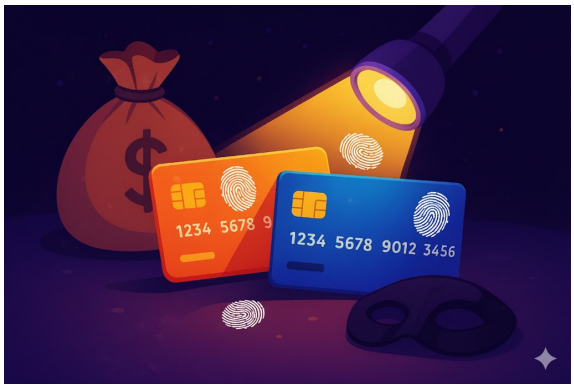
UNIVERSIDAD DE GRANADA

ETSIIT Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

DECSAI

Metodología de la Programación

Curso 2025/2026



Guion de prácticas

Fraud0

Paso de arrays como parámetros de funciones. Filtrado y unión de conjuntos de datos de entrada

Febrero de 2026

Silvia Acid, Andrés Cano, Luis Castillo
Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Granada

Índice

1. Las prácticas del curso, una visión general

El propósito de las prácticas que vamos a realizar a lo largo de este curso es trabajar con un conjunto de datos sobre compras realizadas por ciertos clientes en distintos establecimientos del campus de la Universidad de Princeton en EEUU (vea la figura 1). Un cliente puede estar realizando compras legítimas o bien fraudulentas, por ejemplo, por estar usando una tarjeta de crédito robada. A partir de un histórico de datos sobre compras, en el que haya datos de compras legítimas y compras fraudulentas, es posible construir un modelo que permita predecir si las compras que realiza un cliente son legítimas o fraudulentas. El objetivo de la práctica final será el de construir un clasificador que nos permita detectar posibles patrones de fraude en las compras que realiza un determinado cliente en distintos establecimientos del campus. Esta implementación final será una versión muy simplificada de los modelos que se emplean en el mundo real.

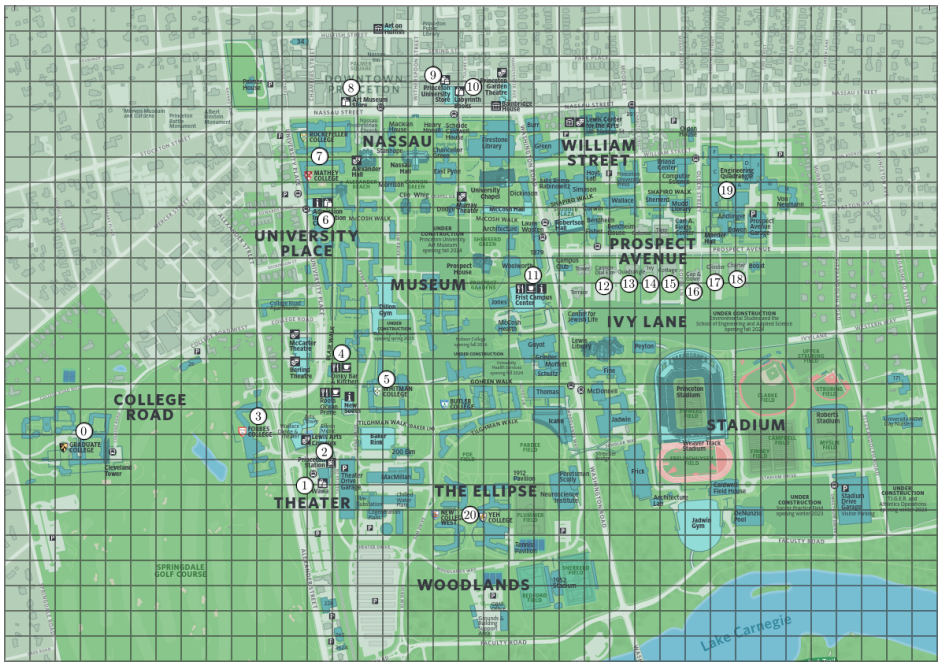


Figura 1: Conjunto de las 21 localizaciones del Campus de la Universidad de Princeton. Imagen "m=21 locations in Princeton", descargada en Enero de 2026 de la web https://www.cs.princeton.edu/courses/archive/spring25/cos226/assignments/fraud/specification.php

Indicar que, la idea original del problema y los datos proceden de una práctica de una asignatura de Algoritmos de la Universidad de Princeton, en la que el objetivo era hacer un estudio de diferentes algoritmos, objeto de esa materia [1]. Sin embargo, el objetivo de nuestras prácticas es utilizar esta aplicación como motivo atractivo para poner en práctica los conocimientos de programación propios de la asignatura Metodología de

[1] Puede consultar detalles de tal práctica en https://www.cs.princeton.edu/courses/archive/fall25/cos226/assignments/fraud/specification.php.

la Programación, y permitir por añadidura un primer contacto con el concepto de aprendizaje automático.

Para abordar el complejo problema descrito anteriormente, a lo largo de las sucesivas prácticas vamos a ir desarrollando un conjunto de clases que nos permitirán descomponer el problema inicial en problemas más sencillos de **resolver** y de **modelar**, y que servirán para practicar con los distintos conceptos que se irán introduciendo en las clases de teoría. Al mismo tiempo desarrollaremos un conjunto de **aplicaciones operativas** tales como la lectura y filtrado de datos; agrupamiento de datos mediante un algoritmo de clustering sencillo obteniendo diferentes configuraciones según los parámetros de entrada utilizados; aplicación de un algoritmo de reducción de dimensionalidad a un dataset con datos sobre compras en el campus de la Universidad de Princeton, algoritmo que se basará en el algoritmo de clustering anterior; y terminando en la práctica final de la asignatura con la construcción de un modelo que permita predecir o clasificar una instancia de una nueva compra como fraudulenta o no. Este proceso es conocido como aprendizaje supervisado. Puede consultar la siguiente url para ampliar conocimientos sobre ello: `https://es.wikipedia.org/wiki/Aprendizaje_supervisado`.

2. Arquitectura de las prácticas

Las prácticas `Fraud` se han diseñado por etapas con productos operativos desde la primera etapa. Las primeras etapas contienen clases más sencillas, sobre las cuales se asientan otras más complejas. Además, a lo largo del curso se irán incorporando nuevas funcionalidades a clases anteriores. La figura 2 muestra el conjunto de módulos que vamos a emplear (*la arquitectura*) y que se va a ir desarrollando progresivamente en esta y las siguientes sesiones de prácticas. Las estructuras de datos que se van a emplear van a sufrir varias modificaciones que se incorporan conforme se van adquiriendo nuevos conocimientos (los impartidos en teoría que se van a aplicar) o bien conforme llegan nuevas especificaciones (nuevos problemas para resolver). Esta situación no es excepcional, sino que la modificación y evolución de clases es un proceso habitual en cualquier proceso de desarrollo de software.

Describamos brevemente los módulos que utilizaremos a lo largo de todas las prácticas de esta asignatura:

A Location
: Implementa la clase `Location`, que almacena datos sobre una localización en la que alguien ha gastado dinero por algún servicio o producto. Una localización contiene datos sobre la posición x e y (valores `double`) del establecimiento y su nombre (una cadena de caracteres que puede estar vacía o contener una o varias palabras).

B VectorLocation
: Implementa la clase `VectorLocation`, un tipo de dato para almacenar un vector (conjunto) de objetos de la clase `Location`. Será

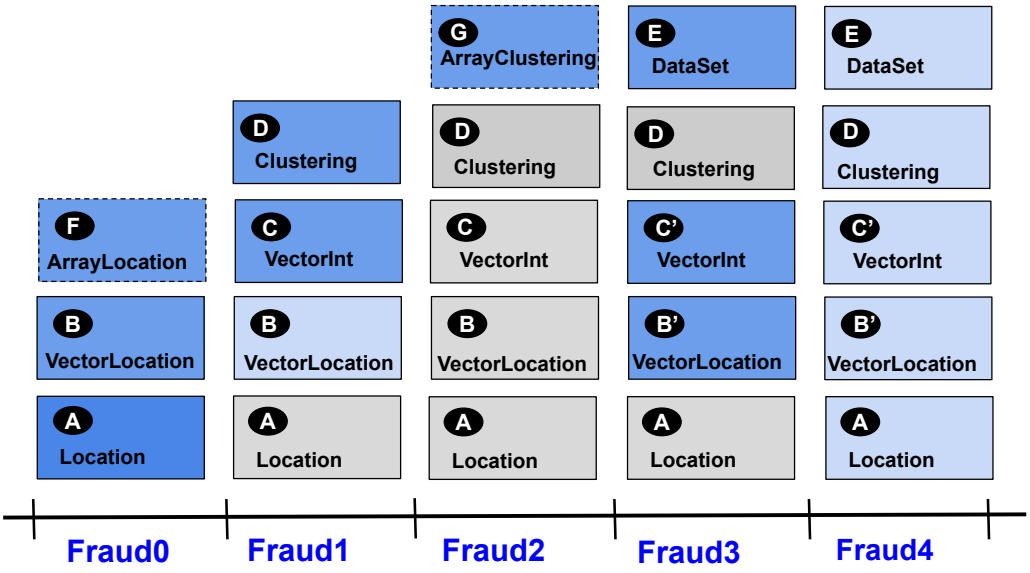


Figura 2: Arquitectura de las prácticas de MP 2026. Como podemos observar, los bloques correspondientes a cada módulo, aparecen en distintos colores. El color azul indica software que requiere ser desarrollado por el estudiante. Los cambios considerables (como los cambios en la estructura interna de una clase) se muestran en color intenso, mientras que pequeños cambios, como la incorporación de nuevas funcionalidades, se muestran en su correspondiente color suave. Finalmente, en gris se muestran módulos que no sufren cambios respecto a la versión anterior de las prácticas.

usado para almacenar el conjunto de distintas localizaciones que utilizaremos en las distintas prácticas. En una primera aproximación se usará un vector estático de un tamaño fijo.

B' VectorLocation
: Manteniendo la interfaz previa de esta clase, se cambiará su estructura interna para alojar el vector de localizaciones en memoria dinámica.

C VectorInt
: Implementa la clase `VectorInt`, un tipo de dato para almacenar un vector de números enteros (`int`). En una primera aproximación, se usará un vector estático de un tamaño fijo. Esta clase la usaremos en la clase `Clustering` para varios fines (como tipo para el dato miembro que almacena el número de clúster de cada localización o como tipo para una variable local del método `run()` que nos servirá de contador del número de localizaciones en cada clúster). En la clase `DataSet` también usaremos esta clase para varios fines (tipo para el dato miembro que guarda la etiqueta de cada instancia o para almacenar los valores de una de las instancias de un dataset).

C' VectorInt
: Manteniendo la interfaz previa de esta clase, se cambiará su estruc-

tura interna para alojar el vector de enteros en memoria dinámica.

D Clustering
: Esta clase será usada para obtener un agrupamiento (*clustering*) de un conjunto de localizaciones. Un algoritmo de agrupamiento se usa cuando se quieren agrupar las localizaciones de entrada en K grupos (*clusters*), de forma que las localizaciones de cada grupo estén próximas.

E Dataset
: Un dataset es un objeto que contiene un conjunto de `n` instancias. Cada instancia corresponde al importe de las compras que ha realizado un determinado cliente en cada una de las `m` localizaciones. Cada instancia es representada con un conjunto de m enteros. Internamente, la clase `Dataset` usa una matriz bidimensional (dinámica) de `n` filas y `m` columnas.

Por razones didácticas se van a elaborar dos módulos adicionales, indicados en la figura con rectángulos con bordes en línea discontinua, que contienen funciones externas cuyas funcionalidades serán trasladadas posteriormente al interior de algunas clases. Indicar que estos módulos desaparecen en versiones posteriores aunque **todo el esfuerzo invertido es reaprovechado** en la refactorización del código.

F ArrayLocation
: Módulo que contiene varias funciones externas, que tienen como parámetro, entre otros, un array de objetos `Location`, y cuya funcionalidad será introducida posteriormente en la clase VectorLocation.

G ArrayClustering
: Módulo que contiene varias funciones externas, que tienen como parámetro, entre otros, un objeto `ArrayClustering`. Este objeto representa un vector dinámico de objetos `Clustering`, cada uno de los cuales corresponde a la ejecución de un algoritmo de clustering con parámetros distintos.

Cada práctica requerirá además de la implementación de alguna(s) funcion(es) externa(s) y de una función `main()` propia para cubrir sus objetivos específicos.

3. Objetivos

El desarrollo de la práctica Fraud0 persigue los siguientes objetivos:

- Repasar conceptos sobre clases y los **calificadores** public, private, const y static en métodos y datos miembro.
- Revisar conceptos básicos de funciones: paso de parámetros por valor y referencia y devolución por valor.

- Practicar el paso de objetos por referencia a métodos o funciones.
- Introducir la devolución por referencia por parte de un método o función.
- Repasar el uso de arrays de objetos.
- Aprender a pasar arrays como parámetros de funciones.
- Construir una clase compuesta de datos miembro de otros tipos: la clase `Location`.
- Profundizar en la lectura de datos por redireccionamiento desde la entrada estándar.
- Reforzar la comprensión de los conceptos de compilación separada.

4. Práctica a entregar

4.1. Finalidad del programa

El programa Fraud0 va a trabajar con un conjunto de datos sobre distintos establecimientos (*localizaciones*), los cuales ofrecen algún tipo de servicio o producto. El propósito del programa será el de obtener la unión de dos subconjuntos de tales localizaciones. Al realizar la unión de los dos subconjuntos, debe tenerse en cuenta que las localizaciones que aparezcan a la vez en ambos subconjuntos, solo deben incluirse una vez en la unión. Cada subconjunto queda definido leyendo de la entrada estándar las coordenadas de las esquinas de las dos subáreas rectangulares de interés (coordenadas x e y de las esquinas inferior izquierda y superior derecha de cada subárea) y los datos del conjunto de todas las localizaciones disponibles. Cada uno de los subconjuntos de localizaciones se obtiene seleccionando del conjunto completo, aquellas que caen dentro de cada subárea.

El programa llevará a cabo los siguientes pasos:

- El programa comenzará leyendo todos los datos necesarios desde la entrada estándar. El conjunto de localizaciones debe almacenarse en un array de objetos `Location` declarado en `main()` como variable local (`arrayLocations`).
- A partir del array `arrayLocations`, el programa construirá un objeto (`locations`) `VectorLocation` con todas las localizaciones.
- Usando el objeto `locations` y las coordenadas de las dos subáreas de interés, se obtendrán los dos subconjuntos de localizaciones (dos objetos `VectorLocation`).
- A continuación se obtendrá la unión (un objeto `VectorLocation`) de los dos subconjuntos de localizaciones.

- El conjunto de localizaciones resultado de la unión será ordenado por orden alfabético del nombre de la localización.
- A partir del objeto `VectorLocation` ordenado obtendremos un nuevo array de objetos `Location`.
- Finalmente, el programa debe mostrar en la salida estándar el array de objetos `Location` obtenido en el paso anterior.

Ejemplo 1 *Un ejemplo de conjunto de datos que podría proporcionarse al programa por la entrada estándar es el siguiente (cuyo contenido también puede verse en el fichero* `CodeProjects/Datasets/dataP0/princeton_5locations.p0in`*):*

```
22.0 13.0
25.25 16.0
24.5 14.0
27.0 17.0
5
20.9 15.4 Frist Campus Center
23.7 14.8 Cannon Dial Elm
24.8 14.9 Quadrangle
25.6 14.9 Ivy
26.4 14.9 Cottage
```

Las dos primeras líneas contienen las coordenadas x e y de las esquinas inferior izquierda y superior derecha del la primera subárea de interés. Las dos siguientes líneas, las coordenadas de las esquinas de la segunda subárea de interés. En la quinta línea, el número entero 5 indica que a continuación deben leerse 5 objetos `Location`*. Fíjese que los campos de cada localización están separados por un espacio en blanco, aunque podría ser también cualquier otro separador (tabulador por ejemplo). También, puede ver que el nombre de la localización puede estar compuesto de varias palabras.*

4.2. Sintaxis y ejemplos de ejecución

En esta práctica, la entrada al programa se realizará completamente desde la entrada estándar. Puesto que el número de datos que hay que proporcionar al programa es muy grande, explicamos la forma de ejecutarlo con redireccionamiento de la entrada estándar desde un fichero, en lugar de usar el teclado, para que sea más cómodo de ejecutar, aunque sigue siendo válido ejecutar el programa leyendo los datos desde teclado.

La **sintaxis de ejecución** del programa desde un terminal es:

```
Fraud0> build/Fraud0 < <inputFile.p0in>
```

Veamos algunos ejemplos de ejecución del programa desde la línea de comandos:

Ejemplo 2 *Redireccionamiento desde el fichero* `princeton_5locations.p0in`

```
Fraud0> build/Fraud0 < ../Datasets/dataP0/princeton_5locations.p0in
```

El contenido del fichero `princeton_5locations.p0in` *es el que aparece en el ejemplo 1.*

En la figura 3, se pueden ver marcadas con rectángulos en rojo las dos subáreas que estamos proporcionando en este ejemplo. El conjunto de localizaciones completo que estamos usando estaría formado por aquellas numeradas desde 11 hasta 15 (ambas incluidas), que son las que entran dentro del rectángulo marcado en azul. El conjunto de localizaciones unión que obtendría el programa estaría formado por aquellas numeradas desde 12 hasta 15 (ambas incluidas).

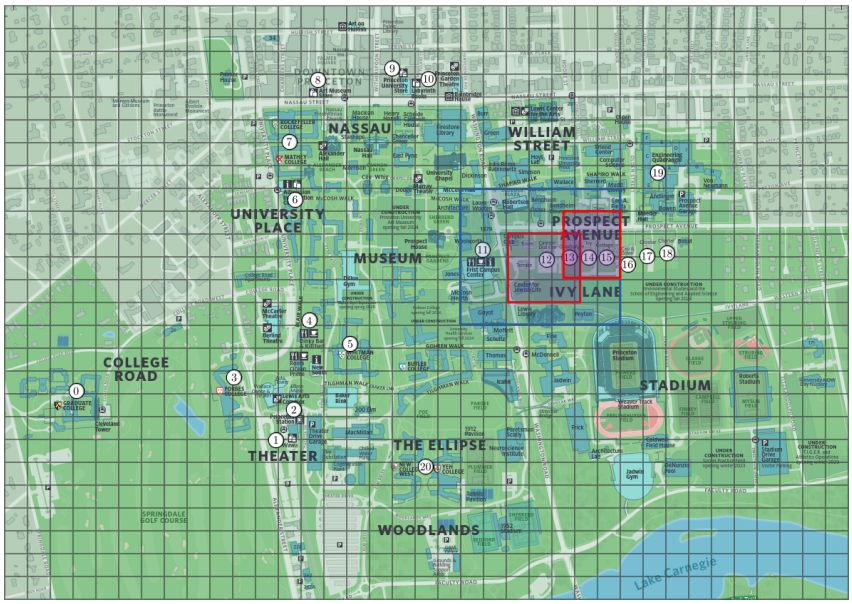


Figura 3: Unión de las dos subáreas del ejemplo 2

La salida que debería obtenerse al ejecutar el programa sería la siguiente:

```
4
23.700000 14.800000 Cannon Dial Elm
26.400000 14.900000 Cottage
25.600000 14.900000 Ivy
24.800000 14.900000 Quadrangle
```

Como puede comprobar, la localización Quadrangle (número 13) ha sido incluída una sola vez en la unión, aunque pertenece a las dos subáreas dadas. Por otro lado, la localización Frist Campus Center (número 11) no ha sido incluída en la unión, pues no pertenecía a ninguna de las dos subáreas dadas.
La salida del programa mostrada más arriba puede verse también en el fichero `CodeProjects/Datasets/dataP0/princeton_5locations.p0out`.

En la carpeta `CodeProjects/Datasets/dataP0`, puede encontrar varios ficheros de texto con extensión `.p0in` con los que podrá hacer diferentes pruebas. Esta carpeta también contiene ficheros con extensión `.p0out`. Cada uno de estos ficheros `.p0out` corresponde con la salida que debería obtenerse ejecutando el programa con el correspondiente fichero de entrada. Así por ejemplo, usando `princeton_5locations.p0in` como entrada, debería obtenerse el contenido del fichero `princeton_5locations.p0out`.

Además, se proporcionan una serie de ficheros de tests con extensión `.test` que contienen tests de integración, y que pueden usarse con el script `runTests.sh` para ejecutar automáticamente tales tests de integración según se explica en la sección 7.4.

4.3. Módulos del proyecto

Para la elaboración de la práctica dispone de una serie de ficheros (`Location.h, VectorLocation.h, ArrayLocation.h, main.cpp`) con código C++, donde se encuentran las especificaciones de lo que tiene que implementar.

Analice con cuidado las **cabeceras y comentarios de cada función o método**, pues en ellos están detalladas todas sus especificaciones. En la declaración del prototipo de cada función o método, *el número de argumentos* y *los tipos* han sido establecidos y **no se han de cambiar**. Sin embargo, **DEBE REVISAR** la forma en que se hace el paso de argumento para cada parámetro (por valor, por referencia o por referencia constante) y el calificador **const** o (no const) para argumentos y métodos. Hay indicaciones de si los parámetros son de entrada, de salida y si los métodos son de consulta o modificación.

Importante: En esta primera práctica, los ficheros `Location.h` y `VectorLocation.h` tienen declarados correctamente todos los prototipos de cada método y función, por lo que no necesita modificarlos. No obstante, estúdielos, ya que en próximas prácticas sí que tendrá que revisar los nuevos métodos y funciones que vayan apareciendo en estos u otros ficheros. Sin embargo, **sí que debe revisar y corregir** los prototipos de las funciones declaradas en el fichero `ArrayLocation.h`.

También debe tener en cuenta las **excepciones** que debe lanzar cada función cuando así se indica en la especificación (ver sección 4.4).

En esta práctica debe tener en cuenta las especificaciones indicadas en los siguientes módulos:

- Clase `Location`. El fichero **Location.h** (ver detalles en sección 8.1) contiene la declaración de la clase `Location`. La clase `Location` contiene los datos miembro `double _x` y `double _y` que se usan para almacenar las coordenadas de la localización. Además contiene el dato `string _name` usado para almacenar el nombre de la localización (una cadena de caracteres que puede estar vacía o contener una o varias palabras).

```
class Location {
...

private:
    /**
     * The x-coordinate of this location
     */
    double _x;

    /**
     * The y-coordinate of this location
     */
    double _y;

    /**
     * The name of this location. It can contain several words.
     */
    std::string _name;
}; // end class Location

/**
 * Removes spaces and \t characters at the beginning and at the end of the
 * provided string @p myString. If the provided string @p myString is empty
 * or contains only spaces or \t characters then @p myString will contain an
 * empty string after calling to this function.
 * @note This function can be easily implemented using the methods
 * find_first_not_of(string) and find_last_not_of(string) of class string.
 * @param myString a string. Input/Output parameter
 */
void Trim(std::string & myString);
```

El fichero **Location.h** contiene también la declaración de la función externa `Trim()` que tendrá que implementar en el fichero **Location.cpp**.

- Clase `VectorLocation`. El fichero **VectorLocation.h** (ver detalles en sección 8.2) contiene la declaración de la clase `VectorLocation`. La clase `VectorLocation` permite guardar un conjunto de objetos de la clase `Location`. En esta práctica la clase usa un array alojado en memoria automática (en la pila) de capacidad fija. En la práctica `Fraud3`, modificaremos esta clase para que pase a usar un array alojado en memoria dinámica. Como puede ver, el dato miembro `_locations` es el array de objetos `Location`, que tiene una capacidad máxima de 100 objetos. El dato miembro `_size` permite conocer en todo momento el número de elementos usados en el array.

```
class VectorLocation {
...
private:
    /**
     * Constant with the capacity of the array _locations
     */
    static const int DIM_VECTOR_LOCATIONS = 100;

    /**
     * Array of Locations
     */
    Location _locations[DIM_VECTOR_LOCATIONS];

    /**
     * Number of Location objects contained in the array _locations
     */
    int _size;
}; // end class VectorLocation
```

- Módulo `ArrayLocation`. Para lograr la operatividad que se va a necesitar en la función `main()`, se van a definir una serie de funciones externas que realicen dichas tareas (ver detalles en sección 8.3). La funciones a desarrollar en este módulo van a necesitar pasar el array de objetos de la clase Location como parámetro, en sus diferentes modalidades de paso de parámetros, según que modifiquen o no los objetos de dicho array. En el fichero `ArrayLocation.h`, se encuentran las especificaciones de cada función.

Como podrá observar en el fichero `ArrayLocation.h`, las distintas funciones a implementar reciben como parámetro el array de objetos ArrayLocation, y un entero que nos permite conocer el número de elementos utilizados en el array, o bien dos enteros que dan la capacidad y el número de elementos utilizados del array.

- Módulo `main.cpp`. Este programa tiene por finalidad lo descrito en la sección 4.1. Teniendo en cuenta los detalles comentados en tal sección y los pasos esbozados en el código proporcionado para esta función (sección 8.4), complete el código de este módulo.

En los ficheros **Location.cpp**, **VectorLocation.cpp** y **ArrayLocation.cpp** implemente los métodos propios de cada clase y las funciones externas mencionadas en los correspondientes ficheros `.h`.

4.4. Las especificaciones sobre excepciones

Una excepción es evento síncrono que interrumpe el flujo normal del programa para indicar un error o situación inesperada en tiempo de ejecución. Por ejemplo, situaciones que podrían provocar estos problemas serían una división por cero, acceso a una posición fuera del rango dentro de un vector, intento de escribir en un fichero para el que no tenemos permiso de escritura, etc. Cuando nuestro programa detecte estas situaciones de error, una opción es lanzar una excepción de algún tipo, interrumpiendo el flujo normal del programa, haciendo que automáticamente se notifique del error a la *función llamante* de aquella en la que se ha producido el error. Esta es una forma muy adecuada de tener controladas las posibles situaciones de error que se pueden producir durante la ejecución de un programa.

Cuando se lanza una excepción, si esta no es capturada, es enviada a la función llamante. En la función llamante, si la excepción no es capturada, se envía de nuevo a su función llamante y así sucesivamente hasta que la excepción llega a `main()`. Si la excepción tampoco es capturada en `main()`, el programa termina y se muestra su descripción por la salida estándar.

A continuación, vamos a analizar las especificaciones sobre excepciones que nos podemos encontrar en esta asignatura en las cabeceras de métodos o funciones.

En primer lugar, veamos el método `Location &at(int pos)` de la clase `VectorLocation`, que devuelve una referencia hacia el elemento del vector situado en la posición recibida como parámetro. Si la posición proporcionada fuese incorrecta (negativa o mayor o igual al número de objetos almacenados en el vector), este método debe lanzar una excepción `std::out_of_range`.

En el fichero `VectorLocation.h` podemos ver la siguiente especificación para el método `Location &at(int pos)`:

```
/**
 * @brief Gets a reference to the Location element at the given position.
 * Modifier method
 * @throw std::out_of_range Throws an std::out_of_range exception if the
 * given position is not valid
 * @param pos position in the VectorLocation object. Input parameter
```

```
 * @return A reference to the Location element at the given position.
 */
Location &at(int pos);
```

Como puede verse, se indica que el método lanza la excepción `std::out_of_range` si la posición no es válida. En posteriores prácticas van a encontrarse otras excepciones diferentes como por ejemplo `std::invalid_argument` que se trataría de forma similar.

Indicar que por simplicidad, en esta asignatura, nuestro propósito será únicamente detectar estas situaciones de error y especificar dónde o cuál es la situación de error que produjo la excepción, mediante el uso de una cadena de texto que describa tal error y que cada uno podrá definir como quiera. Esta descripción acompañará al nombre de la excepción como puede verse más abajo para el ejemplo del método `at()`. El texto definido se mostrará automáticamente en la salida estándar, en caso de que se lance la excepción durante la ejecución del programa, ya que según acabamos de decir, en caso de que se lance una excepción el programa terminará y se mostrará por la salida estándar la descripción de la excepción.

Veamos cómo vamos a proceder en la implementación al encontrarnos con una excepción en la especificación. En primer lugar, se identifica la excepción especificada. En el ejemplo anterior tenemos especificada la excepción `std::out_of_range`. En segundo lugar, se lanza la excepción indicada con la palabra reservada `throw` **si se cumple la condición especificada** en la cabecera. En el caso del método `at()`, la condición para lanzar la excepción es que la posición sea inválida. En tal caso se usa `throw <tipo-excepción>(` mensaje identificativo`)` para lanzar la excepción. A continuación se muestra la definición completa del método `at()`.

```
Location & VectorLocation::at(int pos) {
    if (0 <= pos && pos < _size)
        return _locations[pos];
    else
        throw std::out_of_range(
                string("Location & VectorLocation::at(int pos): ") +
                "invalid position " + std::to_string(pos));
}
```

Indicar de nuevo que, si se capturase una excepción (usando la sentencia `catch`) se podría arreglar la situación de error, pero por simplicidad, en la asignatura Metología de la Programación, tan solo vamos a usar la parte `throw` sin la componente `catch`; o sea, nunca vamos a capturar excepciones en esta asignatura. Si quiere saber más: (Abrir →).

4.5. Entrega de la práctica

La práctica deberá ser entregada en Prado, durante el periodo habilitado en la propia actividad de entrega, y consistirá en un fichero ZIP del proyecto. El nombre del fichero puede ser cualquiera, pero tendrá la extensión zip. Por ejemplo, podría llamarse `PRACTICAMP.zip`. Para obtener este fichero se sugiere utilizar el script `runZipProject.sh` (ver sección 7). Compruebe que lo entregado es compilable y operativo y, no olvide poner el nombre de los componentes del equipo en la cabecera del fichero `main.cpp`.

5. Configuración de las prácticas

Para la elaboración de la práctica dispone de una serie de ficheros que se encuentran en el repositorio de `github`. Una vez descargado, va a encontrar entre otros, los ficheros `Location.h`, `VectorLocation.h`, `ArrayLocation.h` y `main.cpp`. El fichero `documentation.doxy` es un fichero de texto con la configuración para ejecutar el programa `doxygen`.

En la carpeta `CodeProjects/DataSets` puede encontrar varias subcarpetas con distintos ficheros para probar nuestras prácticas. En el caso de Fraud0, la carpeta `dataP0` contiene ficheros con extensión `.p0in` (ficheros a usar como entrada estándar al programa) y ficheros con extensión `.p0out`, que contienen la salida esperada al ejecutar el programa con la entrada correspondiente.

Para montar la primera práctica, tendrá que crear un proyecto nuevo según se especifica en la sección 6. Pero antes, veamos cómo definir el espacio de trabajo.

El espacio de trabajo

La estructura que deberíamos de tener es la que se muestra en la figura 4:

```
CodeProjects
├── Datasets
│   ├── dataP0
│   ├── dataP1
│   └── dataP*
├── Debugger*
├── Fraud0
├── Fraud1
├── Fraud*
├── HelloWorld
├── MPGeometry*
├── MyVector
├── Scripts
└── ValgrindShowcase
```

Figura 4: Árbol de directorios de la rama CodeProjects de la asignatura MP

- **CodeProjects:** Carpeta raíz en la que se van a crear todos los proyectos de la asignatura. Tendremos un directorio por cada proyecto Fraud.
- **DataSets:** Un conjunto de ficheros y subcarpetas que contienen datasets y ficheros para hacer pruebas con los programas de las distintas prácticas.

- **Scripts:** Una serie de scripts Bash de apoyo a la asignatura.

Para tener bien ordenado el contenido de cada proyecto Fraud, se va a organizar en las siguientes carpetas, según muestra la figura 5:

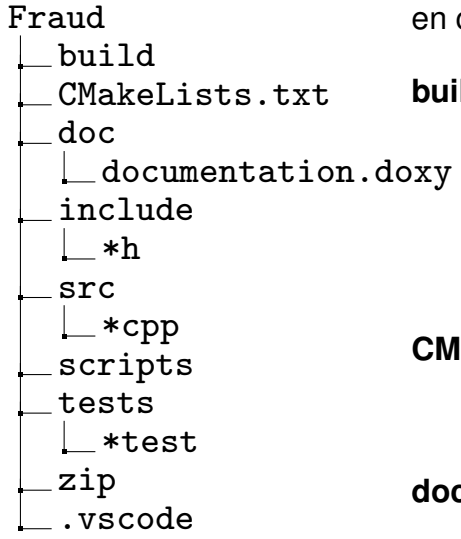


Figura 5: Estructura interna de cada proyecto Fraud

Lista de carpetas que deberíamos tener en cada proyecto Fraud:

build Es una carpeta temporal que crea automáticamente Visual Studio Code para cada proyecto. Entre otros ficheros, ahí colocará el ejecutable resultado de compilar nuestro programa.

CMakeLists.txt Este fichero contiene las instrucciones para compilar nuestro programa mediante `cmake`.

doc Aquí se almacenará la documentación del proyecto al usar el programa `doxygen`.

include Contiene los ficheros de cabeceras **.h** de nuestro proyecto.

src Contiene los ficheros fuente del proyecto **.cpp**.

scripts En esta carpeta colocaremos los scripts de apoyo a Visual Studio Code que se han desarrollado en esta asignatura.

tests Contiene tests de integridad de cada proyecto para comprobar que nuestro si programa funciona bien. Son ficheros con extensión **.test**.

zip Carpeta para dejar las copias de seguridad del proyecto.

.vscode Esta carpeta la crea Visual Studio Code en caso de que hayamos creado algún fichero de configuración para nuestro proyecto, como por ejemplo un fichero `launch.json`. Este fichero permite por ejemplo definir los parámetros con los que ejecutar un programa, que serán tenidos en cuenta en caso de ejecutar con el depurador de Visual Studio Code.

6. Configurar el proyecto Fraud0 en Visual Studio Code

Para crear un proyecto Fraud desde cero haremos los siguientes pasos:

1. Crear el directorio para el proyecto dentro de `CodeProjects`. Para esta primera práctica, puesto que ya tenemos un directorio con el nombre `Fraud0` dentro de `CodeProjects`, cambiaremos el nombre de la nueva carpeta, llamándola por ejemplo `MiFraud0`. Para crear el directorio, puedes hacerlo desde un terminal usando el comando `mkdir` <`nombreDir`>, o con el explorador de archivos de Linux o bien desde Visual Studio Code (VSCode) mediante Menú File --> Open Folder --> New Folder o también con Menú File --> Add Folder to Workspace --> New Folder.

2. Abrir el directorio del proyecto en VSCode. Si hemos creado el directorio con el terminal de Linux o el explorador de archivos, debemos abrir el directorio en VSCode mediante Menú File --> Open Folder o bien Menú File --> Add Folder to Workspace. Si el directorio lo creamos con VSCode, ya lo tendremos abierto.

3. Crear las carpetas `doc`, `include`, `src`, `scripts`, `tests` y `zip`, dentro del directorio del proyecto. De nuevo, esto se puede hacer desde un terminal usando el comando `mkdir` <`nombreDir`>, con el explorador de archivos de Linux o bien con la ventana Explorer de VSCode. Es necesario añadir las carpetas una a una.

   Carpetas como `build` o `.vscode` son creadas y gestionadas automáticamente por VSCode, por lo que no debemos crearlas nosotros manualmente.

4. Copiar en su sitio cada uno de los ficheros proporcionados con el proyecto `Fraud0` del repositorio github. Los ficheros `*.h` en `include`, los `*.cpp` en `src`, etc. Este paso puedes hacerlo desde el terminal de Linux o el explorador de archivos de Linux. También podrías añadir el proyecto `Fraud0` al Workspace, para tener abierto tanto `Fraud0` como `MiFraud0` y copiar los ficheros de uno a otro proyecto usando VSCode. La estructura debe quedar como la indicada en la figura 5.

5. Construir el fichero `CMakeLists.txt` en la carpeta raiz del proyecto. Podemos hacerlo manualmente con el editor de VSCode, o bien con el apoyo del asistente Quick Start de VSCode. El contenido de este fichero debería ser similar a lo siguiente:

```
cmake_minimum_required(VERSION 3.10.0)
project(Fraud0 VERSION 0.1.0 LANGUAGES C CXX)

set(CMAKE_CXX_STANDARD 14)
set(CMAKE_CXX_FLAGS "-Wall -Wextra -pedantic")

include_directories(include)

add_executable(Fraud0 src/Location.cpp src/VectorLocation.cpp
    src/ArrayLocation.cpp src/main.cpp)
```

   Como puede verse hemos añadido las siguientes opciones de compilación:

- `-Wall`: activa todos los mensajes de advertencia del compilador.
- `-Wextra`: activa mensajes de advertencia adicionales que se producen cuando nuestro programa no sigue las normas ISO C e ISO C++.
- `-pedantic`: hará que por ejemplo, nos aparezca una advertencia si declaramos un array de un tamaño que no sea una constante, algo prohibido en esta asignatura como en el siguiente ejemplo:

```
int main(){
    int variable=10;
    double array[variable];
}
```

6. Con esto ya se podría compilar y ejecutar el proyecto normalmente. Obviamente, no hace nada porque todavía está vacío. A partir de aquí empezaría el desarrollo del proyecto para el que cada programador seguirá un orden determinado y, probablemente, diferente a todos los demás.

   Nota: No deje para el final la comprobación del correcto funcionamiento de todos los métodos y no compruebe solo los métodos utilizados en `main()`. **Todos los métodos han de ser testados** para comprobar que cada uno funciona como se espera, aunque no sea utilizado en el `main()` solicitado para la práctica. Para ello deberá realizar una batería de pruebas.

7. Uso de scripts

En primer lugar, hemos de completar nuestro espacio de trabajo con la carpeta **Scripts**, también disponible en el repositorio de la asignatura (ver figura 4).

Como ya se indicara, la carpeta **Scripts** contiene una serie de scripts bash de apoyo a la asignatura. Es única para todos los proyectos que construyamos en Visual Studio Code, y se encuentra al mismo nivel que las carpetas de los proyectos Fraud.

Sin embargo, existe una carpeta `scripts` (con s minúscula) por cada uno de los proyectos Fraud (ver figura 5). No confundir con la carpeta `Scripts` anterior. La carpeta `scripts` es la que contiene los scripts que tú vas a poder ejecutar manualmente [2].

Como se ha indicado, en la carpeta `scripts` se encuentran una serie de utilidades bash, ficheros con extensión `*.sh` tales como:

- `runUpdate.sh` que actualiza los scripts de la carpeta `scripts` con los scripts actualizados que hayamos descargado del repositorio en la carpeta `Scripts`.

[2] Estos hacen uso de las utilidades en Scripts; por ello, para un correcto funcionamiento, es importante que `Scripts` esté correctamente ubicada en su lugar.

- `runDocumentation.sh` que va a facilitar el proceso de generación de la documentación con doxygen y su visualización en un navegador.
- `runZipProject.sh` que facilita el proceso de elaborar un zip, eliminando previamente todos los archivos binarios que no tiene sentido exportar como `*.bin, *.o,` etc. Guarda en la carpeta zip, el fichero que tiene que entregar en Prado.
- `runTests.sh` que ejecuta los tests de integración disponibles en la carpeta `tests` del proyecto proporcionado en el repositorio.

Los scripts proporcionados deben ser ejecutados desde un terminal de Linux, o bien desde el terminal de VSCode. Para que los scripts funcionen correctamente, es necesario que tengamos creada previamente la carpeta `build` dentro de la carpeta raiz de nuestro proyecto. Si no es así, no funcionarán bien los scripts. Recuerda que esta carpeta la crea automáticamente Visual Studio Code para nuestro proyecto. Si no la has podido crear con Visual Studio Code, créala manualmente para poder ejecutar los scripts.

7.1. El script runUpdate.sh

Este script copia la última versión de los scripts que tengamos en la carpeta `Scripts` y que hayamos descargado del repositorio de github. Ejecuta el script `runUpdate.sh`, que debería estar colocado en la carpeta `scripts`. Para ello, en el terminal sitúate dentro de la carpeta `scripts` y ejecuta el script `runUpdate.sh`:

```
scripts> ./runUpdate.sh
```

La ejecución de este script, hará que se copien otros ficheros en la carpeta `scripts` (ficheros runDocumentation.sh, runTests.sh, runZipProject.sh).

7.2. El script runZipProject.sh

El script `runZipProject.sh` nos permite obtener un fichero con extensión `.zip` en el directorio `zip` con el contenido de nuestro proyecto. De nuevo, sitúate dentro de la carpeta scripts, y ejecuta:

```
scripts> ./runZipProject.sh
```

Podrás comprobar que en la carpeta zip hay ahora un fichero llamado `PRACTICAMP.zip`.

7.3. El script runDocumentation.sh

Este script genera la documentación del proyecto en formato html a partir de los comentarios incluidos en el código de nuestros ficheros C++. Es posible también generar la documentación en formato pdf haciendo uso de Latex.

```
scripts> ./runDocumentation.sh
```

7.4. El script runTests.sh

Con cada proyecto Fraud se proporcionan varios tests de integración, cada uno de ellos en un fichero con extensión `.test`, y que deben estar colocados en la carpeta `tests`. Un test de integración es una prueba de nuestro programa, en la que se definen los datos que nuestro programa lee y la salida que debería producir con esos datos.

Por ejemplo, el fichero `fraud0.test` del proyecto Fraud0, tiene el contenido que se muestra a continuación:

```
%%%CALL < ../Datasets/dataP0/princeton_5locations.p0in
%%%DESCRIPTION Test to read a set of 5 locations
%%%OUTPUT
4
23.700000 14.800000 Cannon Dial Elm
26.400000 14.900000 Cottage
25.600000 14.900000 Ivy
24.800000 14.900000 Quadrangle
```

En la asignatura no es imprescindible entender el formato de estos ficheros de tests, pero lo comentamos aquí brevemente. En el test anterior, se indica que si el programa `Fraud0` se llama redirigiendo la entrada estándar desde el fichero `../Datasets/dataP0/princeton_5locations.p0in`, entonces la salida que debería producir el programa es la que se muestra tras la línea `%%%OUTPUT`. El contenido del fichero `princeton_5locations.p0in` aparece en la sección 4.2.

Para ejecutar los tests de integración, usaremos el script `runTests.sh`. Hay varias formas de hacerlo:

1. Desde el terminal de VSCode o de nuestro sistema operativo Linux podemos ejecutar todos los tests de integridad situándonos dentro de la carpeta de nuestro proyecto y ejecutando:

   ```
   Fraud0> scripts/runTests.sh
   ```

2. Desde el terminal de VSCode o de nuestro sistema operativo Linux podemos ejecutar uno solo de los tests situándonos dentro de la carpeta de nuestro proyecto y ejecutando:

   ```
   Fraud0> scripts/runTests.sh tests/nombreTestAEjecutar.sh
   ```

Al hacerlo con cualquiera de las opciones anteriores, nos aparecerá en pantalla la lista de tests ejecutados y podremos ver si nuestro programa los ha pasado o no. Por ejemplo, en Fraud0, si ejecutamos desde un terminal el script de la siguiente forma:

```
Fraud0> scripts/runTests.sh tests/fraud0.test
```

obtendremos por la salida (en caso de que nuestro programa pase el test) lo que aparece a continuación, que nos indica que ha pasado el test correctamente (aparece `OK`):

```
Loading ansiterminal...

VALIDATION TOOL v2.0
Checking tests folder           OK
Setting everything up           OK
Test #1  Test to read a set of 5 locations

Testing tests/fraud0.test               Generating fresh
 ↪  binaries       -- Configuring done (0.0s)
-- Generating done (0.0s)
-- Build files have been written to: /home/acu/Latex/Docencia/⌋
 ↪  MP/MP_2025_2026/Github/CodeProjects/Fraud0Teacher/build

[  OK  ] Test #1 [tests/fraud0.test] (  build/Fraud0Teacher  <
 ↪  ../Datasets/dataP0/princeton_5locations.p0in)
End of tests
```

Si hubiese fallado el test, habría aparecido `FAIL` en lugar de `OK`.

8. Código para la práctica

8.1. Location.h

```
/*
 * Metodología de la Programación
 * Curso 2025/2026
 */

/**
 * @file Location.h
 * @author Silvia Acid Carrillo <acid@decsai.ugr.es>
 * @author Andrés Cano Utrera <acu@decsai.ugr.es>
 * @author Luis Castillo Vidal <L.Castillo@decsai.ugr.es>
 *
 * Created on 6 de octubre de 2025, 12:27
 */

#ifndef LOCATION_H
#define LOCATION_H

#include <iostream>
#include <string>

/**
 * @class Location
 * @brief It stores data about a location where people spend money on some
 * service or product. A location contains data about the x and y position
 * (double values) in the 2D space and the name of the location
 * (a string that can contain several words).
 */
class Location {
public:
    /**
     * @brief Constructor that builds a Location object with a value of 0
     * for the x and y coordinates, and an empty string for the name.
     */
    Location();

    /**
     * @brief Returns the x-coordinate of this location
     * Query method
     * @return The x-coordinate of this location
     */
    double getX() const;

    /**
     * @brief Returns the y-coordinate of this location
     * Query method
     * @return The y-coordinate of this location
     */
    double getY() const;

    /**
     * @brief Returns the name of this location
     * Query method
     * @return The name of this location
     */
    std::string getName() const;

    /**
     * @brief Checks if the position of this object is inside the area
     * determined by the two given locations: Location A (@p bottomLeft) and
     * B (@p topRight).
     *
     *      +------------+(B)
     *      |            |
     *      |            |  +(O3)
     *      |            |
     *      |     +(O2)  |
     *      |            |
     *      |            |
     *   (A)+------------+
     *         +(O1)
     *
     * Examples of returning false:
     * - Object O3 is outside the given area (O3.x > B.x)
     * - Object O1 is outside the given area (O1.y < A.y)
     * Example of returning true:
     * - Object O2 is inside the given area
     * - Object A is inside the given area
     * - Object B is inside the given area
     *
     * Query method.
     * @param bottomLeft The Location of the bottom left point. Input parameter
     * @param topRight The Location of the top right point. Input parameter
     * @return true if the Location of this object is inside the area
     * determined by the two given Locations
     */
    bool isInsideArea(const Location &bottomLeft, const Location &topRight) const;

    /**
     * @brief Calculates the square of the Euclidean distante from this location
     * to the provided location. That is, it loc1 and loc2 are two
     * Location objects, then this method returns:
     * \f$ (loc1.x - loc2.x)^2 + (loc1.y - loc2.y)^2 \f$
```

```
     * Query method
     * @param location A Location object. Input parameter
     * @return Returns the square of the Euclidean distante from this
     * location to the provided location.
     */
    double squaredDistance(const Location& location) const;

    /**
     * @brief Calculates the Euclidean distante from this location to the provided
     * location. That is, it loc1 and loc2 are two Location objects, then
     * this method returns:
     * \f$ \sqrt{ (loc1.x - loc2.x)^2 + (loc1.y - loc2.y)^2 } \f$
     * Note that this method can be implemented using the squaredDistance()
     * method.
     * Query method
     * @param location A Location object. Input parameter
     * @return The Euclidean distante from this location to the provided
     * location.
     */
    double distance(const Location& location) const;

    /**
     * @brief Obtains a string with the x and y coordinates
     * and the name (whitespace separated). If the name is an empty string,
     * then only the x and y coordinates are included in the resulting string.
     * To convert the x and y coordinates to a string you must use the
     * std::to_string(int) C++ function.
     * Query method
     * @return string with information about this Location object
     */
    std::string toString() const;

    /**
     * @brief Sets the x-coordinate in this object
     * Modifier method
     * @param x The new value for the x-coordinate. Input parameter
     */
    void setX(double x);

    /**
     * @brief Sets the y-coordinate in this object
     * Modifier method
     * @param y The new value for the y-coordinate. Input parameter
     */
    void setY(double y);

    /**
     * @brief Sets the name in this object
     * Modifier method
     * @param name A string with the new value for the name. Input parameter
     */
    void setName(const std::string& name);

    /**
     * @brief Sets the x and y coordinates, and the the name in this object
     * Modifier method
     * @param x The new value for the x-coordinate. Input parameter
     * @param y The new value for the y-coordinate. Input parameter
     * @param name A string with the new value for the name. Input parameter
     */
    void set(double x, double y, const std::string& name);

    /**
     * @brief Reads from the provided input stream the information
     * to fill this Location object.
     * @note This method reads two double values (space separated) from the
     * input stream that are used to set the x and y coordinates of this object.
     * Then, it reads the string that can be formed from the current position of
     * the input stream to the end of the line. This string is then trimmed
     * (see the Trim(string) function) to remove its spaces at the beginning
     * and at the end. The resulting string is used to set the name of this
     * Location object. Note that the name can have several words separated by
     * whitespaces.
     * @param is Input stream. Input/output parameter
     */
    void load(std::istream& is);

private:
    /**
     * The x-coordinate of this location
     */
    double _x;

    /**
     * The y-coordinate of this location
     */
    double _y;

    /**
     * The name of this location. It can contain several words.
     */
    std::string _name;
}; // end of class Location

/**
 * Removes spaces and \t characters at the beginning and at the end of the
 * provided string @p myString. If the provided string @p myString is empty
 * or contains only spaces or \t characters then @p myString will contain an
 * empty string after calling to this function.
 * @note This function can be easily implemented using the methods
```

```
 * find_first_not_of(string) and find_last_not_of(string) of class string.
 * @param myString a string. Input/Output parameter
 */
void Trim(std::string & myString);

#endif /* LOCATION_H */
```

8.2. VectorLocation.h

```
/*
 * Metodología de la Programación
 * Curso 2025/2026
 */

/**
 * @file VectorLocation.h
 * @author Silvia Acid Carrillo <acid@decsai.ugr.es>
 * @author Andrés Cano Utrera <acu@decsai.ugr.es>
 * @author Luis Castillo Vidal <L.Castillo@decsai.ugr.es>
 *
 * Created on 11 de diciembre de 2025, 11:27
 */

#ifndef VECTORLOCATION_H
#define VECTORLOCATION_H

#include "Location.h"

/**
 * @class VectorLocation
 * @brief A VectorLocation object contains a vector of Location objects. It has
 * a capacity (maximun number of Locations that can be stored in the vector)
 * and a size (number of Locations the vector currently contains).
 * The public methods of this class do not allow a VectorLocation to contain two
 * Location objects with identical names.
 * This class uses an array of Location objects with a fixed capacity
 * to store the set of locations.
 */
class VectorLocation {
public:
    /**
     * @brief It builds a VectorLocation object (vector of Location objects)
     * with a size equal to the provided value (@p size).
     * Each element in the vector is initialized with the default Location
     * constructor.
     * @throw std::out_of_range Throws a std::out_of_range exception if
     * @p size < 0 or size>DIM_VECTOR_LOCATIONS
     * @param size The size for the vector of Location in this object. Input
     * parameter
     */
    VectorLocation(int size = 0);

    /**
     * @brief Gets the number of elements in the vector of this object
     * Query method
     * @return The number of elements
     */
    int getSize() const;

    /**
     * @brief Gets the capacity of the vector in this object
     * Query method
     * @return The capacity of the vector in this object
     */
    int getCapacity() const;

    /**
     * @brief Obtains a string with information about this VectorLocation object,
     * in the following format:
     * - First line, the number of Location objects in this vector converted to
     * a string (using the to_string(int) C++ function).
     * - For each Location, a line with its x and y coordinates, converted to
     * a string (using the to_string(int) C++ function) and separated by a
     * whitespace.
     * Query method
     * @return string with information about this VectorLocation object
     */
    std::string toString() const;

    /**
     * @brief Searches the provided Location in the array of locations in this
     * object. If found, it returns the position where it was found. If not,
     * it returns -1. We consider that position 0 is the first location in the
     * list of locations and this->getSize()-1 the last location.
     * In order to find a location consider only equality in the name field.
     * Query method
     * @param location A Location. Input parameter
     * @return If found, it returns the position where the location
     * was found. Otherwise it returns -1
     */
    int findLocation(const Location &location) const;

    /**
     * @brief Returns a VectorLocation object with those locations whose
     * positions are inside the area determined by the two given Locations.
     * Query method
     * @param bottomLeft The Location of the bottom left point. Input parameter
     * @param topRight The Location of the top right point. Input parameter
     * @return A VectorLocation with the selected Locations.
     */
    VectorLocation select(const Location &bottomLeft,
                          const Location &topRight) const;

    /**
     * @brief Removes all the elements in this object, leaving the container
     * with a size equal to 0. It only needs to set the number of elements
```

```
     * (_size field) to zero.
     * Modifier method
     */
    void clear();

    /**
     * @brief Gets a const reference to the Location element at the given
     * position
     * Query method
     * @throw std::out_of_range Throws an std::out_of_range exception if the
     * given position is not valid.
     * @param pos position in the VectorLocation object. Input parameter
     * @return A const reference to the Location element at the given position
     */
    const Location &at(int pos) const;

    /**
     * @brief Gets a reference to the Location element at the given position.
     * Modifier method
     * @throw std::out_of_range Throws an std::out_of_range exception if the
     * given position is not valid
     * @param pos position in the VectorLocation object. Input parameter
     * @return A reference to the Location element at the given position.
     */
    Location &at(int pos);

    /**
     * @brief Appends a copy of the given Location object at the first free
     * position in the array of Location in this object. The location is
     * not appended to this object if it was already found in this object.
     * @throw std::out_of_range Throws a std::out_of_range exception if the
     * provided location is going to be appended but the array of Location
     * was full (its capacity was full). If the provided location is not going
     * to be appended because it was already found in this object or its name
     * is an empty string, then no exception is thrown.
     * Modifier method
     * @param value the new Location object to be appended. Input parameter
     * @return true if the given Location could be inserted in this
     * VectorLocation object; false otherwise (the location was already found
     * in this object)
     */
    bool append(const Location &location);

    /**
     * @brief Appends to this VectorLocation object, the list of
     * Location objects contained in the provided VectorLocation object
     * that are not found (using VectorLocation::findLocation(Location)) in
     * this object.
     * This method could be implemented with the help of the method
     * VectorLocation::append(const Location & location), to append to this
     * object, the Locations of the provided VectorLocation object.
     * Modifier method
     * @param crimeSet A VectorLocation object. Input parameter
     */
    void join(const VectorLocation &locations);

    /**
     * Sorts the array of locations in this object by increasing alphabetical
     * order of the name of its location (a string).
     * Modifier method
     */
    void sort();

private:
    /**
     * Constant with the capacity of the array _locations
     */
    static const int DIM_VECTOR_LOCATIONS = 100;

    /**
     * Array of Locations
     */
    Location _locations[DIM_VECTOR_LOCATIONS];

    /**
     * Number of Location objects contained in the array _locations
     */
    int _size;
}; // end of class VectorLocation

#endif /* VECTORLOCATION_H */
```

8.3. ArrayLocation.h

```
/*
 * Metodología de la Programación
 * Curso 2025/2026
 */

/**
 * @file ArrayLocation.h
 * @author Silvia Acid Carrillo <acid@decsai.ugr.es>
 * @author Andrés Cano Utrera <acu@decsai.ugr.es>
 * @author Luis Castillo Vidal <L.Castillo@decsai.ugr.es>
 *
 * Created on 22 de octubre de 2025, 11:57
 */

#ifndef ARRAYLOCATION_H
#define ARRAYLOCATION_H

#include "Location.h"
#include "VectorLocation.h"

/**
 * @brief Displays on the standard output an integer (the value in nLocs,
 * number of locations in arrayLocations) and the content of the provided
 * array of locations. Each location is displayed on a new line with the help
 * of the Location::toString() method. For example, here it is displayed
 * an array with 4 locations:
4
24.8 14.9 Quadrangle
25.6 14.9 Ivy
26.4 14.9 Cottage
27.3 14.5 Cap & Gown
 * @param arrayLocations An array of Location objects. Input parameter
 * @param nLocs Number of Location objects in the array arrayLocations.
 * Input parameter
 */
void PrintArrayLocation(Location arrayLocations[], int nLocs);

/**
 * @brief Reads from the standard input an integer n and then reads
 * n Location objects that are placed into the provided array of locations.
 * @throw Throws a std::out_of_range exception if the integer n read from
 * standard input is negative. In that case, nLocs will be set to 0.
 * @throw Throws a std::out_of_range exception if the integer n read from
 * standard input exceeds the array capacity. In that case, nLocs will be set to 0.
 * @param arrayLocations An array of Location objects. Output parameter
 * @param capacity An integer with the capacity of arrayLocations.
 * Input parameter
 * @param nLocs An integer with the number of objects inserted in
 * arrayLocations. Output parameter
 */
void ReadArrayLocation(Location arrayLocations[], int capacity, int nLocs);

/**
 * @brief Places a copy of each Location of the provided VectorLocation object
 * into the provided array of Location (arrayLocations).
 * @throw Throws a std::out_of_range exception if the number of Location objects
 * in the provided VectorLocation exceeds the array capacity. In that case,
 * nLocs will be set to 0.
 * @param vector A VectorLocation object. Input parameter
 * @param arrayLocations Array of Location objects. Output parameter
 * @param capacity An integer with the capacity of arrayLocations.
 * Input parameter
 * @param nLocs An integer with the number of objects inserted in
 * arrayLocations. Output parameter
 */
void ToArrayLocation(VectorLocation vector, Location arrayLocations[],
                     int capacity, int nLocs);

/**
 * @brief Returns a VectorLocation object filled with a copy of the objects from
 * the provided array of Location objects (arrayLocation)
 * @throw Throws a std::out_of_range exception if nLocs is negative.
 * @param arrayLocations Array of Location objects. Input parameter
 * @param nLocs An integer with the number of objects in
 * arrayLocations. Input parameter
 * @return A VectorLocation object filled with the objects from the provided
 * array of Location objects (arrayLocation)
 */
VectorLocation ToVectorLocation(Location arrayLocations[], int nLocs);

#endif /* ARRAYLOCATION_H */
```

8.4. main.cpp

```
/*
 * Metodología de la Programación
 * Curso 2025/2026
 */

/**
 * @file main.cpp
 * @author Silvia Acid Carrillo <acid@decsai.ugr.es>
 * @author Andrés Cano Utrera <acu@decsai.ugr.es>
 * @author Luis Castillo Vidal <L.Castillo@decsai.ugr.es>
 *
 * Created on 24 de octubre de 2025, 9:27
 */

#include <iostream>

#include "Location.h"
#include "VectorLocation.h"
#include "ArrayLocation.h"

using namespace std;

/**
 * The purpose of this program is to obtain the union of two subsets of Location
 * objects. Each one of the two subsets is obtained by reading from the
 * standard input a set of Location objects and the coordinates of the corners
 * of two rectangular subareas of interest (x and y coordinates
 * of the bottom left and top right corners of the subarea). Each subset of
 * locations is obtained by selecting those locations that are whithin the
 * given subarea.
 *
 * This program first reads all the necessary data from the standard input;
 * then it calculates the union of the two subsets of Location objects;
 * it sorts the resulting set of locations and finally it displays
 * the sorted set in the standard output.
 *
 * Be careful to show the output as in the below example.
 *
 * An example of data read from standard input is the following (see file
 * Datasets/dataP0/princeton_5locations.p0in):
21.0 10.0
26.0 16.0
24.0 10.0
27.0 16.0
5
20.9 15.4 Frist Campus Center
23.7 14.8 Cannon Dial Elm
24.8 14.9 Quadrangle
25.6 14.9 Ivy
26.4 14.9 Cottage
 *
 * Running sintax:
 * > build/Fraud0 < <inputFile.p0in>
 *
 * Running example:
 * > build/Fraud0 < ../Datasets/dataP0/princeton_5locations.p0in
4
23.700000 14.800000 Cannon Dial Elm
26.400000 14.900000 Cottage
25.600000 14.900000 Ivy
24.800000 14.900000 Quadrangle
 */
int main(int argc, char *argv[]) {
    const int MAX_NLOCATIONS = 20;   // capacity of the location arrays
    Location bottomLeft1, topRight1, // coordinates that define the first input area
        bottomLeft2, topRight2;      // coordinates that define the second input area
    VectorLocation locations,        // VectorLocation with all the locations
        selectedLocations1,          // first vector of selected locations
        selectedLocations2;          // second vector of selected locations
    Location arrayLocations[MAX_NLOCATIONS], // array used to read all the input locations
        arrayUnionLocations[MAX_NLOCATIONS]; // array to store the resulting union
    int nLocs, // number of locations in arrayLocations
        nUnionLocs; // number of locations in arrayUnionLocations

    // Read bottomLeft1 and topRight1 from standard input
    // Read bottomLeft2 and topRight2 from standard input

    // Read from standard input all the locations and insert them in arrayLocations

    // Insert the locations in arrayLocations in the locations object

    // Take from the locations object the locations within each area and
    // insert them in selectedLocations1 and selectedLocations2

    // Calculate the union of selectedLocations1 and selectedLocations2

    // Sort the resulting VectorLocation object

    // Convert the sorted VectorLocation object to an array of Location objects

    // Print the resulting array of Location objects in the standard output
}
```