ANEXO DE AYUDA:

1) Hacer que `Contrato` muestre un comportamiento polimórfico dinámico: polimorfismo.

El comportamiento polimórfico se refiere a la capacidad que determinar en tiempo de ejecución el método que se tiene que ejecutar en función del objeto que invoca el método cuando dicho objeto es apuntado (o referenciado) por un puntero (o referencia) de una clase superior de la jerarquía de clases a la que pertenece.

Un método declarado en una clase puede ser redefinido en una clase hija para hacer una cosa distinta a la que hace en la clase base. Cuando se invoca cualquier método el compilador determina en tiempo de compilación (es decir, antes de ejecutarse el programa) la versión (la original o la redefinida) del método que se ejecutará seleccionando por defecto la del tipo del puntero (o referencia) que apunta (o referencia) al objeto. El problema estriba en que un puntero o referencia de una clase también puede apuntar a objetos de una clase hija, por lo que interesa que la versión del método que se ejecute se elija mirando el tipo del objeto al que apunta el puntero (no dando por hecho que será del mismo tipo que el puntero). Esto sólo se puede determinar en tiempo de ejecución (no en tiempo de compilación) y para indicarle al compilador que lo determine de forma dinámica mirando la clase a la que apunta el puntero (y no la de la clase del puntero) hay que poner delante del método en cuestión la palabra clave `virtual`.

Por tanto, para hacer que una clase tenga comportamiento polimórfico basta con poner en la clase superior (es decir, en la clase de la que heredan las otras) la palabra clave `virtual` delante de todos los métodos que están redefinidos en las clases hijas.

Además es muy importante también declarar el destructor de la clase superior virtual. La razón es la siguiente: si creamos de forma dinámica (con `new`) un objeto de una clase hija y usamos un puntero de la clase superior para apuntarlo, cuando invoquemos el método `delete` usando dicho puntero el compilador ejecutará por defecto el destructor de la clase del puntero, por lo que ejecutará el destructor de la clase superior (cuando lo correcto sería ejecutar el destructor de la clase hija, que es la verdadera clase del objeto creado). Si el destructor de la clase superior es declarado `virtual` y ocurre lo comentado anteriormente el destructor que se ejecutará será el de la clase hija (el correcto) ya que con `virtual` le estamos indicando al compilar que determine en tiempo de ejecución lo que debe ejecutar atendiendo al tipo al que apunta el puntero (en lugar de mirar el tipo del puntero).

En el caso concreto de la práctica 1, sólo el método `ver()` está definido en la clase `Contrato` y redefinido en las clases hijas `ContratoTP` y `ContratoMovil`. Por tanto debemos declarar `virtual` dicho método y el destructor de `Contrato`

```
class Contrato {
  ...
  virtual ~Contrato();
  virtual void ver() const;
}
```

2) Hacer que `Contrato` sea una clase abstracta

Para que la clase `Contrato` sea abstracta debe tener un método que no esté implementado. De esta forma al faltarle la implementación de un método concreto, el compilador no va a permitir crear objetos de dicho tipo al no saber qué es lo que tiene que hacer si se invoca dicho método. Para evitar que el compilador de un error porque piense que la falta de implementación del método *"es un olvido del programador"* le tenemos que indicar explícitamente que no tenemos intención de implementarlo declarando el método `virtual` y poniendo al final de la cabecera la coletilla "`= 0`".
Al método declarado de esa manera se le denomina *método virtual puro*.

Una clase hija de una clase abstracta hereda todos sus métodos, incluido el *método virtual puro*. Si dicha clase hija tampoco lo implementa se convertirá también en una clase abstracta y no se podrán crear objetos de dicha clase hija. Por tanto, para evitar que una clase hija de una clase abstracta también sea abstracta, simplemente se tiene que implementar en la clase hija el método virtual puro heredado de la clase abstracta.

Por tanto, para hacer que la clase `Contrato` sea abstracta y sus hijas no lo sean, simplemente hay que crear un método virtual puro en dicha clase e implementarlo en sus clases hijas `ContratoTP` y `ContratoMovil`. Si no encontrarnos ningún método candidato que podamos dejar de implementar en la clase superior, nos inventamos uno que no sirva para nada en la clase superior y lo implementamos en sus clases hijas.

```
class Contrato {
  ...
  virtual void nada() const = 0; //indico que este método no lo vamos a implementar
}

class ContratoTP: public Contrato {
  ...
  virtual void nada() const { ; } //lo implemento (si no quiero que haga nada pongo ;
};

class ContratoMovil: public Contrato {
  ...
  virtual void nada() const { ; } //lo implemento (si no quiero que haga nada pongo ;
};
```

En el caso concreto de la práctica, como ambas clases hijas tienen un método `factura()` podemos declarar dicho método en la clase base `Contrato` y no implementarlo (no tiene sentido y no es posible hacerlo en la clase base al carecer de la información necesaria para ello), haciendo que dicha clase se convierta en abstracta.

3) Programar la clase *Empresa*

La clase `Empresa` necesita utilizar 2 arrays, uno para almacenar la lista de `Clientes` y otro para almacenar los `ContratoTP` y `ContratoMovil` (para poder meter en la misma tabla objetos de tipo `ContratoTP` y tipo `ContratoMovil` utilizaremos una tabla de tipo `Contrato *` por ser ésta la clase de la que derivan los 2 tipos de contratos existentes).

Por otro lado, para poder crear objetos `Cliente`, `ContratoTP` o `ContratoMovil` necesitamos pasarle cierta información al constructor (nombre, fecha, etc.). Debido a ello no va a ser posible crear arrays de tipo `Cliente` o de tipo `Contrato` (al no existir un constructor sin parámetros para ellos). Por tanto, sólo podremos crear arrays de punteros a objetos de tipos `Cliente` o `Contrato`.

```
Cliente cli[6];  //no permitido al no existir un constructor sin parámetros Cliente()
Contrato con[6];   //idem

Cliente*  cli[6]; //si permitido ya que no estamos creando ningún cliente sino punteros
                  //hemos creado 6 punteros a Cliente, no hemos creado ningún Cliente
Contrato* con[6];   //idem

cli[0]=new Cliente(75547001, "Juan Luis", Fecha(29,2,2000));
cli[1]=new Cliente(75547555, "Pepe", Fecha(01,3,2008));
//creo (con new) los Clientes y hago que los punteros cli[0] y cli[1] apunten a ellos
//lo mismo podemos hacer con Contrato...
```

Los arrays que hemos creado son estáticos (de tamaño fijo 6) y cuando se llenen no caben más elementos. Nos interesa que los arrays sean dinámicos de forma que su tamaño puede ir creciendo cuando se vayan llenando.

Un array dinámico se crea usando new y usando un puntero que lo apunte. Cuando el array esté lleno y se necesite ampliar su capacidad hay que reservar una nueva zona de memoria mayor (con new), copiar la información de la zona de memoria antigua a la nueva, liberar la antigua y hacer que el puntero apunte a la zona de memoria nueva.

A continuación explicamos cómo realizar esto con un array de enteros:

Ejemplo de creación y ampliación de arrays dinamicos

```
Los arrays estáticos NO pueden cambiar su tamaño (una vez creados no pueden crecer)

int t[5];          //array estatico de enteros de 5 elementos     [?, ?, ?, ?, ?]
                   // ? indica que los valores son aleatorios

Los array dinámicos SI pueden cambiar su tamaño (una vez creados si “pueden crecer”)

int *t;            //puntero a entero (aun no se ha creado el array)
t=new int[5];      //array dinamico de enteros de 5 elementos     [?, ?, ?, ?, ?]
for(i=0; i<5; i++)
  t[i]=i+1;        //                                             [1, 2, 3, 4, 5]

Supongamos que queremos que t tenga 8 elementos en vez de 5 ¿que hacer?

int *aux;         //creamos un puntero auxiliar de tipo int
aux=t;            //hacemos que dicho puntero auxiliar apunte al array dinamico
t=new int[8];     //creamos un nuevo array dinamico mayor y haceos que t apunte a él
                                                      [?, ?, ?, ?, ?, ?, ?, ?]
for(i=0; i<5; i++)
  t[i]=aux[i];    //copiamos del array antiguo al nuevo  [1, 2, 3, 4, 5, ?, ?, ?]
delete [] aux;    //liberamos la memoria del array dinamico antiguo

Asi hemos conseguido que en t quepan ahora 8 elementos en vez de 5
```

Si en lugar de array de enteros tuviéramos array de punteros a enteros, el proceso a seguir sería el mismo explicado antes:

```
Idem con array de punteros a enteros:

int *t[5];        //array estatico de 5 punteros a enteros
                  // *? indica que son punteros aleatorios     [*?, *?, *?, *?, *?]

int **t;          //puntero a puntero a entero (aun no se ha creado el array)
t=new int *[5];   //array dinamico de 5 punteros a enteros     [*?, *?, *?, *?, *?]
...
int **aux;        //creamos un puntero auxiliar de tipo int *
aux=t;            //hacemos que dicho puntero auxiliar apunte al array dinamico
t=new int*[8];    //creamos un nuevo array dinamico mayor y haceos que t apunte a él
                                                  [*?, *?, *?, *?, *?, *?, *?, *?]
for(i=0; i<5; i++)
  t[i]=aux[i];    //copiamos del array antiguo al nuevo [*1, *2, *3, *4, *5, *?, *?, *?]
delete [] aux;    //liberamos la memoria del array dinamico antiguo
```

Si en lugar de array de enteros o de punteros a enteros, tuviéramos arrays de `Contrato` o array de punteros a `Contrato` los pasos a realizar serían exactamente los mismos que hemos indicado arriba (solo habría que cambiar el tipo `int` por el tipo `Contrato`).

Atendiendo a todo lo explicado anteriormente y lo que se nos pide en el enunciado, la definición de la clase Empresa quedaría de la siguiente manera:

Empresa.h

```
#ifndef EMPRESA_H
#define EMPRESA_H

#include "Fecha.h" //definicion clase Fecha
#include "Cliente.h" // definicion clase Cliente
#include "Contrato.h" // definicion de la clase Contrato
#include "ContratoTP.h" // definicion de la clase ContratoTP
#include "ContratoMovil.h" // definicion de la clase ContratoMovil

using namespace std;

class Empresa {
    Cliente *clientes[100]; //array estático (tamaño 100) de punteros a Clientes
    int ncli;           //para saber cuántos clientes hay en el array (al principio 0)
    const int nmaxcli;  //constante que indica cuántos caben en el array clientes (100)
    Contrato **contratos; //array dinámico de punteros a Contratos (capacidad ilimitada)
    int ncon;            //para saber cuántos Contratos hay en el array (al principio 0)
    int nmaxcon;         //para saber cuántos caben en array Contratos. No es constante
                         //porque varía conforme la tabla se llena (10, 20, 40, 80...)

  protected:             //métodos auxiliares usados por los métodos públicos
    int buscarCliente(long int dni) const; //si no existe devuelve -1 y si existe devuelve
                                           //la posición del cliente en el array clientes

    int altaCliente(Cliente *c); //añade el cliente apuntado por c al array clientes
                                 //devuelve la posición donde lo mete (-1 si no cabe)
  public:
    Empresa();
    virtual ~Empresa();
    //EL CONTRUCTOR DE COPIA Y EL OPERADOR DE ASIGNACION NO LO IMPLEMENTAMOS
    //PORQUE EXPLICITAMENTE SE INDICA EN LA PRACTICA QUE NO SE HAGA

    void crearContrato();
    bool cancelarContrato(int idContrato); //true si el Contrato existe, false si no
    bool bajaCliente(long int dni);        //true si el Cliente existe, false si no
    int descuento (float porcentaje) const; //devuelve a cuantos aplica el descuento
    int nContratosTP() const;
    void cargarDatos();//crea 3 clientes y 7 contratos. metodo creado para no
};                     //tener que meter datos cada vez que pruebo el programa

#endif // EMPRESA_H
```