PRÁCTICA 8ª: Plantillas y excepciones.

OBJETIVOS: Repaso de todo lo anterior. Uso de plantillas y de excepciones.

TEMPORIZACIÓN:
Publicación del enunciado: Semana del 2 de diciembre.
Entrega: Semana del 16 de diciembre.
Límite de entrega (con penalización): Semana del 16 de diciembre.

BIBLIOGRAFÍA
Programación orientada a objetos con C++
Autor: Fco. Javier Ceballos
Editorial: RA-MA.

Una empresa de seguros de reparaciones de electrodomésticos desea una aplicación que automatice sus procesos de gestión utilizando una metodología orientada a objetos. La empresa tiene dos departamentos: comercial y técnico. El departamento comercial se encarga de gestionar los contratos de los clientes tras la compra de electrodomésticos así como de estimar si es rentable un seguro, y el departamento técnico se encarga de contabilizar y atender las averías (siniestros) que pudieran ocurrir durante el periodo en vigor del contrato, así como de elaborar presupuestos de reparación.

Una vez analizada la forma de operar de la empresa, se observa que cada cliente podrá tener varios contratos de seguro ligados a distintos electrodomésticos adquiridos, y a su vez es necesario llevar un registro de los siniestros sufridos por cada electrodoméstico durante la vigencia del contrato de seguro.

Para crear la aplicación de gestión, se ha ideado la siguiente jerarquía de clases:

La clase `CCliente` representa a un cliente de la compañía y tiene la siguiente estructura:

- `m_Nombre`: almacena el nombre y apellidos del cliente.
- `m_Contratos`: lista de contratos de un cliente.
- `SetNombre()`: función miembro que establece el nombre del cliente.
- `GetNombre()`: función miembro que obtiene el nombre del cliente.
- `AgregarContrato()`: función miembro que añade un contrato al cliente.

```
class CCliente
{
  private:
    string m_Nombre;
    CLista<CContrato> m_Contratos;
  public:
    CCliente(const string& Nom="Sin Nombre") : m_Nombre(Nom) {};
    CCliente(const string& Nom, const CContrato& c);
    void SetNombre(const string& Nom) { m_Nombre = Nom; };
    string GetNombre() const { return m_Nombre; }
    void AgregarContrato(const CContrato& c);
};
```

La clase `CContrato` representa un contrato de seguro de un electrodoméstico concreto, y tiene la siguiente estructura:

- `m_NumSerie`: número de serie del electrodoméstico.
- `m_Descripcion`: descripción del electrodoméstico.
- `m_FechaFin`: fecha de finalización del contrato.
- `m_Poliza`: valor de la póliza del contrato.
- `m_ValorCompra`: valor de compra del electrodoméstico.
- `m_Siniestros`: lista de siniestros sufridos por el electrodoméstico.
- `Get/SetNumSerie()`: obtiene/establece el número de serie del electrodoméstico.
- `Get/SetDescripcion()`: obtiene/establece la descripción del electrodoméstico.
- `Get/SetFechaFin()`: obtiene/establece la fecha de finalización del contrato.
- `Get/SetPoliza()`: obtiene/establece el valor de la póliza del contrato.
- `Get/SetValorCom()`: función miembro que obtiene/establece el valor de compra del electrodoméstico.
- `AgregarSiniestro()`: añade un nuevo siniestro a la lista.

```
class CContrato
{
  private:
    int    m_NumSerie;
    string m_Descripcion;
    string m_FechaFin;
    long   m_Poliza;
    long   m_ValorCompra;
    CLista<CSiniestro *> m_Siniestros;
  public:
    CContrato(int NumSerie, const string& Desc="",
     const string& FF="", long Poliza = 0, long ValorCompra = 0);
    CContrato(const CContrato& c);
    CContrato& operator=(const CContrato& c);
    ~CContrato();

    void SetNumSerie(int ns) { m_NumSerie = ns; }
    void SetDescripcion(const string& d) { m_Descripcion = d; }
    void SetFechaFin(const string& ff) { m_FechaFin = ff; }
    void SetPoliza(long p) { m_Poliza = p; }
    void SetValorCom(long vc) { m_ValorCompra = vc; }

    int GetNumSerie() const { return m_NumSerie; }
    string GetDescripcion() const { return m_Descripcion; }
    string GetFechaFin() const { return m_FechaFin; }
    long GetPoliza() const { return m_Poliza ;}
    long GetValorCom() const { return m_ValorCompra ;}

    void AgregarSiniestro(CSiniestro& s);
};
```

La clase `CSiniestro` representa un siniestro genérico que se ha producido en un electrodoméstico. Todo siniestro tiene un *coste* asociado de reparación para la empresa y un *presupuesto* asociado para el cliente, ambos dependientes del tipo de siniestro. Esta

clase no se empleará para describir siniestros como tales, sino como clase base para tipos concretos de siniestro. Tiene la siguiente estructura:

- `m_Codigo`: código de identificación de la avería (o siniestro).
- `m_SigCodigo`: código que se asignará automáticamente al siguiente siniestro que se cree. Esta variable `static` se incrementa automáticamente cada vez que se crea un siniestro.
- `m_Descripcion`: descripción de la avería.
- `m_HorasMO`: horas de mano de obra de reparación.
- `m_CostePiezas`: coste de las piezas de la reparación.
- `m_Coste`: coste total de la reparación para la empresa.
- `GetPresupuesto()`: devuelve el presupuesto de la reparación para el cliente, que se calculará en función del tipo de siniestro y el coste para la empresa.
- `GetCodigo()`: devuelve el código del siniestro.
- `GetCoste()`: devuelve el coste del siniestro para la empresa.
- `Presupuestar()`: función que, dados unas horas de mano de obra y unos costes de piezas, calcula el coste de la reparación para la empresa.
- `Mostrar()`: función que muestra por pantalla los datos miembro de su clase.
- `Clonar()`: función que duplica el objeto para el cual es invocada.

```
class CSiniestro
{
  private:
    int    m_Codigo;
    static int m_SigCodigo;
  protected:
    string m_Descripcion;
    float  m_HorasMO;
    float  m_CostePiezas;
    float  m_Coste;
  public:
    CSiniestro(const string& Desc = "Sin Descripción");
    virtual ~CSiniestro() {};
    virtual float GetPresupuesto() const = 0;
    int GetCodigo() const { return m_Codigo; };
    float GetCoste() const { return m_Coste; };
    virtual void Presupuestar(float Horas, float Piezas) = 0;
    virtual void Mostrar(ostream& os = cout) const;
    virtual CSiniestro *Clonar() const = 0;
};
```

La clase `CSiniestroUrgente` representa los siniestros que requieren una reparación en un plazo de un día. Los siniestros de este tipo tienen unos costes adicionales asociados a la urgencia: un recargo, mano de obra más cara y un “plus” por transporte, según el tipo de servicio sea local, nacional o internacional. Por ello, al contrario que los demás tipos de siniestro, tiene un coste (presupuesto) para el cliente. Esta clase tiene la siguiente estructura:

- `m_Situacion`: tipo de cobertura del siniestro, según el tipo `TSituacion` descrito a continuación.
- `m_Recargo`: recargo por urgencia, fijo para todos los siniestros urgentes.

- `m_CosteHoraMO`: coste de la hora de mano de obra para este tipo de siniestro.
- `Presupuestar()`: ver clase `CSiniestro`.
- `Mostrar()`: ver clase `CSiniestro`.
- `Clonar()`: ver clase `CSiniestro`.
- `GetPresupuesto()`: ver clase `CSiniestro`.

```
enum TSituacion
{
  local, nacional, internacional
};

class CSiniestroUrgente : public  CSiniestro
{
  private:
    TSituacion   m_Situacion;
    static float m_Recargo;
    static float m_CosteHoraMO;
  public:
    CSiniestroUrgente(TSituacion s,
                      const string& Desc= "Sin Descripción");
    void Presupuestar(float Horas=0.5f, float Piezas=0);
    void Mostrar(ostream& os = cout) const;
    CSiniestroUrgente *Clonar() const;
    float GetPresupuesto() const;
};
```

La clase `CSiniestroNormal` representa un siniestro normal, de los que cubre el seguro, y que por lo tanto no tiene costes asociados para el cliente (presupuesto), pero si para la empresa. Tiene la siguiente estructura:

- `m_CosteHoraMO`: coste de la hora de mano de obra para este tipo de siniestro.
- `Presupuestar()`: ver la declaración de la clase `CSiniestro`.
- `Mostrar()`: ver la declaración de la clase `CSiniestro`.
- `Clonar()`: ver la declaración de la clase `CSiniestro`.
- `GetPresupuesto()`: ver la declaración de la clase `CSiniestro`.

```
class CSiniestroNormal : public CSiniestro
{
  private:
    static float m_CosteHoraMO;
  public:
    CSiniestroNormal(const string& Desc = "Sin Descripción");
    void Presupuestar(float Horas=0.5f, float Piezas=0);
    void Mostrar(ostream& os = cout) const;
    CSiniestroNormal *Clonar() const;
    float GetPresupuesto() const;
};
```

Para facilitar la realización de las clases así como su posterior integración en la aplicación, se ha creado también una *clase contenedor* auxiliar llamada `CLista`, que encapsula una lista enlazada. Esta lista está formada por elementos de tipo `CNodoLista`, y ambas clases se plantean como plantillas (*templates*) con el fin de que puedan contener objetos de cualquier tipo.

Un objeto `CLista` se crea: vacío, con un objeto inicial, o en base a otra lista. Una vez creada tiene sólo unos pocos métodos para manejarlo como contenedor de objetos. Se añaden objetos siempre al final, y a la hora de recuperarlos internamente tiene un indicador de cuál es el siguiente elemento a devolver, que avanza en la lista después de cada lectura. Los métodos son los siguientes:

- `AgregarObjeto()`: añade un objeto a la lista.
- `GetPrimero()`: devuelve el objeto de tipo `T` del elemento de la lista referenciado por `Primero` y avanza al siguiente.
- `GetActual()`: devuelve el objeto de tipo `T` del elemento de la lista referenciado por `Actual` y avanza al siguiente.
- `EstaVacia()`: indica si la lista contiene objetos. Esta función devuelve un valor de tipo `bool`. una variable de tipo `bool` puede almacenar un valor `true` o `false`. La constante `true` se corresponde con un valor distinto de `0` y `false` con `0`.
- `TieneMas()`: indica si hemos llegado o no al final de la lista. Esta función devuelve un valor de tipo `bool`.
- `Vaciar()`: borra los elementos `CNodoLista` de la lista y la deja preparada para aceptar nuevos elementos, pero no borra los objetos referenciados, si los hubiere, por los elementos `CNodoLista`, como es lógico.

```
template <class T>
class CLista
{
  private:
    CNodoLista<T> *m_Primero;
    CNodoLista<T> *m_Actual;
    CNodoLista<T> *m_Ultimo;
  public:
    CLista();
    CLista(const T& Obj);
    CLista(const CLista<T>& Lista);
    ~CLista();
    CLista<T>& operator=(const CLista<T>& Lista);
    bool EstaVacia() const { return m_Primero == NULL; }
    bool TieneMas() const  { return m_Actual != NULL); }
    void AgregarObjeto(const T& Obj);
    void Vaciar();
    T& GetPrimero() const;
    T& GetActual() const;
};
```

Un objeto `CNodoLista` permite encapsular cualquier dato o estructura de datos de tipo `T` (tipo genérico). Esta clase proporciona los métodos siguientes:

- `Get/SetSigNodo()`: Obtiene/establece el atributo `pSigNodo` de un objeto `CNodoLista`.
- `GetDato()`: Retorna el atributo `m_Dato` de tipo `T`.

```
template <class T>
class CNodoLista
{
    friend CLista<T>;
```

```
  private:
    T m_Dato;
    CNodoLista<T> *m_pSigNodo;
  public:
    CNodoLista(){ m_pSigNodo = NULL };
    CNodoLista(const CNodoLista<T>& NodoLista);
    CNodoLista(const T& Obj, CNodoLista<T> *pNodo = NULL);
    CNodoLista<T>& operator=(const CNodoLista<T>& NodoLista);
    CNodoLista<T> *GetSigNodo() const { return m_pSigNodo; };
    T& GetDato() { return m_Dato; };
    void SetSigNodo(CNodoLista<T> *pNodo) { m_pSigNodo = pNodo;};
};
```

A continuación se detalla parte del código de la clase `CLista`:

```
template <class T>
CLista<T>& CLista<T>::operator=(const CLista<T>& Lista)
{
  Vaciar();
  if(!Lista.EstaVacia())
  {
    m_Actual = m_Ultimo = m_Primero =
       new CNodoLista<T>(Lista.GetPrimero());
    while(Lista.TieneMas())
    {
      m_Actual->SetSigNodo(new CNodoLista<T>(Lista.GetActual()));
      m_Ultimo = m_Actual = m_Actual->GetSigNodo();
    }
  }
  return *this;
}

template <class T>
void CLista<T>::AgregarObjeto(const T& Obj)
{
  if(EstaVacia())
    m_Primero = m_Actual = m_Ultimo = new CNodoLista<T>(Obj);
  else
  {
    m_Ultimo->SetSigNodo(new CNodoLista<T>(Obj));
    m_Ultimo = m_Ultimo->GetSigNodo();
  }
}

template <class T>
T& CLista<T>::GetActual() const
{
  CNodoLista<T> *temp = m_Actual;
  const_cast<CLista<T>*>(this)->m_Actual =
     m_Actual->GetSigNodo();
  return temp->GetDato();
}

template <class T>
T& CLista<T>::GetPrimero() const
{
  const_cast<CLista<T>*>(this)-> m_Actual =
     m_Primero->GetSigNodo();
```

```
    return m_Primero->GetDato();
  }
```

Complete la funcionalidad de las clases descritas anteriormente escribiendo el código que dé respuesta a las preguntas que se realizan a continuación. Lógicamente esto requerirá implementar otro código necesario para el correcto funcionamiento de la aplicación. Como ayuda, descargue el fichero `ayuda-practica8.zip` y extraiga los ficheros de código contenidos en él. De los ficheros extraídos, lea el fichero `_leame.txt` donde encontrará instrucciones que le facilitarán la solución.

La aplicación deberá mostrar un menú en el que cada opción, 1, 2, 3, etc., se corresponda con la pregunta de igual número. La opción probará el buen funcionamiento del código que se pide implementar y/o mostrará la respuesta (en texto) a la pregunta o preguntas realizadas.

Verifique que no hay lagunas de memoria.

1.- Implemente la función `Vaciar` de la clase `CLista`. A la vista del código de las clases `CLista` y `CNodoLista` que se da, ¿es necesario que la clase `CLista` sea `friend` de `CNodoLista`? Razone su respuesta.

2.- Se desea implementar el operador de indexación (`operator[]`) para la clase `CLista`. Para ello, es necesario que el operador lance una excepción de tipo `CIndiceIncorrecto` en caso de que el índice sea mayor que el número de elementos de la lista o menor que cero (el primer índice es el `0`).

a) Implemente la clase `CIndiceIncorrecto`.
b) Implemente el operador de indexación de la clase `CLista`, incluyendo en el código el lanzamiento de la excepción.
c) Ponga, en la función `main`, un ejemplo simple de cómo capturaría y trataría la posible excepción. Utilice por ejemplo una lista de `int`.

3.- Implemente el operador de asignación de la clase `CContrato` y después el constructor copia en función de éste. Pruebe el correcto funcionamiento de este operador mediante este código (`original` debe tener, al menos, un siniestro):

```
CContrato copia(original);
original = copia;
```

Nota: utilice directa o indirectamente la función `Clonar` correspondiente.

4.- Implemente la función `AgregarSiniestro` de la clase `CContrato` utilizando la función `CSiniestro::Clonar`. ¿Es necesario que `Clonar` sea virtual? ¿Por qué?

5.- Implemente el operador de inserción correspondiente para que dado un objeto `cli` de tipo `CCliente` sea posible escribir:

```
cout << "Datos del cliente: \n" << cli << endl;
```

Escriba de forma explícita la sentencia anterior.

Nota: Considere implementada la siguiente función `friend` de la clase `CContrato`:

```
friend ostream& operator<<(ostream& os, const CContrato& c);
```

6.- Inicie la variable `m_SigCodigo` de la clase `CSiniestro` para que el primer objeto derivado de `CSiniestro` que se cree tenga como código el `1`. ¿Dónde se debe realizar la iniciación? ¿Es imprescindible? Razone la respuesta.

7.- En la clase `CSiniestro` y en sus derivadas la función `Presupuestar` necesita acceder, en principio para obtener o establecer los valores, a los datos miembro `m_Coste`, `m_HorasMO` y `m_CostePiezas`. Según esto, ¿podría ser `private` la variable `m_Coste` en lugar de `protected`? Y pensando en la función miembro `GetCoste` ¿podría ser `private` la variable `m_Coste` en lugar de `protected`? Razone las respuestas.

8.- Implemente la función `AgregarContrato` de la clase `CCliente`.

9.- Añada las funciones en las clases correspondientes para que se compile y se ejecute correctamente el siguiente código:

```
const int MAX_CLIENTES 10;

int main()
{
  CCliente *seguros;
  //... Se crea una matriz dinámica de tipo CCliente apuntada por
  //... seguros. Se realizan operaciones con seguros.

  long total = 0;
  for(int i = 0; i < MAX_CLIENTES; i++)
    total += seguros[i];

  cout << "\nEl número total de contratos de los clientes asciende a: ";
  cout << total << " contratos\n ";
}
```

No se permite sobrecargar el operador +=.

10.- Implemente las funciones que se llaman cuando se ejecuta la sentencia siguiente:

```
CSiniestroNormal s("Rotura de tambor");
```

11.- ¿Se puede eliminar el código sombreado? Razone la respuesta.

```
template <class T>
T& CLista<T>::GetPrimero() const
{
  const_cast<CLista<T>*>(this)->m_Actual = m_Primero->GetSigNodo();
  return m_Primero->GetDato();
}
```

12.- En la clase `CSiniestro`, ¿podríamos llamar a la función `GetCoste` desde la función `Mostrar` tal como se indica a continuación? Razone la respuesta. En caso negativo proponga soluciones para que sí se pueda.

```
void CSiniestro::Mostrar(ostream& os) const
{
  // ...
  GetCoste();
  // ...
}
```

13.- Dada la siguiente secuencia de sentencias, especifique las funciones que son llamadas en la línea 3 y el orden de llamada de las mismas:

```
1. CContrato c(12345, "Cafetera C1Z", "2/1/2002", 100, 1000);
2. CSiniestroUrgente s(nacional, "Fallo general");
3. c.AgregarSiniestro(s);
```

14.- Se ha diseñado una clase `CEmpresa` para manejar los clientes de la empresa. La declaración de la clase es la siguiente:

```
class CEmpresa
{
  private:
    CCliente *m_pElem;
    int m_nElem;
  public:
    CEmpresa() : m_nElem(0), m_pElem(NULL) {};
    CEmpresa(const CEmpresa& a);
    CEmpresa& operator=(const CEmpresa& a);
    ~CEmpresa();
    void AgregarElemento(const CCliente& elem);
    int Tamanyo() const { return m_nElem; }
    CCliente& GetElemento(int nElem) const;
    CCliente& operator[](int nElem) const;
};
```

Rescriba la declaración anterior de la clase `CEmpresa` para que permita manejar no solo elementos de tipo `CCliente` sino de cualquier otro tipo.