Examen de Metodología de la Programación
Curso 2020-2021, Convocatoria Febrero

Sección Informativa: **Duración del examen: 2 horas 30 minutos**

1- (6 Puntos). C++.

1) Dada la siguiente clase genérica:

```
class Fecha {
  int dia;
  int mes;
  int anio;
public:
  Fecha(int d, int m, int a);

};
```

```
template <class T>
class Jugador {
  const int dni;
  Fecha fecha;
  T id;
public:
  Jugador(int d, Fecha f, T id);
  ~Jugador();
};
```

Indica exclusivamente cómo sería el código del constructor y del destructor para que sirva y funcione correctamente para los tipos <**int**>, <**float**>, <**char**> y <**char***> especializando aquellos métodos que consideres necesarios. A partir de Jugador crea una clase derivada JugadorPlus que tenga un atributo **const int edad** e indica cómo sería su constructor y su destructor. **(1.50 puntos)**

SOLUCION:
La plantilla sirve para **int**, **float** y **char**. Para **char *** hay que realizar una especialización ya que debemos reservar memoria con **new** en el constructor y liberarla en el destructor con **delete**.

Aunque en la clase `Fecha` solo tenemos un constructor y no tenemos el de copia, el que genera de oficio el compilador (al no implementarlo nosotros) funciona perfectamente ya que la clase no tiene memoria dinámica y por tanto lo podemos utilizar.

Como en la clase jugador el atributo `dni` es constante lo tenemos que poner obligatoriamente en la zona de inicializadores del constructor. Lo mismo ocurre con el atributo `fecha` ya que es un objeto de otra clase y hay que invocar su constructor en la zona de inicializadores. Por último el destructor de Jugador lo debemos declarar **virtual**

```
template <class T> Jugador<T>::Jugador(int d, Fecha f, T id):dni(d), fecha(f) {  //0.20 puntos
    this->id=id;
}

//especialización para char*                                                    //0.40 puntos
template <> Jugador<char*>::Jugador(int d, Fecha f, char *id):dni(d), fecha(f) {
    this->id=new char[strlen(id)+1];
    strcpy(this->id,id);
}

template <class T> Jugador<T>::~Jugador() { } //no hace nada                    //0.10 puntos

//especialización para char*                                                    //0.30 puntos
template <> Jugador<char*>::~Jugador() {
    delete [] id;
}

template <class T>
class JugadorPlus: public Jugador<T> {                                         //0.10 puntos
  const int edad;
public:
  JugadorPlus(int d, Fecha f, T id, int e):Jugador<T>(d,f,id), edad(e) {  }    //0.30 puntos
  ~JugadorPlus() { } //no hace nada                                            //0.10 puntos
};
```

Consideraciones a tener en cuenta

Algunos erróneamente no especializan los métodos sino que hacen lo siguiente, pensando que el compilador chequea unas líneas u otras en tiempo de compilación, cuando no es así. Al ejecutarse el programa (en tiempo de ejecución) es cuando se ejecutan unas líneas u otras en función de si el if es cierto o no:

```
template <class T> Jugador<T>::Jugador(int d, Fecha f, T id):dni(d), fecha(f) {
    if (typeid(T)==typeid(int) || typeid(T)==typeid(float) || typeid(T)==typeid(char)) {
        this->id=id;
    }
    else if (typeid(T)==typeid(char *)) { //da error al compilar
        this->id=new char[strlen(id)+1];  //los tipos int, float y char
        strcpy(this->id,id);              //ya que strlen y strcpy solo se aplica a char *
    }
}

template <class T> Jugador<T>::~Jugador() {
    if (typeid(T)==typeid(char *)) //da error al compilar
        delete [] id;              //delete espera un puntero y int, float y char no lo son
}
```

Esto da error al compilar ya que al no especializar el compilador detecta que hay algunas líneas que no sirven para todos los tipos a los que vamos a instanciar el tipo genérico. Es decir, el compilador comprueba que todas las líneas de código sirvan para los tipos **int**, **float, char** y **char *** (que son a los que instanciamos el tipo genérico **T**), y al encontrar alguna líneas que no sirve para alguno de esos tipos lanza un error.

Otras consideraciones

Si un método no es necesario para unos tipos concretos y hay que especializarlo para otros, el prototipo de dicho método hay que declararlo en la clase y hay que implementarlo tanto para los tipos que no son necesarios como para los tipos a los que hay que especializar.

A modo de ejemplo, si tuviéramos que hacer también el constructor de copia veríamos que para los tipos **int**, **float, char** no es necesario ya que el que genera de oficio el compilador funciona, pero como tenemos que especializarlo para el tipo **char *** entonces tendremos que declarar el prototipo en el .h e implementar en el .cpp tanto para unos como para otros.

```
template <class T>
class Jugador {
  const int dni;
  Fecha fecha;
  T id;
public:
  Jugador(int d, Fecha f, T id);
  virtual ~Jugador();
  Jugador (const Jugador<T> &j); //hay que ponerlo ya que vamos a especializarlo
};                               //el compilador no lo genera de oficio porque lo hemos puesto
```

```
template <class T> Jugador<T>::Jugador(int d, Fecha f, T id):dni(d), fecha(f) {
    this->id=id;
}

template <> Jugador<char*>::Jugador(int d, Fecha f, char *id):dni(d), fecha(f) {
    this->id=new char[strlen(id)+1];  //especialización para char*
    strcpy(this->id,id);
}

template <class T> Jugador<T>::~Jugador() { } //no hace nada

template <> Jugador<char*>::~Jugador() { delete [] id; } //especialización para char*

//al poner el prototipo en el .h tenemos que implementarlo ya que el compilador ya no lo hará
template <class T> Jugador<T>::Jugador(const Jugador<T> &j):dni(j.dni), fecha(j.fecha) {
    id=j.id;
}

template <> Jugador<char*>::Jugador(const Jugador<char *> &j) :dni(j.dni), fecha(j.fecha) {
    this->id=new char[strlen(j.id)+1]; //especialización para char*
    strcpy(this->id,j.id);
}
```

Otra posibilidad es en vez de especializar métodos para un tipo concreto especializar la clase para ese tipo concreto:

Creamos una clase genérica para los tipos **int**, **float, char** en la que solo hace falta implementar el constructor ya que el destructor no es necesario y el constructor de copia que genera el compilador (si nosotros no lo ponemos) funciona bien al no haber memoria dinámica

```
template <class T>
class Jugador {      //clase generica para los tipos int, float, char
  const int dni;
  Fecha fecha;
  T id;
public:
  Jugador(int d, Fecha f, T id); //solo hace falta implementar el constructor
  //virtual ~Jugador(); //no hace falta siquiera ponerlo
};                      //si lo ponemos tenemos que implementarlo  OJO!!!
```

```
template <class T> Jugador<T>::Jugador(int d, Fecha f, T id):dni(d), fecha(f) {
    this->id=id;
}

//template <class T> Jugador<T>::~Jugador() { } //no hace nada     por eso no lo implementamos
                                                //al no haberlo puesto en el .h
```

Especializamos la clase para el tipo **char *** e implementamos el destructor (para liberar memoria) y el constructor de copia (ya que el que genera el compilador si nosotros no lo ponemos en el .h) no funciona bien al haber memoria dinámica

```
template <>
class Jugador<char *> { //especializacion de la clase para char *
  const int dni;
  Fecha fecha;
  char * id;
public:
  Jugador(int d, Fecha f, char * id);
  virtual ~Jugador();                       //hay que implementar el destructor
  Jugador (const Jugador<char *> &j);       //y el constructor de copia
};
```

```
template <> Jugador<char*>::Jugador(int d, Fecha f, char *id):dni(d), fecha(f) {
    this->id=new char[strlen(id)+1];
    strcpy(this->id,id);
}

template <> Jugador<char*>::~Jugador() { delete [] id; }

template <> Jugador<char*>::Jugador(const Jugador<char *> &j) :dni(j.dni), fecha(j.fecha) {
    this->id=new char[strlen(j.id)+1];
    strcpy(this->id,j.id);
}
```

Observe que no hay que poner `template <>` al implementar en el .cpp la clase especializada

`template <>` se pone cuando se va a especializar un método de una clase, pero no cuando se va a implementar un método de una clase especializada

2) Dado el siguiente código:

```
class Error: public exception {
  char *mensaje;
public:
  Error(char *m): exception() {
    mensaje=new char[strlen(m)+1];
    strcpy(mensaje, m);
  }
  const char *what() const throw() { return mensaje;  }
};

class Fecha { //clase inmutable
  int dia, anio;
  char *mes;
  static int diasValidos[];
  static const char* Validos [];
public:
  Fecha(int d, char* m, int a) {  dia=d; mes=m; anio=a; }
  int getDia() { return this->dia; }
  char* getMes() { return this->mes; }
  int getAnio() { return this->anio; }
};

int Fecha::diasValidos[]={31,28,31,30,31,30,31,31,30,31,30,31};
const char *Fecha::mesesValidos[]={"ene", "feb", "mar", "abr", "may", "jun", "jul", "ago",
                                   "sep", "oct", "nov", "dic"};
```

Corrige los errores de la clase Fecha y haz que sea robusta, segura y sirva para cualquier main( ) posible. La clase Fecha solo debe permitir crear fechas válidas. Si la fecha no es válida debe lanzar una excepción indicando si el error está en el mes o en el día (lanza la primera que se produzca, en ese orden).
Para simplificar las cosas suponed que ningún año es bisiesto y que una vez creada una fecha solo podemos modificarla asignándole el valor de otra fecha ya existente. **(2.00 puntos)**

SOLUCION:
Como la clase tiene memoria dinámica debemos implementar además del constructor, el destructor, constructor de copia y operador de asignación

```
class Fecha {
  ...
protected:
  int buscarMes(char *mes) const { //-1 si no lo encuentra                    //0.30 puntos
    for (int i=0;i<12;i++)
      if (strcmp(mes, mesesValidos[i])==0)
        return i;
    return -1;
  }

public:
  Fecha(int d, char* m, int a) throw (Error) {                                //0.40 puntos
    int im=buscarMes(m);
    if (im==-1) throw Error("mes no valido");
    if (d<1 || d>diasValidos[im]) throw Error("dia no valido");
    mes=new char[strlen(m)+1]; strcpy(mes, m);
    dia=d; anio=a;
  }

  Fecha::~Fecha () { delete [] mes; }                                         //0.20 puntos

  int getDia() const { return this->dia; }                                    //0.10 puntos
  const char* getMes() const { return this->mes; }                            //0.20 puntos
  int getAnio() const { return this->anio; }                                  //0.10 puntos

  Fecha &operator=(const Fecha &f) {                                          //0.40 puntos
    if (this!=&f) {
      this->dia=f.dia;       this->anio=f.anio;
      delete [] this->mes;
      this->mes=new char[strlen(f.mes)+1];
      strcpy(this->mes, f.mes);
    }
    return *this;
  }

  Fecha(const Fecha& f) {                                                     //0.30 puntos
    this->dia=f.dia;         this->anio=f.anio;
    this->mes=new char[strlen(f.mes)+1];
    strcpy(this->mes, f.mes);
  }
};
```

3) Dado el siguiente código:

```
class Fecha {
  int dia, anio;
  char *mes;
public:
  Fecha(int d, char* m, int a)
  ...
};

class Cliente {
  char *nombre, *dni;
  Fecha fecha;
public:
  Cliente(const char *d, const char *nom, Fecha f);
  Cliente(const Cliente& c);
  ...
};
```

```
class ClienteVIP: public Cliente {
  ...
};

class ClienteNOVIP: public Cliente {
  ...
};

class Empresa {
  Cliente **clientes;
  int ncli, nmaxcli;
public:
  Empresa();
  void alta(Cliente *c); //implementado
  void bajaCliente(char *dni);
  ...
};
```

Corrige los errores del método bajaClientesVIPFecha( ) e implementa todos los métodos necesarios en cada clase respectiva para que dicho método pueda funcionar (se deben corregir todos los errores de dicho método e implementar exclusivamente aquellos métodos y operadores que son llamados desde dentro de dicho método). **(2.50 puntos)**

```
void Empresa::bajaClientesVIPFecha(Empresa e, Fecha f) {
   for(i=0; i<100; i++) {
     if (clientes[i]->fecha==f) {
       bajaCliente(clientes[i].dni); //al eliminar se corren los siguientes hacia atras
     }
   }
}

int main() {
  Fecha f1(2, "ene", 2017);
  Empresa ING, BBVA;
  ING.alta(new ClienteVIP("juan", ...));
  ING.alta(new ClienteNOVIP("eva", ...));
  ...
  Empresa::bajaClientesVIPFecha(ING, f1); //elimina de ING y de la memoria los Clientes VIP
}                                         //dados de alta la fecha f1
```

SOLUCION:

```
class Fecha {
  ...
  bool operator==(const Fecha &f) const {                                    //0.30 puntos
    return (dia==f.dia && strcmp(mes,f.mes)==0 && anio==f.anio);
  }
}

class Cliente {
  ...
  Fecha getFecha() const { return this->fecha; }                             //0.10 puntos
  const char *getDni() { return this->dni; }                                 //0.10 puntos
}

class Empresa {
  ...
  void bajaCliente(char *dni);
  static void bajaClientesVIPFecha(Empresa &e, Fecha f);
};

void Empresa::bajaClientesVIPFecha(Empresa &e, Fecha f) {                    //1.00 puntos
   for(int i=0; i<e.ncli; i++) {
     if (e.clientes[i]->getFecha()==f && typeid(*e.clientes[i])==typeid(ClienteVIP)) {
        e.bajaCliente(e.clientes[i]->getDni()); //al eliminar se corren los siguientes hacia atras
       i--;
     }
   }
}

void Empresa::bajaCliente(char *dni) {                                       //1.00 puntos
  for(int i=0; i<this->ncli; i++) {
     if (strcmp(this->clientes[i]->getDni(),dni)==0) {
       delete this->clientes[i];           //liberamos la memoria usada por el cliente borrado
       for(int j=i+1;j<this->ncli;j++)     //al eliminar se corren los siguientes hacia atras
         this->clientes[j-1]=this->clientes[j];
       this->ncli--;
       break; //rompemos el bucle para no seguir buscando
     }
  }
}
```

2- (4 Puntos). Java:

1) Indica los posibles errores del siguiente código justificando el motivo: **(1.00 puntos)**

```
public class A {
  private final int at = 5;

  public static int met() { return at; }
  public void proceso() { System.out.println(at); at++; }
  public final void saludo() {  System.out.println("hola"); }
}

public class B extends A {
  public void saludo() { System.out.println(super.saludo()+" bienvenido"); }

}
```

SOLUCION:

Error 1: public static int met( ) { return at; } //0.25 puntos
no se puede accede a una variable miembro desde un método estático.

Error 2: at++ //0.25 puntos
at es una variable final, no puede ser modificado su valor.

Error 3: public final void saludo( ) //0.25 puntos
Saludo es un método final en A y por tanto no se puede sobreescribir en la clase hija.

Error 4: super.saludo( ) //0.25 puntos
Saludo( ) en A no devuelve nada (void) y por tanto no se puede poner dentro de un println ni concadenarlo con +.

2) ¿Cuál será el resultado de la ejecución del método *main*? Justifica la respuesta: **(0.75 puntos)**

```
public class HiloEjecucion {
  public static int[] datos = new int[3];

  public static String met(int i) {
    String salida = "";
    salida = salida + datos[++i];
    try {
      salida += datos[i+1];
      salida += "OK";
    } catch (Exception e){
      salida += "Excepcion";
    } finally {
      salida += "Finally";
    }
    salida += "--";
    return salida;
  }

  public static void main(String [] args) {
    System.out.println(met(-1));                    //0.25 puntos
    System.out.println(met(1));                     //0.25 puntos
    System.out.println(met(2));                     //0.25 puntos
  }
}
```

SOLUCION:
Salida:

```
00okFinally--
0ExcepcionFinally--
Aborta el programa ya que se produce una excepción antes de entrar en el
try catch
```

3) Dado la siguiente clase implementa los métodos **estrictamente necesarios** para que el main( ) que aparece a continuación pueda ejecutarse y produzca la salida indicada: **(2.25 puntos)**

```
package libClases;

public class Persona implements Cloneable {
  private int edad;
  private int [] telefonos; //capacidad inicial 2 ampliable dinámicamente
  private int n;
  private String nombre;

  public Persona(String nombre, int e) {
    this.nombre = nombre;
    edad = e;
    n = 0;
    telefonos = new int[2];
  }

  public void setEdad(int e) { edad=e;      }

  //A RELLENAR POR EL ALUMNO
}
```

```
package libPruebas;

import libClases.*;

public class Prueba1 {
  public static void main(String[] args) {
    Persona a=new Persona("juan", 23), b=(Persona)a.clone();
    a.setEdad(46);
    System.out.println(a);
    a.agregar(959217388); a.agregar(959217373);
    Persona c=(Persona)a.clone();
    c.agregar(959444444);
    System.out.println(a);
    System.out.println(b);
    System.out.println(c);
  }
}
```

Salida:

```
juan tiene 46 años y 0 telefonos                                                    //a
juan tiene 46 años y 2 telefonos ( 959217388 959217373 )                            //a
juan tiene 23 años y 0 telefonos                                                    //b
juan tiene 46 años y 3 telefonos ( 959217388 959217373 959444444 )                  //c
```

SOLUCION:

Persona.java **(2.25 puntos)**

```
package libClases;

public class Persona implements Cloneable {
   ...
   public String toString() {                                    //0.50 puntos
      String s=nombre + " tiene " + edad + " años y " + n + " telefonos ";
      if (n>0) {
         s=s+ "( ";
         for(int i=0;i<n;i++)
            s=s+telefonos[i]+" ";
         s=s+ ")";
      }
      return s;
   }

   public void agregar(int num) {                                //0.75 puntos
      if (n==telefonos.length) {
         int [] aux=telefonos;
         telefonos=new int[n*2]; //duplica su tamaño
         for(int i=0;i<n;i++)
            telefonos[i]=aux[i];
      }
      telefonos[n]=num;
      n++;
   }
```

```
   public Object clone() {                                          //1.00 puntos
      Persona obj=null;
      try {
         obj=(Persona) super.clone();
//los atributos de tipos primitivos y de clases inmutables no hay que clonarlos
//los atributos de clases mutables y los atributos arrays si hay que clonarlos
         obj.telefonos=(int[])obj.telefonos.clone(); //array es mutable cloneable -> clone()
       //nombre es un String y no tiene clone() al ser inmutable
      } catch(CloneNotSupportedException ex) {
         System.out.println(" no se puede duplicar");
      }
      return obj;
   }


Una solución alternativa para el método clone puede ser la siguiente:

   public Persona(Persona p) {
      nombre = p.nombre;
      edad=p.edad;
      n=p.n
      telefonos = (int [])p.telefonos.clone();
    //telefonos = new int[p.telefonos.length];
    //for(int i=0; i<n; i++)
    //   telefonos[i]=p.telefonos[i];
   }


   public Object clone() {
      return new Persona (this);
   }
```

El método **clone( )** en los arrays crea un nuevo array del mismo tamaño que el original realizando una copia superficial de los elementos del array original (es decir, los elementos del array clonado referencian o apuntan a los mismos elementos del array original).

Cuando los elementos son de tipo primitivo o de clases inmutables esto no tiene importancia ya que los objetos a los que apunta cada elemento del array no se puede modificar.

Pero cuando los elementos del array son de clases mutables, cualquier cambio que hagamos en uno de los elementos del array se verá reflejando en ambos arrays (original y clonado) ya que en ambos los elementos referencian (apuntan) a los mismos objetos.

Para hacer que el array clonado sea independiente del array original, aparte de clonar el array debemos clonar cada uno de los elementos.

En nuestro caso, como el array es de tipo int no hace falta que clonemos cada elemento del array (nos basta con clonar el array)