UNIVERSIDAD DE GRANADA

Práctica 2

Abstracción. Implementación de TDAs simples.

Estructura de Datos

Curso 2025/2026

Departamento de Ciencias de la Computación e I.A.
Universidad de Granada

30 de octubre de 2025

Índice

1 Introducción

En esta práctica vamos a implementar un sistema de gestión de eventos académicos que permitirá a los estudiantes organizar su calendario personal y detectar posibles solapamientos entre asignaturas, clases prácticas, seminarios, entregas u otros compromisos.

El objetivo principal es diseñar dos tipos de datos abstractos (TDA): uno que represente un evento individual con información relevante (nombre, día, hora de inicio y fin), y otro que gestione un conjunto de eventos de manera estructurada, permitiendo al usuario realizar operaciones como añadir, eliminar, buscar o listar eventos, así como detectar conflictos de horario entre ellos.

De esta forma, se desarrollará un sistema para gestionar eventos académicos, como clases, prácticas, seminarios o reuniones. El objetivo es representar y manipular este tipo de información mediante tipos de datos abstractos (TDA), permitiendo operaciones básicas de gestión y la detección de posibles solapamientos entre eventos. A lo largo de la práctica se reforzarán los conceptos de abstracción, encapsulamiento, especificación, representación e implementación de TDAs. Se prestará especial atención al diseño modular, al uso adecuado de estructuras dinámicas y a la correcta documentación del código. **No se permitirá el uso de contenedores de la STL.**

Objetivos y requisitos

Los objetivos de este guión de prácticas son los siguientes:

1. Asimilar los conceptos fundamentales de abstracción, aplicados al desarrollo de programas.
2. Diseñar y documentar un tipo de dato abstracto (T.D.A.) que represente entidades del mundo real.
3. Practicar con el uso de `doxygen` para documentar las operaciones públicas de un módulo.
4. Profundizar en los conceptos relacionados con la especificación, representación, función de abstracción e invariante de representación de un T.D.A.
5. Aplicar estos conocimientos al desarrollo de un sistema de gestión de eventos, prestando especial atención a la detección de solapamientos entre elementos del TDA.

Los requisitos para poder realizar esta práctica son:

1. Haber estudiado el Tema 1: Introducción a la eficiencia de los algoritmos.
2. Haber estudiado el Tema 2: Abstracción de datos.

2 Tipos de datos abstractos

Los tipos de datos abstractos (T.D.A.) permiten definir nuevos tipos que encapsulan tanto los datos como el conjunto de operaciones válidas sobre ellos, ocultando su representación interna. Esto permite desarrollar programas más modulares, reutilizables y fáciles de mantener.

En esta práctica, se trabajará con dos tipos de datos abstractos principales:

- Un T.D.A. **Evento**, que representará información básica sobre un evento académico: nombre, día, hora de inicio y hora de fin.
- Un T.D.A. **AgendaEventos**, que gestionará un conjunto de eventos, permitiendo su almacenamiento, recuperación, visualización y detección de solapamientos entre ellos.

Cada T.D.A. se implementará de forma modular, separando su especificación, representación e implementación. **Será necesario definir la función de abstracción y el invariante de representación para garantizar la validez de los objetos creados**.

2.1 Selección de operaciones

Una tarea fundamental en el desarrollo de un T.D.A. es la selección del conjunto de operaciones que lo definen. Estas operaciones deben responder a los problemas que se desean resolver con dicho tipo de dato, y deben cumplir las siguientes características:

1. Debe existir un conjunto mínimo de operaciones que garantice la abstracción.
2. Las operaciones seleccionadas deben ser las más utilizadas en los contextos de uso del T.D.A.
3. El diseño del conjunto de operaciones debe facilitar posibles modificaciones futuras del tipo de dato, evitando una implementación excesivamente rígida.

Las operaciones de un T.D.A. pueden clasificarse en dos grupos:

- **Operaciones fundamentales:** aquellas necesarias para construir, consultar o modificar el estado del tipo de dato, sin comprometer la abstracción.
- **Operaciones no fundamentales:** pueden derivarse de las anteriores y suelen implementarse como funciones auxiliares o de conveniencia.

2.2 TDAs a desarrollar

En esta práctica se desarrollarán los siguientes tipos de datos abstractos:

TDA Evento

Este tipo de dato representará un evento individual. Cada evento contendrá al menos:

- **Nombre**: una cadena que identifica el evento.
- **Día de la semana**: representado como un entero (por ejemplo, 1 = lunes, 2 = martes, etc.).
- **Hora de inicio y hora de fin**: representadas en formato de 24 horas (por ejemplo, 14 representa las 14:00).

Operaciones fundamentales:

- Crear un evento (constructor por parámetros).
- Obtener y modificar sus atributos.
- Comprobar si dos eventos se solapan (comparando día y franja horaria).

Operaciones no fundamentales:

- Mostrar por pantalla la información de un evento.
- Comparar eventos por nombre (por ejemplo, para ordenarlos o buscar duplicados).

TDA AgendaEventos

Este tipo de dato gestionará una colección de eventos. Se encargará de almacenar, buscar y mostrar los eventos registrados por el usuario, y de detectar posibles conflictos entre ellos.

Operaciones fundamentales:

- **Constructor por parámetros**: construye una agenda de eventos a partir de un fichero de texto. Cada línea del fichero contiene los datos de un evento, separados por comas.
- Añadir un nuevo evento a la agenda (si no se solapa con otro existente).
- Eliminar un evento dado su nombre.
- Comprobar si existe solapamiento entre eventos.
- Buscar eventos por nombre o por día.
- Mostrar todos los eventos almacenados.

Operaciones no fundamentales:

- Mostrar los eventos ordenados por día y hora.
- Detectar franjas horarias libres en un día determinado.
- Mostrar resumen semanal de la agenda.

Formato del fichero de entrada

El constructor por parámetros de la clase `AgendaEventos` recibirá como argumento un fichero de texto que contendrá la lista de eventos. Cada línea del fichero representará un evento individual, cuyos campos estarán separados por comas en el siguiente orden:

nombre del evento, día de la semana, hora de inicio, hora de fin

- El día de la semana se representará como un número entero del 1 al 7 (1 = lunes, 7 = domingo).
- Las horas se representarán como números en formato decimal, donde la parte decimal indica los minutos:
  - Por ejemplo: 9.0 = 09:00, 14.5 = 14:30, 15.25 = 15:15.
- No debe haber espacios entre campos, salvo dentro del nombre del evento si se desea.

A continuación se muestra un ejemplo de fichero de entrada válido:

Ejemplo de fichero `agenda.txt`

```
Clase de Matemáticas,1,9.0,11.0
Laboratorio de Física,1,11.0,13.0
Tutoría con María,2,10.0,11.0
Clase de Programación,3,9.0,11.0
Reunión de grupo,3,10.5,12.0
```

Este fichero incluye cinco eventos distribuidos entre lunes, martes y miércoles. El programa deberá detectar, por ejemplo, que hay solapamiento el miércoles entre las 10:30 y las 11:00.

3 Documentación

El objetivo fundamental de un programador es que los tipos de datos abstractos (T.D.A.) que desarrolle puedan ser reutilizados por él mismo o por otros programadores. Para que esto sea posible, los módulos donde se implementan los T.D.A. deben estar bien documentados. Esta documentación debe distinguir entre dos niveles:

1. **Especificación**: documento que presenta las características sintácticas y semánticas de la parte pública del T.D.A. (la visible desde otros módulos). Permite el uso del tipo sin conocer los detalles internos de su implementación.

2. **Implementación**: documento que describe las características internas del T.D.A. Es útil para facilitar modificaciones, detectar errores y mantener el código a lo largo del tiempo.

3.1 Definición del T.D.A.

El objetivo fundamental de un programador es que los T.D.A. que programe sean reutilizados en el futuro por él u otros programadores. Para que esta tarea puede llevarse a cabo los módulos donde se materializa un T.D.A. deben de estar bien documentados. Para llevar a cabo una buena documentación de T.D.A. se deben crear dos documentos bien diferenciados:

1. Especificación. Es el documento donde se presentan las características sintácticas y semánticas que describen la parte pública (la parte del T.D.A. visible a otros módulos). Con este documento cualquier otro módulo podría usar el módulo desarrollado y además es totalmente independiente de los detalle internos de construcción del mismo.

2. Implementación. Corresponden al documento que presenta las características internas del módulo. Para facilitar futuras mejoras o modificaciones de los detalles internos del módulo es necesario que la parte de implementación sea documentada.

Se dará una definición del T.D.A en lenguaje natural. Para ello el T.D.A se distinguirá como una nueva clase de objetos en la que cualquier instancia de esta nueva clase tomará valores (dominio) en un conjunto establecido. Por ejemplo, si queremos definir un Racional, diremos:

> *Una instancia f del tipo de dato abstracto Racional es un objeto del conjunto de los números racionales, compuesto por dos valores enteros que representan, respectivamente, numerador y denominador. Lo representamos num/den.*

3.2 Operaciones

Cada operación del T.D.A. representa una función, y su especificación debe incluir los siguientes elementos documentales. Se recomienda utilizar el sistema de anotaciones `doxygen`:

1. **@brief:** Breve descripción de la función.

2. **@param:** Explicación de cada parámetro, indicando si se modifica o no.

3. **@pre:** Condiciones que deben cumplirse antes de ejecutar la función.

4. **@return:** Valor devuelto por la función (si existe).

5. **@post:** Condiciones que deben cumplirse tras la ejecución.

Ejemplo de documentación de funciones:

```
/**
 * @brief Compara dos racionales.
 * @param r racional a comparar.
 * @return Devuelve 0 si este objeto es igual a r,
 *         <0 si este objeto es menor que r,
 *         >0 si este objeto es mayor que r.
 */
bool comparar(Racional r);

/**
 * @brief Asignación de un racional.
 * @param n numerador del racional a asignar.
 * @param d denominador del racional a asignar.
 * @return Asigna al objeto implícito el número racional n/d.
 * @pre d debe ser distinto de cero.
 */
void asignar(int n, int d);
```

3.3 Implementación del T.D.A.

Para implementar un T.D.A., es necesario en primer lugar escoger una representación interna adecuada, es decir, una forma de estructurar la información que permita representar correctamente y de forma eficaz todos los objetos del tipo abstracto.

A esta estructura se le denomina **tipo de representación** o *tipo rep*. Una vez seleccionado, las operaciones del T.D.A. se implementan sobre este tipo rep.

Ejemplo:

Supongamos que estamos implementando el T.D.A. `Racional`, que representa un número racional. Algunas posibles representaciones internas serían:

- Un vector de dos posiciones para almacenar numerador y denominador.
- Dos enteros individuales: `num` y `den`.

Una vez elegido el tipo rep, se deben especificar dos elementos fundamentales:

1. **Invariante de la representación:** conjunto de condiciones que deben cumplir los valores del tipo rep para ser considerados válidos. Por ejemplo: `den != 0`.
2. **Función de abstracción:** describe cómo se obtiene el valor abstracto a partir del tipo rep. Por ejemplo, el par `(num, den)` representa el racional $\frac{\texttt{num}}{\texttt{den}}$.

Propiedades de la función de abstracción:

- Es una función *parcial*: no todos los valores del tipo rep son válidos (por ejemplo, cuando `den` es cero).
- Cada objeto del tipo abstracto debe tener al menos una representación válida.
- Puede haber múltiples representaciones válidas para el mismo objeto abstracto (por ejemplo, `4/8` y `1/2`).

Estas descripciones deben incluirse en la documentación interna del fichero de implementación, usando las anotaciones adecuadas con `doxygen`.

4 Ejercicio

En esta práctica el alumno deberá desarrollar un programa que gestione información mediante tipos de datos abstractos. Para ello, se propone la implementación de dos T.D.A. que trabajen de forma conjunta: uno para representar entidades individuales y otro para gestionarlas de forma estructurada.

Los pasos a realizar son los siguientes:

1. Diseñar un tipo de dato abstracto que represente una entidad básica con varios atributos relevantes.
2. Diseñar un segundo tipo abstracto que almacene múltiples elementos del tipo anterior y permita su gestión: inserción, eliminación, búsqueda, etc.
3. Proporcionar operaciones adicionales que permitan analizar y consultar información derivada (por ejemplo, detectar repeticiones, calcular estadísticas, mostrar filtrado por atributos, etc.).

Para cada tipo de dato abstracto, el alumno deberá:

1. Establecer una **definición** clara en lenguaje natural.
2. Determinar y justificar diferentes opciones posibles para el **tipo de representación** (*tipo rep*).
3. Escoger una representación y justificar su elección.
4. Establecer el **invariante de representación** y la **función de abstracción**.
5. Implementar las operaciones necesarias de forma modular.
6. Documentar adecuadamente todo el código con `doxygen`.

Además, se deberá desarrollar un programa principal que pruebe todos los módulos y permita validar el funcionamiento de los T.D.A. implementados. Este programa debe construirse de forma flexible, permitiendo modificar los datos de entrada sin cambiar el código fuente.

Restricciones: No se permite el uso de estructuras de la STL (`vector`, `list`, etc.). Todas las estructuras de datos deberán implementarse manualmente.

4.1 Módulos a desarrollar

El desarrollo de esta práctica deberá estructurarse en varios módulos claramente diferenciados. Como mínimo, el alumno deberá implementar:

- Un módulo para el primer tipo de dato abstracto (`evento.h` y `evento.cpp`), que contendrá la definición y funcionalidad de las entidades individuales.
- Un módulo para el segundo tipo de dato abstracto (`agendaeventos.h` y `agendaeventos.cpp`), que gestionará un conjunto de objetos del tipo anterior.
- Un programa principal (`main.cpp`) que permita probar las funcionalidades desarrolladas.

Cada módulo deberá tener su propia cabecera (`.h`) y su implementación (`.cpp`). Es posible añadir módulos adicionales si se considera necesario para mejorar la organización del código o añadir funcionalidades extra.

Además, el proyecto deberá incluir un archivo `Makefile` para facilitar la compilación de todos los componentes de forma automatizada. La estructura del proyecto debe seguir el mismo modelo mostrado en clase.

Todos los ficheros fuente, incluyendo cabeceras, implementaciones, pruebas y documentación, deberán entregarse de forma organizada y limpia. No se incluirán ficheros objeto (`.o`), ejecutables ni carpetas innecesarias.

4.2 Compilación y Makefile

El proyecto deberá incluir un archivo `Makefile` que permita compilar automáticamente el código fuente y generar la documentación del proyecto. Las reglas mínimas exigidas son:

- `make`: compila el programa principal.
- `make clean`: elimina ficheros objeto y ejecutables.
- `make doc`: genera la documentación con `doxygen`, usando el fichero `doc/Doxyfile`.

Ejemplo de Makefile compatible con la estructura de carpetas propuesta:

```
# Carpetas
SRC    = src
INC    = include
OBJDIR = build
DOC    = doc
DATOS  = datos

# Archivos fuente
CPP    = $(SRC)/main.cpp $(SRC)/evento.cpp $(SRC)/agendaeventos.cpp
OBJ    = $(patsubst $(SRC)/%.cpp,$(OBJDIR)/%.o,$(CPP))

# Compilador y opciones
CXX    = g++
CXXFLAGS = -Wall -std=c++11 -I$(INC)

# Ejecutable
TARGET = programa

# Regla principal
all: $(TARGET)

$(TARGET): $(OBJ)
       $(CXX) $(CXXFLAGS) -o $(TARGET) $(OBJ)

$(OBJDIR)/%.o: $(SRC)/%.cpp | $(OBJDIR)
       $(CXX) $(CXXFLAGS) -c $< -o $@

$(OBJDIR):
       mkdir -p $(OBJDIR)

clean:
       rm -rf $(OBJDIR) $(TARGET)

doc:
       doxygen $(DOC)/Doxyfile
```

5 Entrega de la práctica

La entrega de la práctica se realizará a través de la plataforma de la asignatura, dentro de la actividad habilitada específicamente para ello. El alumno deberá comprimir todo el contenido

del proyecto en un único fichero con formato `.zip`, con el nombre:

`apellidos_nombre_p2.zip`

No se admitirán prácticas que no respeten esta estructura de entrega, ni aquellas que incluyan ficheros generados automáticamente como ejecutables, objetos o temporales.

Importante: Asegúrese de que la práctica se puede compilar correctamente ejecutando el comando:

```
make
```

y que se ejecuta sin errores con:

```
./programa
```

La práctica deberá estar correctamente documentada con `doxygen` y acompañada del correspondiente archivo `Makefile`.

5.1 Estructura de carpetas

El proyecto deberá organizarse siguiendo la siguiente estructura:

```
apellidos_nombre_p2/
|
|-- include/
|   |-- evento.h
|   |-- agendaeventos.h
|
|-- src/
|   |-- evento.cpp
|   |-- agendaeventos.cpp
|   |-- main.cpp
|
|-- build/               (carpeta generada automáticamente, no se entrega)
|
|-- datos/
|   |-- agenda.txt
|
|-- doc/
|   |-- Doxyfile
|   |-- html/            (generado automáticamente por doxygen)
|
|-- Makefile
|-- README.md (opcional)
```

- El código fuente debe organizarse en las carpetas `include/` y `src/`.
- La carpeta `datos/` debe incluir los ficheros de entrada necesarios para probar la práctica.
- La carpeta `doc/` debe contener el archivo de configuración `Doxyfile` y la documentación generada por `doxygen`.
- El `Makefile` debe compilar el proyecto desde esa estructura y permitir generar la documentación con `make doc`.
- Todos los archivos objeto (`.o`) se generarán en la carpeta `build/`. Estos archivos.o no deberán entregarse, se eliminarán con la regla `make clean`.

6 Rúbrica de evaluación

La práctica se calificará sobre un total de **10 puntos**, distribuidos según los siguientes criterios:

Criterio	Puntuación	Descripción
Implementación del T.D.A. Evento	2 puntos	- Correcta representación del TDA individual. - Métodos bien diseñados y funcionales. - Uso adecuado de tipos, memoria y comprobaciones.
Implementación del T.D.A. AgendaEventos	2 puntos	- Gestión correcta de colecciones de eventos. - Detección de solapamientos. - Métodos de búsqueda, inserción y visualización bien implementados.
Organización y estructura del código	1.5 puntos	- Separación en módulos `include/`, `src/`, etc. - Código legible, estructurado y limpio. - Uso correcto del `Makefile`.
Documentación técnica del código y T.D.A.	2 puntos	- Uso correcto de `doxygen` (`@brief`, `@param`, etc.). - Inclusión del invariante de representación y función de abstracción. - Generación de documentación en HTML.
Programa principal de pruebas	1 punto	- Incluye un `main.cpp` funcional. - Permite cargar datos, probar métodos y mostrar resultados.
Estructura de proyecto y entrega	1 punto	- Proyecto organizado correctamente. - Entregado comprimido en un único fichero `.zip`. - Estructura de carpetas respetada.
Calidad técnica y restricciones	0.5 puntos	- No uso de STL. - Código eficiente, sin errores de compilación o ejecución. - Uso adecuado de memoria dinámica.

Cuadro 1: Rúbrica de evaluación de la práctica