichero llamado registro.log.

2.2. Análisis

2.2.1. Identificación de las clases potenciales

Prisión, Celda, Planta, Llave, Puerta, Preso, Ruta, Alarma, Registro, Cerradura, Sistema, Mapa.

2.2.2. Diagrama de modelo conceptual (umbrello)



2.2.3. Descripción de las clases conceptuales

>>Arbol<<

Atributos:

T* dato_raiz_; Raíz del árbol
Arbol<T*> *hizq_, *hder_; Ramas del árbol
bool vacio_; Estado del árbol

Operaciones:

Arbol();
Efecto: Construye un árbol vacío.

Arbol(Arbol *hizq,T* dato,Arbol *hder);
Efecto: Construye un árbol a partir de los parámetro indicados.

Arbol *hijoIzq() const;
Efecto: Obtiene el hijo izquierdo del árbol

Arbol *hijoDer() const;
Efecto: Obtiene el hijo derecho del árbol.

T* raiz() const;
Efecto: Obtiene la raíz del árbol.

bool vacio() const;
Efecto: Comprueba si un árbol esta vacío.

bool insertar(T* dato);
Efecto: Inserta un dato en el árbol

bool pertenece (T* dato) const;
Efecto: Comprueba si pertenece un dato al árbol.

void borrar (T* dato);
Efecto: Borra un dato del árbol.

int Profundidad() const;
Efecto: Obtiene la profundidad del árbol.

bool EsHoja() const;
Efecto: Comprueba si un árbol es una hoja.

bool EsHoja(T* dato) const;
Efecto: Comprueba si el dato se encuentra en una hoja.

void NumNodos(int &hojas, int& nodos) const;
Efecto: Halla el número de nudos y hojas que tiene el árbol

int NumNodos() const;
Efecto: Halla el número de nodos que tiene un árbol

void Destruir();
Efecto: Libera la memoria y destruye el árbol

bool Equilibrado() const;
Efecto: Comprueba si un árbol esta equilibrado.

Arbol<T*>* Equilibrar();
Efecto: Equilibra un árbol.

bool operator==(const Arbol &A) const;
Efecto: Compara dos arboles (iguales)

bool operator!=(const Arbol &A) const;
Efecto: Compara dos arboles (distintos)

void ACola(queue<T*> &inorden) const;
Efecto: Copia los datos del árbol a una cola

void InOrden() const;
Efecto: Muestra el árbol en un recorrido en profundidad inorden

void PreOrden() const;
Efecto: Muestra el árbol en un recorrido en profundidad preorden

void PostOrden() const;
Efecto: Muestra el árbol en un recorrido en profundidad postorden

>>Cargador<<

Atributos:

```
typedef struct DatoMapeo
{
   char nombre[50];
   int numCampos;
};
```

Operaciones:

Cargador(Prision* p);
Efecto: Construye un cargador con la prisión indicada

void crear(string elto, int numCampos, string vCampos[kMaxCampos]);
Efecto: Crea un objeto a través de los parámetros pasados.

>>Celda<<

Atributos:

int marca_; atributo usado por el algoritmo de kruskall
int id_; identificador de la celda (-1 si no esta inicializado)
priority_queue<Llave*, vector<Llave*>, CompLlavesMayor>* llaves_; Llaves que contiene la celda
queue<Persona*>* personas_; personas que están en la celda

Operaciones:

Celda();
Efecto: Construye una celda por defecto

Celda(const int &id);
Efecto: Construye una celda con un identificador especifico

~Celda();
Efecto: Destruye una celda, liberando toda la memoria contenida en ella, destruye personas y llaves

inline int get_id() const
Efecto: obtiene la id de la celda

void set_id(const int &id);
Efecto: establece la id de la celda

void set_llave(Llave *llave);
Efecto: inserta una llave en la celda

Llave* sacar_llave();
Efecto: saca una llave de la celda

inline void set_marca(const int &marca)
Efecto: establece la marca de la celda

inline int get_marca() const
Efecto: obtiene la marca de la celda

void set_persona(Persona* p);
Efecto: inserta una persona

int ocupacion() const;
Efecto: obtiene el numero de personas en la celda

Persona* get_persona() const;
Efecto: obtiene la primera persona de la celda

Persona* sacar_persona() const;
Efecto: obtiene y elimina la primera persona de la celda

bool Simular();
Efecto: realiza la simulación haciendo actuar a los personajes

>>ColaGen<<

Atributos:

TipoNodo *frente_; Puntero al frente de la cola
TipoNodo *fin_; Puntero al final de la cola
int num_elementos_; Número de elementos de la cola

Operaciones:

ColaGen ();
Efecto: Inicializa los datos de la cola (punteros a NULL)

ColaGen (const ColaGen &C);
Efecto: Inicializa los datos de la cola por copia

void insertar_dato (const T &dato);
Efecto: Inserta un dato en la cola (por el final)

void insertar_nodo (const TipoNodo &nodo);
Efecto: Inserta un dato enviado en un nodo en la cola (por el final)

T get ()const;
Efecto: Devuelve el dato que se encuentra en el frente de la cola

bool borrar ();
Efecto: Elimina el dato que se encuentre en el frente de la cola

ostream& mostrar () const;

Efecto: Muestra la cola por pantalla y guarda la información mostrada en un flujo

inline bool vacia () const
Efecto: Devuelve verdadero si la cola está vacía

inline int get_num_elementos ()
Efecto: Devuelve el número de elementos de la cola

~ColaGen ();
Efecto: Destruye la cola pero no la memoria almacenada

>>FicheroCarga<<

Atributos:

ifstream f_in_; atributo que contiene el flujo de entrada utilizado para extraer información del fichero.

string nombre_fichero_; atributo que contiene el nombre del fichero, se inicializa con el constructor.

Operaciones:

FicheroCarga(string nombre);
Efecto: Construye el objeto con un nombre de fichero especifico.

void ProcesarFichero();
Efecto: Procesa el fichero con nombre almacenado

void ProcesarDatos(int numCampos,string vCampos[kMaxCampos]);
Efecto: Procesa unos datos específicos.

>>GenAleatorios<<

Atributos:

static const int SEMILLA=1976; Semilla para inicializar la generación de números aleatorios
int numGenerados; Contador de números aleatorios generados
static GenAleatorios* instancia; Instancia de la propia clase (sólo habrá una en el sistema)

Operaciones:

static int generarNumero(int limiteRango);
Efecto: Genera un numero aleatorio

static int numerosGenerados();
Efecto: Obtiene el numero de números generados

static void destruir();
Efecto: Destruye la instancia

>>Grafo<<

Atributos:

int num_nodos_; Numero de nodos del grafo
TipoVectorNodos nodos_; Vector que almacena los nodos del grafo

TipoMatrizAdyacencia arcos_; Matriz de adyacencia, para almacenar los arcos del grafo

TipoMatrizBooleana warshall_path_; Matriz booleana de Camino (Warshall - Path)
TipoMatrizAdyacencia floyd_cost_; Matriz de Costes (Floyd - Cost)
TipoMatrizAdyacencia floyd_path_; Matriz de Camino (Floyd – Path)

Operaciones:
Grafo(void);
inline int get_num_nodos(void) const
Efecto: Método que devuelve el numero de nodos del grafo

bool EsVacio(void) const ;
Efecto: Comprueba si un grafo esta vacío.

bool set_arco(const TipoNodoGrafo &origen, const TipoNodoGrafo &destino, const int &valor);
Efecto: Inserta un arco con los parámetros especificados

bool del_arco(const TipoNodoGrafo &origen, const TipoNodoGrafo &destino);
Efecto: Borra un arco especifico

bool Adyacente(const TipoNodoGrafo &origen, const TipoNodoGrafo &destino) const ;
Efecto: Comprueba si dos nodos son adyacentes

int get_arco(const TipoNodoGrafo &origen, const TipoNodoGrafo &destino) const ;
Efecto: Obtiene el valor de un arco determinado

bool set_nodo(const TipoNodoGrafo &n);
Efecto: Inserta un nodo

bool del_nodo(const TipoNodoGrafo &nodo);
Efecto: borra un nodo

void Warshall(void);
Efecto: Halla la matriz wharshall de caminos

void Floyd(void);
Efecto: Halla las matrices de Floyd

void Siguiente(const TipoNodoGrafo &origen, const TipoNodoGrafo destino, TipoNodoGrafo &sig) const ;
Efecto: halla el siguiente nodo de una ruta dada

void Adyacentes(const TipoNodoGrafo &origen, TipoCjtoNodos &ady) const ;
Efecto: Obtiene los adyacentes de un nodo

bool ExisteCiclo(const TipoNodoGrafo &nodo) const ;
Efecto: Comprueba si existe algún ciclo partiendo de un nodo dado

>>Guardia<< (hereda de persona)

Atributos:

Operaciones:

string P2String();
Efecto: Pasa los datos del guardia a una cadena

Guardia(const string &nombre, const char &marca, const int &espera, const int &p, const int &c);
Efecto: construye una instancia de un guardia a partir de unos datos

~Guardia();
Efecto: destruye un guardia y libera la memoria asociada

>>Llave<<

Atributos:

int id_; identificador de la llave

Operaciones:

Llave();
Efecto: construye una instancia con valores por defecto

Llave(const int &i);
Efecto: construye una llave con un id indicado

Llave(const Llave &L);
Efecto: Construye una llave copiando datos de otra

int get_id() const;
Efecto: obtiene el identificador de una llave

void set_id(const int& a);
Efecto: Pone el identificador de una llave

~Llave();
Efecto: destruye una llave

Llave operator=(const Llave &L);
Efecto: asigna una llave

bool operator<(const Llave &L) const;
Efecto: Compara dos llaves

bool operator>(const Llave &L) const;
Efecto: Compara dos llaves

bool operator==(const Llave &L) const;
Efecto: Compara dos llaves

bool operator!=(const Llave &L) const;
Efecto: Compara dos llaves

>>Persona<<

Atributos:

string nombre_; Nombre de la persona (tomaremos -1 como no inicializado)
char marca_; Marca de la persona, (tomaremos 0 como no inicializado)
queue <Posicion> *ruta_; Ruta que debe seguir la persona.
stack <Llave*> llaves_; Llaves que tiene la persona.
int planta_; Planta en la que se encuentra la persona.
int celda_; Celda en la que se encuentra la persona.
int turno_; Atributo que indica el ultimo turno en el que se han actuado (comienza con el primer turno en el que se puede mover)

Operaciones:

Persona(const string &nombre, const char &marca, const int &espera, const int &p, const int &c);
Efecto: Construye una persona a partir de unos datos específicos

virtual ~Persona();
Efecto: destruye una persona liberando toda la memoria asociada

string get_nombre()
Efecto: devuelve el nombre de la persona

inline char get_marca()
Efecto: devuelve la marca de la persona

inline void set_celda(const int &celda)
Efecto: Cambia el atributo celda por el que entra por parámetro

inline int get_turno()
Efecto: obtiene el turno

Llave* sacar_llave();
Efecto: obtiene y elimina la primera llave

void put_llave(Llave* l);
Efecto: inserta una llave

bool Actuar();
Efecto: realiza las 4 acciones indicadas por la simulación

virtual string P2String();
Efecto: pasa los datos a cadena

void MostrarRuta();
Efecto: muestra la ruta y la escribe en el registro

>>Planta<<

Atributos:

int id_; identificador de la planta.
int ancho_; ancho de la planta
int alto_; alto de la planta
int entrada_; entrada de la planta
int salida_; salida de la planta
Grafo grafo_; grafo que controla los caminos de la planta
Puerta *puerta_; puntero a puerta de la planta
vector <Celda*> celdas_; vector que almacena punteros a todas las celdas.
int n_llaves_; Numero de llaves diferentes generadas en la planta.
vector< Guardia*> guardias_; Lista de los guardias de la planta.

Operaciones:

Planta(const int &identificador, const int &ancho, const int &alto, const int &entrada, const int &salida, const int &n_llaves, const int &altura_control);
Efecto: crea una planta con los datos dados

~Planta();
Efecto: destruye una planta y libera la memoria asociada

int get_id() const;
Efecto: obtiene el id de la planta

void Pintar() const;
Efecto: pinta la planta por pantalla y en el registro

void set_persona(const int pos, Persona * p);
Efecto: inserta una persona en una celda indicada

Persona* get_persona(const int &pos) const;
Efecto: obtiene una persona de una celda indicada

Persona* sacar_persona(const int &pos);
Efecto: obtiene y elimina la primera persona de una celda indicada

void set_llave(const int &celda, Llave* llave);
Efecto: inserta una llave en una celda indicada

Llave* sacar_llave(const int &celda);
Efecto: obtiene y elimina una llave de una celda indicada

int ocupacion(const int &pos) const;
Efecto: obtiene el numero de personas que hay en una celda

int get_ancho() const;
Efecto: obtiene el ancho de la planta

int get_alto() const;
Efecto: obtiene el alto de la planta

bool Accesible(const int &origen, const int &destino) const
Efecto: Comprueba si dos accesibles entre si
inline void set_id(const int &id)
Efecto: Asigna un nuevo id a la planta

inline int get_entrada() const
Efecto: Devuelve la entrada de la planta

inline int get_salida() const
Efecto: Devuelve la salida de la planta

inline bool ProbarLlave(Llave * llave)
Efecto: Prueba una llave en la puerta

bool AbrirPuerta()
Efecto: intenta abrir la puerta de la planta

bool CerrarPuerta()
Efecto: Cierra la puerta de la planta

TipoEstado EstadoPuerta()
Efecto: Obtiene el estado de la puerta

void CaminoMinimo(const int &origen, const int &destino, queue<Posicion> &camino) const;
Efecto: obtiene el camino mínimo de una celda a otra

int get_vecina(const Posicion &p, const int &id) const;
Efecto: obtiene la celda vecina de una celda dada hacia orientación dada

bool Simular();
Efecto: Realiza la simulación en la planta

void RegistrarGuardia(Guardia *g);
Efecto: Registra un Guardia en la lista de guardias de la planta

int get_num_guardias() const;
Efecto: obtiene el numero de guardias registrados

Guardia* get_guardia(const int &numero) const;
Efecto: obtiene un guardia de la lista de guardias

>>Preso<< (hereda de persona)

Atributos:

Operaciones:

Preso(const string &nombre, const char &marca, const int &espera, const Posicion orientacion[4], const int &p, const int &c);
Efecto: Construye un preso con los datos dados

string P2String();
Efecto: pasa los datos de un preso a una cadena

~Preso();
Efecto: destruye un preso junto con la memoria asociada a este

>>Prision<<

Atributos:

list <Persona*> celda_castigo_; Lista de las personas almacenadas en la celda de castigo
list <Persona*> fugados_; Lista de personas que se han fugado
int turno_; Entero que almacena el turno actual de la simulacion
vector<Planta*> *plantas_; Vector que contiene todas las plantas de la prision
int n_presos_; Numero de presos que hay en la prisión

Operaciones:

~Prision();
Efecto: Destructor de prisión

static Prision* get_instancia();
Efecto: obtiene la instancia de la prisión

void set_planta(Planta* p);
Efecto: inserta una planta

int get_numero_plantas() const;
Efecto: obtiene el numero de plantas

Planta* get_planta(const int &posicion) const;
Efecto: obtiene una planta especifica

void MostrarPrision() const;
Efecto: muestra la prisión

void Destruir();
Efecto: Destruye la prisión y toda la memoria asociada

void set_persona(const int &planta, const int &pos, Persona *p);
Efecto: Inserta una persona.

int ocupacion_celda(const int &planta, const int &celda) const;
Efecto: obtiene el numero de personas que hay en una celda y una planta especificas

Persona* sacar_persona(const int &planta, const int celda);
Efecto: saca una persona de un sitio indicado

Persona* get_persona(const int &planta, const int &celda) const;
Efecto: obtiene una persona de la celda indicada

int get_ancho_planta(const int &planta) const;
Efecto: obtiene el ancho de una planta

int get_alto_planta(const int &planta) const;
Efecto: obtiene el alto de una planta

int get_entrada_panta(const int &planta) const;
Efecto: obtiene la posición de la entrada de una planta

int get_salida_planta(const int &planta) const;
Efecto: obtiene la posición de la salida de una planta

void meter_celda_castigo(Persona *p);
Efecto: mete una persona en la celda de castigo

void meter_en_fugados(Persona *p);
Efecto: mete una persona en la lista de fugados

void IniciarSimulacion();
Efecto: realiza la simulación

>>Puerta<<

Atributos:

TipoEstado estado_; estado de la puerta
int altura_; altura que no debe superar el árbol para poder abrir la puerta
Arbol <Llave*>* combinacion_; árbol binario de búsqueda que almacena la combinación correcta.
Arbol <Llave*>* probadas_; árbol binario de búsqueda que almacena las llaves probadas en la puerta.
ColaGen <Llave*>* copia_combinacion_; copia de la combinación secreta para poder reconfigura la puerta.

Operaciones:

Puerta();
Efecto: Construye una puerta con valores por defecto

Puerta(const int &alt);
Efecto: Construye una puerta con una altura de apertura determinada

void Configurar(ColaGen <Llave*>* cop);
Efecto: copia la combinación desde la copia

bool ProbarLlave(Llave *L);
Efecto: Prueba una llave

bool Abrir();
Efecto: intenta abrir una puerta

bool Cerrar();
Efecto: cierra una puerta(si esta abierta)

Árbol <Llave*>* get_combinacion();
Efecto: obtiene la combinación

Arbol <Llave*>* get_probadas();

Efecto: obtiene las llaves probadas

ColaGen <Llave*>* get_copia_combinacion();
Efecto: obtiene la copia de la combinación

TipoEstado get_estado();
Efecto: obtiene el estado de la puerta

static void Alarma();
Efecto: lanza la alarma

~Puerta();
Efecto: destruye la puerta y la memoria asociada

>>Registro<<

Atributos:

ofstream f_out_; flujo a través del cual se mandan los datos
char* nombre_fichero_; Nombre de fichero en el que se guardan los datos (por defecto "registro.log")
static Registro* instancia_; Instancia de la propia clase (solo habrá una en el sistema)

Operaciones:

static Registro* get_instancia();
Efecto: obtiene la instancia

~Registro();
Efecto: destructor del registro

void Escribir(char* mensaje);
Efecto: escribe en el fichero.

void Escribir(const string mensaje);
Efecto: escribe en el fichero.

void Escribir(const int mensaje);
Efecto: escribe en el fichero.

ostream& Flujo();
Efecto: obtiene el flujo del registro

void Destruir();
Efecto: Destruye el objeto y cierra el fichero

2.3. Diseño

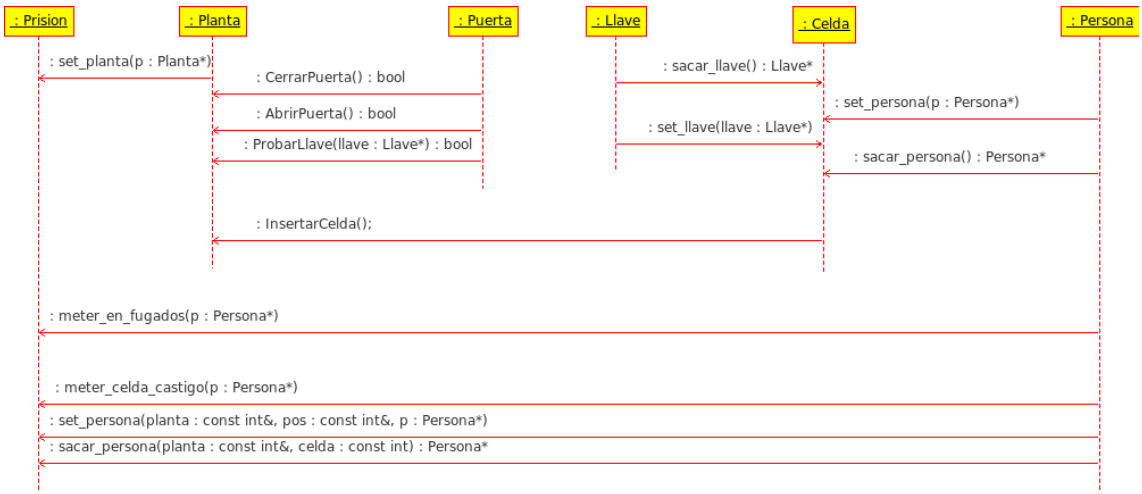
2.3.1. Diagramas de clases (umbrello)

(copia adjunta en la carpeta de documentación)

2.3.2. Detalle de clases diseñadas (doxygen)

Esta documentación se genera de forma automática en la carpeta del proyecto /html ejecutando Construir/Construir documentación del API.

2.3.3. Diagrama de secuencia del algoritmo principal



2.3.4. Algoritmos de especial interés

Árbol Árbol::Equilibrar()
Equilibra un árbol binario de búsqueda.
Void Prision::IniciarSimulacion()
Realiza todo el proceso de la simulación de la cárcel.

Void Preso::HallarRuta()
Halla una ruta de escape siguiendo un esquema algorítmico de vuelta atrás.

Void IniciarParedes()
Usa el algoritmo de kruskal para realizar el laberinto de la planta

2.3.5. Definición de entradas/salidas

La aplicación recibe a través de un fichero de texto la información necesaria para crear la prisión, y saca por fichero también el resultado de la simulación, información relativa a los presos, las celdas, las plantas y los guardias y el estado de la puerta. Además, muestra por pantalla algunos datos de interés como los momentos en los que un preso es detenido o cuando abre un puerta.

2.3.6. Variables o instancias más significativas y uso

La instancia de la clase Prisión(un Singleton) es la encargada principal de la ejecución y controla todo el programa a través de las clases de las que se compone.
La clase registro(otro Singleton) facilita bastante la escritura de los datos en archivo, puesto que nos permite escribir en nuestro registro.log desde cualquier parte del programa con bastante comodidad.

2.3.7. Lista de errores que el programa controla

El Programa detecta y avisa pero no corrige los siguientes errores:
-El orden de elección de direcciones para la ruta del preso no es correcto.
-Errores de apertura, acceso o cierre del fichero de configuración o del registro.

El programa avisa y corrige los siguientes errores:

-Se crea una persona en una planta incorrecta.
-El número de llaves indicado en el fichero para la planta es incorrecto.
-El tamaño de una planta es incorrecto(debe ser mayor que 2x2).
-La entrada o la salida están mal colocadas.
-Se inserta una planta con un id que no procede. Ej.: Inserta planta 2 sin haber ninguna.
-Se inserta una persona en una planta que no existe.

2.3.8. Justificación de estructuras de datos

-Celda:
Usamos una cola para guardar las personas por que las trataremos en el mismo orden que entran.
Usamos una cola de prioridad para las llaves porque necesitamos tenerlas ordenadas dentro de la celda.
-Persona:
Usamos una cola para almacenar la ruta puesto que la vamos a procesar en el orden en el que hemos insertando las direcciones.
Usamos una pila de llaves puesto que estas se procesarán en orden inverso a como se han guardado.
-Planta:
Usamos un vector de celdas para guardar las celdas puesto que necesitamos tener acceso a cualquier celda en cualquier momento.
Usamos un vector de guardias puesto que necesitamos tener acceso a cualquier guardia, por que necesita ser recorrido para comprobar si una persona pertenece al grupo de guardias.
-Prisión:
La celda de castigo sera implementada con una lista de personas por simplicidad.
Los presos fugados sera implementado como una lista de personas por simplicidad.
Las plantas son almacenadas en un vector para tener un acceso directo a cada una de ellas.
-Puerta:
La combinación de llaves es almacenada en un árbol binario de búsqueda equilibrado puesto que las búsquedas necesitan ser muy eficientes y esta estructura nos ofrece un coste de búsqueda de O(log n)
La copia de la combinación es almacenada en una cola, pues es una estructura para volcar en un árbol.
Las llaves probadas son almacenada en un árbol binario de búsqueda equilibrado puesto que las búsquedas necesitan ser muy eficientes y esta estructura nos ofrece un coste de búsqueda de O(log n)

2.4. Batería de pruebas

Las pruebas son realizadas con la librería de pruebas GTEST dentro del programa, al añadir TEST en los parámetros de ejecución.

2.5. Historial de desarrollo

Este proyecto a sido desarrollado de forma gradual a través de la evaluación continua de la asignatura, primero se diseño la clase y llave junto con su mecanismo de apertura, cierre y configuración, en esta fase la única dificultad se hallaba en el equilibrado del

árbol binario de búsqueda, que dio algunos problemas al tratarse de un árbol dinámico puesto que no tenia demasiada base de punteros.
La segunda fase se centro en la creación de las plantas y tratamiento de los archivos de inicio y el registro. La lectura del fichero de inicio fue sencilla debido a que nos proporcionaron el código para cargar los datos y respeto a la escritura en registro fue sencilla a través de la clase Registro, un singleton que permite escribir en el fichero desde cualquier punto del proyecto, facilitando mucho esta tarea. La verdadera dificultad se hallaba en la planta, a la cual añadimos un grafo para indicar que celdas eran accesibles entre si. Lo primero fue crear el laberinto a través del algoritmo de kruskal, usando una serie de números aleatorios, tras lo cual deberíamos de crear una serie de "atajos" elegidos también aleatoriamente, la dificultad de esta parte se hallaba en controlar las paredes que debían tirarse para no generarse "espacios", por ultimo, debíamos de hallar una ruta de escape especifica, problema fácilmente resuelto con un esquema algorítmico de vuelta atrás. En esta fase también había que realizar algunos test, una tarea mas larga que dificultosa. Por último debíamos de controlar las excepciones, una tarea entretenida y no demasiado difícil de implementar.
La tercera fase trataba a los presos, los guardias y la relación de todas las clases. La parte relativa a la herencia resulto relativamente fácil, al igual que relacionar todas las clases, el verdadero problema llego al tener que adaptar el registro crea con el registro de muestra, puesto que muchos algoritmos diseñados daban una solución, pero no la misma que la que se pedía, por lo cual tuve que rediseñar la mayoría de los algoritmos. Aquí la dificultad la encontré al tratar las perdidas de memoria que se me generaban al esparcirse las llaves por toda la planta, puesto que implemente las llaves con el mismo identificador como una sola llave apuntada por varios punteros, cosa que complico la liberación de memoria hasta el punto de tener que abandonar esta idea, una vez reservada memoria para cada llave, la liberación de memoria se hizo fácil de tratar. Por ultimo añadí un par de control de excepciones para las nuevas ampliaciones.
Tras esto, la implementación del software estaba terminada

2.6. Valoración Final del proyecto

El proyecto no ha sido demasiado dificultoso, tal vez haya sido un poco larga y tediosa la parte relacionada con la implementación de los test y la documentación, pero en general ha sido un proyecto entretenido. En las dos primeras entregas hemos contado con tiempo suficiente para realizarlas sin problemas, pero la tercera fase, se convirtió en una cuenta atrás debido al resto de exámenes con los que contábamos en junio, aun así, el tener las 2 entregas anteriores terminadas fue lo que marco la decisión de terminar la tercera, puesto que muchos alumnos al no tener la segunda entrega terminada no se decidieron a hacer la tercera.
Cabe destacar que aunque a lo largo del proyecto nos hemos encontrado con muchos problemas y dudas, hemos contado con una demo ejecutable para comprobar los resultados, con el foro para preguntar dudas y con las tutorías y el correo para contactar con los alumnos y los profesores.
En conclusión, ha sido un proyecto interesante, entretenido, y en el que he hemos mejorado mucho tanto nuestros conocimientos en c++ como la manera de afrontar los problemas a la hora de programar de una manera mas abstracta.