UNIVERSIDAD DE GRANADA

Ciencias de la Computación e Inteligencia Artificial

**Metodología de la Programación (2020/21)**
**1º GII / GII-M / GII-ADE**
**Conv. Extraordinaria - 14 de Julio de 2021**

**Normas para la realización del examen:** **Duración: 2 horas**

- **El único material permitido durante la realización del examen es un lápiz o bolígrafo**
- **No olvide escribir su nombre completo y grupo en todos y cada uno de los folios que entregue.**

Vamos a trabajar sobre una versión simplificada de la gestión de las clasificaciones de los participantes en una carrera popular.

Para poder participar en una carrera es necesario inscribirse previamente. En el proceso de inscripción se registran los datos personales y se asigna un número de dorsal (éste es único y mayor que cero). Los datos de los inscritos a una carrera se guardan en un fichero de texto. Para disponer en un programa de los datos de los inscritos se propone la siguiente estructura básica para las clases que permiten modelizar una colección de inscritos:

```
class Inscrito {
private:
  int dorsal;
  string apellidosNombre;
  Fecha fechaNacimiento;
  char sexo;
public:
  ...
};
```

```
class Inscritos {
private:
  // Núm.casillas disponibles (capacidad) por defecto
 static const int TAM_INICIAL = 5;
 // Núm.casillas adicionales por redimensionamiento
  static const int TAM_BLOQUE = 2;

  Inscrito * inscritos;
  int capacidad;  // PRE: 0 <= capacidad
  int utilizados; // PRE: 0 <= utilizados <= capacidad
public:
  ...
};
```

Suponga que dispone de la clase **Fecha*** completa, mientras que de la clase **Inscrito** dispone de los siguientes métodos públicos:

```
Inscrito();
Inscrito(int elDorsal, const string & elNombre, const Fecha & laFechaNac, char elSexo);
int    getDorsal() const;

string getApellNbre() const;
Fecha  getFechaNac() const;
char   getSexo() const;

void   setApellNbre(const string & elNombre);
void   setFechaNac(const Fecha & laFechaNac);
void   setSexo(char elSexo);
```

y de los operadores de inserción y extracción en/de flujo (<< y >>). El constructor sin argumentos de la clase **Inscrito** crea el *corredor con dorsal 0* que estará presente en todos los objetos de la clase **Inscritos**, y siempre ocupará la casilla 0.

Suponga también que dispone de la implementación en la clase **Inscritos** de los siguientes métodos públicos:

```
Inscritos(const string & nombre);
int  numInscritos() const;
```

El constructor que recibe un dato `string` construye un objeto **Inscritos** tomando los datos de un fichero de texto. Dispone, además, de los operadores combinados `+=` (para añadir un objeto **Inscrito**) y `-=` (para eliminar un objeto **Inscrito**, dado su dorsal) y, finalmente, de los operadores de inserción y extracción en/de flujo (<< y >>).

**Muy importante:** No tiene que implementar los métodos que indicamos que están *disponibles* en las clases **Inscrito** e **Inscritos**. Cualquier otro método/función cuya disponibilidad no se indique explícitamente deberá escribirse, al igual que los métodos/funciones auxiliares que se precisaran.

---

### ◁ Ejercicio 1 ▷ Métodos de la clase Inscritos [1.5 puntos]

1. *(0.75 puntos)* **Métodos básicos**. Constructor con reserva, constructor de copia, operador de asignación y destructor. El constructor con reserva de espacio tendrá el prototipo: `Inscritos (int laCapacidad=TAM_INICIAL);`
2. *(0.25 puntos)* **Operadores de acceso por posición**. Definir el operador `[]` para la clase **Inscritos**. El acceso se realiza por posición. La convención que se sigue acerca de las *posiciones* es que se numeran desde 1: *inscrito 1*, *inscrito 2*, ....
3. *(0.5 puntos)* **Operadores de acceso por dorsal**. Definir el operador `()` para la clase **Inscritos** de manera que la clave de acceso sea el **número de dorsal**. Si el dorsal no existe, o fuera cero, debe acceder al *corredor con dorsal 0*.

---

Cuando se celebra la carrera los *inscritos* que toman la salida se convierten en *participantes*. Para la gestión de clasificaciones no es preciso tener la información personal de los participantes (suponemos que ya están disponibles en un objeto **Inscritos**). De hecho solo nos interesa el dorsal (a partir de él podremos acceder a los datos personales de cada participante) y la hora de salida y llegada (para calcular el tiempo en carrera).

Las clases para la gestión de participantes son **Participante** y **Participantes**, cuya estructura básica es:

```
class Participante {
private:
  int dorsal;
  Hora horaSalida;
  Hora horaLlegada;
public:
  ...
};
```

```
class Participantes {
private:
   // Núm.casillas disponibles (capacidad) por defecto
   static const int TAM_INICIAL_PARTICIPANTES = 5;
   // Núm.casillas adicionales por redimensionamiento
   static const int TAM_BLOQUE_PARTICIPANTES = 2;
   Participante * participantes;
   int capacidad; // PRE: 0 <= capacidad
   int utilizados; // PRE: 0 <= utilizados <= capacidad
public:
   ...
```

Suponga que dispone de la clase **Hora*** completa, mientras que de la clase **Participante** dispone de los siguientes métodos públicos:

```
Participante(int elDorsal=0, const Hora & laSalida=HORANULA, const Hora & Hora laLlegada=HORANULA);
int  getDorsal() const;                    void setDorsal(int elDorsal);
Hora getHoraSalida() const;                void setHoraSalida(const Hora & laHora);
Hora getHoraLlegada() const;               void setHoraLlegada(const Hora & laHora);
Hora getTiempoEnCarrera() const; // Calcula el tiempo total entre la hora de llegada y salida
```

y de los operadores de inserción y extracción en/de flujo (<< y >>).

Suponga que dispone de la implementación en la clase **Participantes** de los siguientes métodos públicos:

```
Participantes(int laCapacidad=TAM_INICIAL_PARTICIPANTES);
int  numParticipantes() const;
```

y del constructor de copia, operador de asignación y destructor. Dispone también de los operadores de inserción y extracción en/de flujo (<< y >>) y del operador -sobrecargado- de acceso *por posición* **[]**. La convención que se sigue acerca de las *posiciones* es que se numeran desde 1: *participante 1*, *participante 2*, ...).

**Muy importante:** No tiene que implementar los métodos que indicamos que están *disponibles* en las clases **Participante** y **Participantes**. Cualquier otro método/función cuya disponibilidad no se indique explícitamente deberá escribirse, al igual que los métodos/funciones auxiliares que se precisaran.

---

### ◁ Ejercicio 2 ▷ Sobrecarga de operadores += y -= en Participantes [2 puntos]

1. *(1 punto)* El operador += añade (al final) un dato **Participante**.
2. *(1 punto)* El operador -= elimina un dato **Participante**. Se usará como argumento el *número de dorsal*. Si el participante indicado no está en la colección, no se hace nada.

---

### ◁ Ejercicio 3 ▷ Ordenación de un objeto Participantes [1.5 puntos]

Escribir el método `void ordenar()` para ordenar "in situ" el contenido de un objeto **Participantes** usando el algoritmo de selección. Para este ejercicio la ordenación será creciente, y el criterio será el *tiempo empleado en la carrera*.

---

### ◁ Ejercicio 4 ▷ Aplicación - Listado de inscritos según año de nacimiento [1 punto]

Escribir un programa que reciba desde la línea de órdenes dos años y el nombre de un fichero de inscritos. El programa mostrará los datos de los inscritos cuyo año de nacimiento esté comprendido entre los dos años indicados. **Importante:** Los años indicados no pueden ser menores que cien años menos que el año actual ni mayores que tres años menos que el año actual.

Por ejemplo: `selecciona_inscritos 1975 1995 inscritos_PRUEBA_FONDO_MP`

mostrará los inscritos cuyos datos están en `inscritos_PRUEBA_FONDO_MP` nacidos entre 1975 y 1995.

Considere y controle todas las situaciones de error. Suponga que el fichero de inscritos es correcto.

---

*Sobre las clases `Fecha` y `Hora`:

- Clase **Fecha**: El constructor sin argumentos establece la *fecha actual*. Dispone además de los métodos set/get típicos (`setDia`, `getDia`, `setMes`, etc.), los operadores de inserción y extracción en/de flujo (<< y >>) y el método `toString`.
- Clase **Hora**: El constructor sin argumentos establece la *hora actual*. Dispone además de los métodos set/get típicos (`setHora`, `getHora`, `setMinuto`, etc.), los operadores de inserción y extracción en/de flujo (<< y >>), el método `toString` y los operadores relacionales (<, <=, etc).