PRÁCTICA 8ª: Plantillas, excepciones y flujos.

OBJETIVOS: Repaso de todo lo anterior la clase `CSiniestro` con sus derivadas es equivalente a `CFicha` con sus derivadas, práctica 7). Además, se incluye el uso de plantillas, flujos y punteros inteligentes.

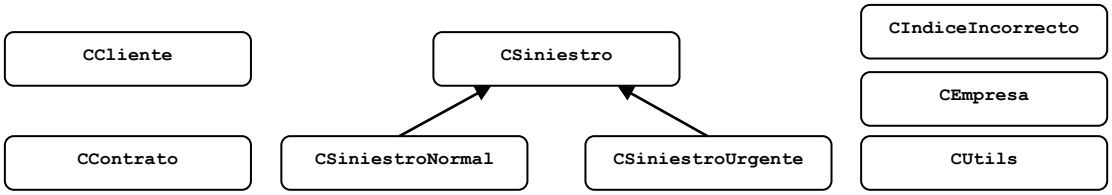
TEMPORIZACIÓN:
Publicación del enunciado: Semana del 20 de diciembre.
Entrega: Semana del 10 de enero.
Límite de entrega: Semana del 26 de enero.

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 seguro 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 biblioteca 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 (o método) 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;
    vector<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.
- `MostrarSiniestros()`: escribe los siniestros del contrato.

```
class CContrato
{
  private:
    int    m_NumSerie;
    string m_Descripcion;
    string m_FechaFin;
    long   m_Poliza;
    long   m_ValorCompra;
    vector<ptr_unique<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);
    void MostrarSiniestros(std::ostream& os = std::cout);
};
```

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 miembro que, dados unas horas de mano de obra y unos costes de piezas, calcula el coste de la reparación para la empresa.
- `Mostrar()`: función miembro que muestra por pantalla los datos miembro de su clase.
- `Clonar()`: función miembro 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;
    };
```

1.- 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 (a través de `AgregarSiniestro`) la función miembro `Clonar` correspondiente.

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

3.- 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: Tenga en cuenta la siguiente función `friend` de la clase `CContrato`:

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

4.- 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.

5.- En la clase `CSiniestro` y en sus derivadas la función miembro `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.

6.- Implemente la función miembro `AgregarContrato` de la clase `CCliente`.

7.- 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()
{
// Crear matriz dinámica
vector<unique_ptr<CCliente>> seguros(0);

// Rellenar matriz dinámica (El cliente i tiene i+1 contratos)
```

```
for (int i = 0; i < MAX_CLIENTES; i++)
{
      seguros.push_back(unique_ptr<CCliente>(new CCliente));

      for (int j = 0; j < (i + 1); j++)
            seguros[i]->AgregarContrato(CContrato(j + 1));
          }

      // Contar número total de contratos
      long total = 0;

      for(int i=0; i < MAX_CLIENTES; i++)
            total += *seguros[i]; // No implementar operator+=

      cout << "\n El número total de contratos de los "
           << "clientes asciende a: ";
      cout << total << " contratos\n";

}
```

No se permite sobrecargar el operador +=.

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

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

9.- En la clase `CSiniestro`, ¿podríamos llamar a la función miembro `GetCoste` desde la función miembro `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();
  // ...
}
```

10.- 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);
```

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

```
class CEmpresa
{
  private:
    vector<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.

A continuación, construya una empresa `CEmpresa<CCliente> empresa1`, agregue objetos `CCliente` a `empresa1`. Para probar el constructor copia y el operador de asignación, en su totalidad, de `CEmpresa<CCliente>` incluya las siguientes sentencias:

```
CEmpresa<CCliente> empresa2(empresa1); // constructor copia
empresa2 = empresa1;                   // operador de asignación
```

y muestre el contenido de `empresa2` en la pantalla utilizando el operador de indexación.

12.- Cree una empresa igual que en el punto anterior. Después escriba el código necesario para grabar en un "fichero de texto" (`clientes.txt`) el nombre de los clientes de la empresa. Los datos tendrán que ser debidamente separados para poderlos recuperar posteriormente. Verificar que el fichero puede ser abierto. Finalmente, muestre el contenido del fichero.

NOTA: si al incluir el fichero de cabecera `fstream` obtiene errores de compilación, pruebe a incluirlo en la línea número 1.

Un puntero inteligente de C++ se asemeja a la creación de objetos en lenguajes como C# o Java: se crea el objeto y después es el sistema (en este caso, el destructor del puntero inteligente) el que se ocupa de eliminarlo en el momento correcto. La diferencia es que ningún recolector independiente de elementos no utilizados se ejecuta en segundo plano; la memoria se administra con las reglas estándar de ámbito de C++, método más rápido y eficaz.

```
#include <iostream>
#include <memory>   // para unique_ptr
using namespace std;

struct fecha
{
  int dd, mm, aaaa; // día, mes y año
};

void func1(fecha f)
{
  // Puntero inteligente: ptr a fecha
  unique_ptr<fecha> p(new fecha(f));

  // Otro código que puede lanzar alguna excepción
```

```
  // ...
  cout << (*p).mm << endl;
  cout << p->aaaa << endl;
  // p sale fuera de ámbito y su destructor libera el objeto fecha
}

int main()
{
  fecha cumple{ 1, 2, 2050 };
  func1(cumple);
}
```

Fíjese en el uso de los punteros inteligentes en el caso de la clase CContrato:

```
class CContrato
{
friend ostream& operator<<(ostream& os,  CContrato& c);
private:
  int m_NumSerie;
  std::string m_Descripcion;
  std::string m_FechaFin;
  long m_Poliza;
  long m_ValorCompra;
  vector<std::unique_ptr<CSiniestro>> m_Siniestros;
...
```

En lugar de añadir a `m_Siniestros` punteros a `CSiniestro` se añaden objetos de tipo `unique_ptr<CSiniestro>` que encapsulan esos punteros a `CSiniestro`. De esta forma, ya no se necesita el destructor de `CContrato`.