



**Fundamentos de Programación**  
*Grado en Ing. Informática*

**Guion práctico nº 4**

***Tema 5.- Diseño Descendente***



DEPARTAMENTO DE  
TECNOLOGÍAS DE  
LA INFORMACIÓN

**Universidad de Huelva**

## Diseño Descendente

---

1. Rediseñar la solución al ejercicio 1 del guion 3 de manera que se utilicen las siguientes funciones:

```
void rellenar (int T[10]);  
  
/* Rellena la tabla recibida como parámetro, con valores leídos  
desde teclado. */  
  
int minimo (int T[10]);  
  
// Devuelve el menor valor de la tabla.  
  
int maximo (int T[10]);  
  
// Devuelve el mayor valor de la tabla.
```

2. Modificar la solución al ejercicio 2 del guion 3 haciendo uso de las funciones que se indican a continuación:

```
void rellenar (int T[10]);  
  
/* Rellena la tabla recibida como parámetro, con valores leídos  
desde teclado. */  
  
bool buscar (int T[10], int x);  
  
/* Devuelve true si el valor x pasado como parámetro se encuentra  
en la tabla y false en caso contrario. */
```

3. Actualice la solución al ejercicio 3 del guion 3, para utilizar las funciones que se detallan:

```
void rellenar (int T[10][15]);  
  
// Rellena la tabla con valores aleatorios del intervalo [0,100]  
  
bool buscar (int T[10][15], int x);  
  
/* Devuelve true si el valor x pasado como parámetro se encuentra  
en la tabla y false en caso contrario. */
```

4. Reformar la solución al ejercicio 4 del guion 3 de tal modo que se haga uso de las funciones que se indican:

```
void rellenar (cadena D[3][4]);  
  
// Rellena la tabla con palabras introducidas desde teclado.  
  
bool buscar (cadena D[3][4], cadena S, int &f, int &c);  
  
/* Si la palabra S se encuentra en la tabla D devuelve true y la  
posición (fila y columna en f y c respectivamente) en la que está.  
En caso contrario devuelve false.*/
```

5. Rediseñar la solución al ejercicio 5 del guion 3 de manera que se utilicen las siguientes funciones:

```
void rellenar (persona T[2][4]);  
  
// Rellena la tabla con datos introducidos desde teclado.  
  
bool buscar (persona T[2][4], long dni, int &f, int &c);  
  
/* Devuelve true si existe en la tabla una persona con el dni  
indicado a través del parámetro de entrada y false en caso  
contrario. Además si existe, los parámetros de E/S f y c devolverán  
la fila y columna respectivamente en la que se encuentra dicha  
persona.*/
```

6. ¿Cómo debe ser el prototipo de una función que reciba como parámetro de entrada una frase y la devuelva con los espacios en blanco sustituidos por asteriscos, de modo que pueda ser utilizada en el ejercicio 6 del guion 3?

7. Reformar la solución al ejercicio 7 del guion 3 de modo que se utilice la función:

```
void compactar (char f[250], char c);  
  
// Elimina el carácter indicado por c, de la frase contenida en f.
```

8. Modificar la solución al ejercicio 8 del guion 3 para hacer uso de la función:

```
void invertir (char f[250], int ini, int fin);  
  
/* Invierte los caracteres comprendidos entre las posiciones ini y  
fin de la frase f. Esta función tiene como precondition que los  
valores ini y fin sean correctos (ini<fin, ini>=1,  
fin<=tamaño_de_la_frase).*/
```

9. Rediseñar la solución al ejercicio 9 del guion 3 de manera que se utilice la siguiente función:

```
bool palindroma (char f[250]);  
  
/* Devuelve true si la frase f es palíndroma, y false en caso  
contrario. Esta función debe hacer uso obligatoriamente de las  
funciones compactar() (para eliminar espacios en blanco) e  
invertir() realizadas para los ejercicios 7 y 8 respectivamente. */
```

10. Modificar la solución al ejercicio 13 del guion 3 haciendo uso de las funciones que se indican a continuación:

```
int menu ();  
  
/* Muestra el menú por pantalla, le solicita al usuario un opción y  
la devuelve. Solo podrá devolver un número de opción válido, por lo  
tanto mientras el usuario introduzca uno incorrecto se le volverá a  
solicitar. */
```

```
void rellenar (int v[10]);  
// Rellena el vector con valores leídos desde teclado.  
  
void mostrar (int v[10]);  
/* Muestra por pantalla el vector v, en una línea y con sus valores  
separados por comas. */  
  
bool iguales (int v1[10], int v2[10]);  
// Devuelve true si v1 y v2 son iguales y false en caso contrario.
```

11. Rediseñar la solución al ejercicio 16 del guion 3 de manera que se utilicen las siguientes funciones:

```
char generaVocal ();  
/* Devuelve una vocal generada de modo aleatorio según se indica en  
el enunciado del ejercicio en cuestión. */  
  
void estadistica (char t[1000], int N, char v, int &f, int &p);  
/* Devuelve la frecuencia y el porcentaje de aparición, en los  
parámetros de E/S f y p respectivamente, de la vocal indicada por v  
en el vector t, cuya cantidad de casillas rellenas se especifica  
por el parámetro N. */
```

12. Con respecto al ejercicio 18 del guion 3:

- ¿Cómo debería ser el prototipo de la función que permita rellenar un vector con la cantidad de elementos solicitada al usuario desde el programa principal?. ¿Qué ventaja aporta esta función?
- ¿Cómo debe ser el prototipo de la función que permita mostrar uno de los vectores?
- ¿Cómo debería ser el prototipo de la función que permita realizar la opción 4?
- ¿Podrías utilizar la función creada en el apartado b para realizar la opción 5 del menú? Si no pudieras ¿cómo debería ser el prototipo de dicha función para que fuese válida en ambas opciones del menú?

13. Describe cómo debería ser el prototipo de las funciones que utilizarías para resolver cada una de las opciones de menú planteadas en el ejercicio 20 del guion 3. ¿Cómo quedaría el programa principal haciendo uso de dichas funciones?

## Programación Orientada a Objetos (ejercicios cortos)

---

### Ejercicio 1. Vector

---

Codificar la siguiente clase:

```
class Vector
{   float valores[15];
    int nElem; //Nº de elementos del atributo valores
public:
    Vector();
    bool rellenar (int cuantos);
    void mostrar();
    int getnElem();
    float getValor(int pos);
    int menorRelativo(int P);
    void intercambio (int pos1, int pos2);
};
```

Teniendo en cuenta que la funcionalidad de los métodos es la que se describe a continuación:

- **Vector();** *//Pone a cero el atributo nElem*
- **bool rellenar (int cuantos);**  
*/\* Rellena el atributo valores con tantos datos solicitados al usuario como indique el parámetro cuantos. Si cuantos no se encuentra en el rango [1,15], no se podrá rellenar y devolverá false, en caso contrario devolverá true. \*/*
- **void mostrar ();** *// Muestra el vector por pantalla*
- **int getnElem();** *//Devuelve el valor de nElem*
- **float getValor(int pos);** *//Devuelve valores[pos]*
- **int menorRelativo (int P);**  
*/\* Devuelve la posición en la que se encuentra el elemento más pequeño del vector valores, comenzando su búsqueda desde la posición que indica el parámetro P. El valor P debe ser correcto, es decir, desde el programa principal se deberá comprobar dicha circunstancia y no llamar a esta función con un valor de P erróneo. \*/*
- **void intercambio (int pos1, int pos2);**  
*/\* Intercambia el contenido de las posiciones pos1 y pos2 del vector valores. Los parámetros pos1 y pos2 deben contener posiciones correctas dentro del vector, es decir, no se podrá llamar a esta función con valores erróneos en estos parámetros. \*/*

Implementar una función `main()` que compruebe la correcta codificación de la clase `vector`. Para ello se deberá mostrar el siguiente menú (mediante una función):

```
--- Menú ---
1. Insertar datos en vector
2. Mostrar vector
3. Menor relativo
4. Ordenar vector de menor a mayor
5. Salir
Elige Opción:
```

**Opción 1.** Rellena el vector con tantos elementos como indique el usuario. En caso de error en el proceso se mostrará un mensaje.

**Opción 2.** Muestra el vector por pantalla. Si el vector estuviese vacío se deberá indicar al usuario.

**Opción 3.** Pedirá una posición **P** (que deberá ser correcta) del vector a partir del cual muestre el elemento más pequeño y su posición. Ejemplo:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2.1	0.2	5.3	2.4	14.5	-10.5	3.7	0.8	4.9	81	92.3	2.6	0.15	5.1	0.4

Si P es 2 el valor más pequeño desde la posición 2 a la 14 es -10.5 y su posición es la 5

Si P es 6 el valor más pequeño desde la posición 6 a la 14 es 0.15 y su posición es la 12.

**Opción 4.** Ordena el vector de menor a mayor haciendo uso del método `intercambio()`.

**Opción 5.** Única forma de salir del programa.

## Ejercicio 2. Palabra oculta

---

Implementar la clase **PalabraOculta**, que nos permita jugar a adivinar una palabra. Dicha clase tendrá las siguientes características:

### Atributos privados:

- **char palabraSecreta[25];** // Palabra a adivinar.
- **char palabraJugada[25];**  
*/\* Inicialmente contendrá tantos guiones como letras tenga el atributo **palabraSecreta**. Los guiones irán siendo sustituidos por las letras que el jugador vaya adivinando en cada jugada. \*/*
- **int Puntos;** // Puntos conseguidos por el jugador.

### Métodos públicos:

- **void Iniciar();**  
*/\* Se establecerán los parámetros de inicio del juego:  
a) Se le pedirá al usuario la palabra a adivinar, teniendo en cuenta que dicha palabra no se podrá ver por pantalla y que en su lugar se mostrará una cadena de asteriscos. Además la cadena leída deberá almacenarse siempre en mayúsculas. Por todo lo anterior, al alumno le serán de utilidad las siguientes funciones:  
▪ **getch()** (librería **conio.h**), que lee un carácter desde teclado sin producir eco en pantalla.  
▪ **strupr()** (librería **cstring**), que convierte una cadena de texto a mayúsculas.  
b) El atributo **palabraJugada** contendrá tantos guiones como letras tenga **palabraSecreta**.  
c) Se establecerá a **9** la cantidad de puntos de partida del usuario.\*/*
- **bool Jugada(char letra);**  
*/\* Si la letra se encuentra en la palabra oculta se sustituirán todos los guiones por dicha letra en **palabraJugada**, en caso contrario se le quitará **1** punto al jugador. Si tras la jugada la **palabraSecreta** se ha adivinado, se devuelve **true** (**false** en caso contrario). La letra pasada como parámetro deberá ser transformada a mayúsculas, utilizando la función **toupper()** (librería **cctype**).\*/*
- **int getPuntos();** // Devuelve el valor del atributo **Puntos**.
- **void MostrarJugada();** // Muestra por pantalla **palabraJugada**.
- **void DescubrirSecreta();** // Muestra por pantalla **palabraSecreta**.

Codificar además un programa que nos permita jugar haciendo uso de esta clase, para ello:

- 1) Se establecerán los parámetros de inicio del juego, de manera que se le solicitará al usuario la palabra secreta (se deberán visualizar '\*' mientras se escribe) y se establecerán los puntos de partida a 9.
- 2) Comenzará la partida mostrándole al jugador tantos guiones como letras tenga la palabra secreta.
- 3) Se le solicitará una letra al jugador:
  - Si la letra está en la palabra secreta, se mostrará en sus correspondiente/s posición/es junto con el resto de guiones y letras ya descubiertas.
  - Si la letra no está, se le quitará un punto al jugador.
- 4) Si el jugador se queda con 0 puntos o bien adivina la palabra, el juego finaliza y se muestran los puntos conseguidos. En caso contrario se vuelve al paso 3).



## Programación Orientada a Objetos (ejercicios largos)

---

### Ejercicio 1. Productos-Almacén

---

Se va a realizar un programa en C++ para controlar los productos guardados en un almacén. De cada producto tendremos su **nombre**, **precio** y **stock** (cantidad en almacén). En primer lugar se implementará la clase **tprod**, para tratar un producto, y después codificaremos la clase **almacen**, que maneje una tabla de productos. Comencemos por la clase **tprod** para un producto:

```
typedef char cad[20];

class tprod
{
    cad nombre;    // Nombre del producto
    float precio;  // Precio del producto
    int stock;     // Cantidad de producto en el almacén
public: ...
};
```

Diseña los siguientes **métodos públicos** para esta clase:

- Un **constructor** para esta clase  
*// Pone "NO HAY PRODUCTO" en nombre, y 0 en precio y en stock.*
- **void cambiarnombre(cad nom);**  
*// Recibe nom como parámetro y lo copia en el atributo nombre.*
- **void cambiarprecio(float prec);**  
*// Recibe prec como parámetro y lo copia en el atributo precio.*
- **void cambiarstock(int stoc);**  
*// Recibe stoc como parámetro y lo copia en el atributo stock.*
- **void leenombre (cad nom);**  
*// Devuelve en el parámetro nom el valor del atributo nombre.*
- **float leeprecio ();**  
*// Devuelve el contenido del atributo precio.*
- **int leestock();**  
*// Devuelve el contenido del atributo stock.*
- **bool vender(int cantidad, float &total);**  
*/\* Simula la venta del producto, quitando del stock la cantidad pasada como parámetro, además devolverá en total el precio a cobrar (precio unitario \* cantidad vendida). Si la venta se ha podido realizar (hay suficiente cantidad en el stock) el método devolverá true, en caso contrario devolverá false. \*/*

Diseñe un `main()` que compruebe el funcionamiento correcto de todos estos métodos. Cree un objeto de tipo `tprod`, visualice su contenido, cambie el `nombre`, `precio` y `stock` con valores leídos desde teclado, muestre por pantalla el nuevo contenido, intente realizar una venta con más cantidad que la existente en el `stock`, y otra con menos, visualice el producto, etc.

Pasemos a tratar la **clase almacen**:

```
#define MAX 5
class almacen
{
    tprod productos[MAX];
    int nprod;    //Nº de objetos tipo tprod almacenados en productos

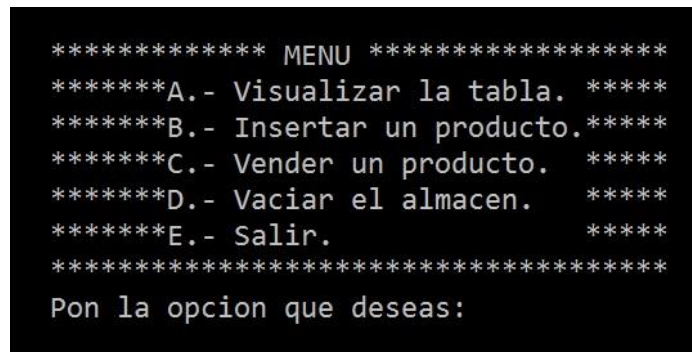
public:
};
```

Diseñe los siguientes **métodos públicos** para esta clase:

- `almacen();` */\* Constructor que pondrá el almacén vacío \*/*
- `void vaciar();` */\* Pondrá el almacén vacío \*/*
- `int existe(cad nom);`  
*/\* Recibe el nombre de un producto como parámetro y devuelve en qué posición de la tabla se encuentra almacenado o bien -1 si no está. \*/*
- `void verprod (int pos, tprod &prod);`  
*/\* Copia en prod el contenido del producto que se encuentra en la posición pos de la tabla de productos. \*/*
- `int insertar(tprod P);`  
*/\* Intentará insertar un nuevo producto P pasado como parámetro en la tabla de productos, y devolverá:*
  - **2**, si la tabla está llena.
  - **1**, si ya existe el producto (no se insertará de nuevo).
  - **0**, si lo ha podido insertar. *\*/*
- `void vertabla ();`  
*/\* Visualiza por pantalla el contenido del almacén. Cada producto deberá mostrarse en una línea diferente con su nombre, precio y stock. Si el almacén está vacío expresará esta situación por pantalla. \*/*
- `bool vender (int pos, int cant, float &total);`  
*/\* Intentará realizar la venta de una cantidad cant del producto que está en la posición pos de la tabla. Devolverá true o false en función de que se haya podido o no realizar la venta. \*/*

Diseñar además la siguiente **función menu ()** que muestre por pantalla las opciones que se indican en la imagen:

```
char menu ();  
  
/* Muestra el menú por pantalla y devuelve la opción seleccionada  
por el usuario. No podrá devolver una opción incorrecta. */
```



```
***** MENU *****  
*****A.- Visualizar la tabla. *****  
*****B.- Insertar un producto.*****  
*****C.- Vender un producto. *****  
*****D.- Vaciar el almacen. *****  
*****E.- Salir. *****  
*****  
Pon la opcion que deseas:
```

Diseñar un **main()** que compruebe el funcionamiento correcto de los métodos de la **clase almacen**, para ello:

- Borrar el **main()** que se realizó anteriormente.
- En el nuevo **main()** :
  - Crear al menos un objeto de tipo **almacen**.
  - Mostrar el menú anterior:
    - La única manera de que acabe el programa es pulsando la opción de salir (**E** o **e**).
    - Si se pulsa **A** (o **a**), mostrará por pantalla el contenido completo del almacén.
    - Si se pulsa **B** (o **b**), solicitará por teclado el **nombre** del producto nuevo, el **precio** y el **stock**, e intentará insertarlo. Si el almacén está lleno expresará dicha situación por pantalla y si el producto ya existía también.
    - Si se pulsa **C** (o **c**), pedirá por teclado el **nombre** del producto, si ese producto no existe lo indicará por pantalla, en caso de que exista le pedirá la **cantidad** a vender, e intentará realizar la venta, descontando la cantidad vendida del **stock** e indicando el **precio** de la venta total por pantalla o bien que no había suficiente cantidad en **stock**.
    - Si se pulsa **D** (o **d**), se pondrá el almacén vacío de productos.

## Ejercicio 2. Cuentas corrientes

Realizar un programa en C++ para controlar las cuentas corrientes de una persona. Para ello definimos la siguiente clase que es capaz de almacenar y gestionar una sola cuenta.

```
typedef char Cadena[50]; // Tipo de datos Cadena
#define MAX_CUENTAS 100 // Número de Cuentas

class Cuenta //Contiene los datos de una cuenta bancaria
{
    float Saldo; // Saldo de la cuenta
    int NoCuenta; // Número de la cuenta
    bool Bloqueada; // true si está bloqueada
public:
    Cuenta();
    Cuenta(int pNo, float pSal);
    bool ActualizarSaldo(int pSal);
    void ActualizarBloqueo(bool pBloq);
    float DameSaldo();
    int DameNoCuenta();
    bool EstaBloqueada();
};
```

Diseñar los siguientes métodos:

- Un constructor sin parámetros

```
/* Inicializa a 0 los atributos Saldo y NoCuenta y a false el
atributo Bloqueada. */
```

- Un constructor parametrizado

```
/* Inicializa los atributos Saldo y NoCuenta con el valor de los
parámetros pSal y pNo respectivamente, y el atributo Bloqueada a
false. */
```

- **bool ActualizarSaldo(int pSal);**

```
/* Actualiza el atributo Saldo con el valor del parámetro pSal
siempre y cuando la cuenta no esté bloqueada. Devuelve true si se ha
actualizado el saldo y false en caso contrario. */
```

- **void ActualizarBloqueo(bool pBloq);**

```
// Actualiza el atributo Bloqueada con el valor de pBloq.
```

- **float DameSaldo();**

```
// Devuelve el atributo Saldo.
```

- **int DameNoCuenta();**

```
// Devuelve el número cuenta (NoCuenta).
```

- **bool EstaBloqueada();**

```
// Devuelve true si la cuenta está bloqueada, false en caso contrario.
```

Diseñar las siguientes **funciones**:

- `int BuscarCuenta (Cuenta Ctas[MAX_CUENTAS], int NCuentas, int NoCuenta);`  
*/\* Recibe un vector de cuentas (Ctas), cuántas cuentas están utilizándose (NCuentas) y el nº de cuenta a buscar (NoCuenta). Devuelve la posición dentro del vector Ctas que contiene el nº de cuenta especificado por parámetro. Si no existe ninguna cuenta con ese nº devolverá -1. \*/*
- `int MenuCuentas();`  
*// Muestra el siguiente menú y devuelve la opción seleccionada.*

```
--- Menú Gestión de Cuentas ---  
1. Añadir una cuenta a un cliente  
2. Mostrar las cuentas del cliente  
3. Borrar una cuenta del cliente  
4. Modificar saldo de una cuenta  
5. Modificar estado de una cuenta  
6. Salir  
Elige Opción:
```

Diseñe un `main()` que compruebe el funcionamiento correcto de los métodos de esta clase. Para ello definirá las siguientes variables locales de contendrán las cuentas de un cliente de banca:

```
Cuenta DatosCuentas[MAX_CUENTAS];  
int nCuentas = 0; //Nº de elementos del vector DatosCuentas
```

La función `main()` mostrará el menú y realizará las siguientes acciones según la opción elegida:

#### Opción 1.

Si hay espacio en `DatosCuenta`, solicitará por teclado el nº de la nueva cuenta y buscará en dicho vector si existe ya una con dicho número. Si existe mostrará un mensaje de error, y en caso contrario solicitará el saldo de la cuenta y actualizará el vector `DatosCuenta`. En caso de no haber espacio suficiente mostrará un mensaje de error.

#### Opción 2.

Listará por pantalla los datos de cada cuenta del vector `DatosCuenta`.

**Opción 3.**

Solicitará por teclado el nº de cuenta a ser eliminada y la buscará en el vector **DatosCuenta**. Si no la encuentra mostrará un mensaje de error y si la encuentra la eliminará desplazando todos los objetos que hay en el vector **DatosCuenta** una posición hacia la izquierda a partir de la posición de la cuenta a eliminar.

**Opción 4.**

Solicitará por teclado el nº de cuenta a ser actualizada y buscará dicha cuenta en el vector **DatosCuenta**. Si no la encuentra mostrará un mensaje de error, y si la encuentra actualizará dicha cuenta con el saldo solicitado por teclado. Si no se actualiza el saldo de la cuenta, mostrará un mensaje de error por estar la cuenta bloqueada.

**Opción 5.**

Solicitará por teclado el número de cuenta a ser actualizada y buscará dicha cuenta en el vector **DatosCuenta**. Si no la encuentra mostrará un mensaje de error, y si la encuentra solicitará por teclado un carácter (**s** o **n**) para indicar si se desea bloquear la cuenta o no. A continuación actualizará el bloqueo de la cuenta según la opción elegida.

**Opción 6.**

Termina el programa.

### Ejercicio 3. Clientes de un banco

Realizar un programa en C++ para **gestionar los clientes de un banco**. Este problema reutiliza prácticamente todo el código del ejercicio anterior de **Cuentas**. Para almacenar un cliente y todas sus cuentas bancarias definimos la siguiente clase:

```
typedef char Cadena[50]; // Tipo de datos Cadena
#define MAX_CUENTAS 10 // Número de Cuentas
#define MAX_CLIENTES 100 // Número de clientes

class Cliente
{
    Cadena Nombre; // Nombre y dirección
    Cadena Direccion;
    Cuenta Cuentas[MAX_CUENTAS]; // cuentas corrientes
    int NoCuentas; // N° de cuentas abiertas
public:
    Cliente();
    void ActualizarCliente(Cadena pNomb, Cadena pDir);
    void DameNombre(Cadena pNom);
    void DameDireccion(Cadena pDir);
    int BuscarCuenta(int pNoCuenta);
    bool CrearCuenta(Cuenta pCu);
    bool ActalizarCuenta(Cuenta pCu);
    bool BorrarCuenta(int pNoCuenta);
    int DameNoCuentas();
    Cuenta DameCuenta(int pos);
    void Mostrar(char Campo);
};
```

Diseñe los siguientes **métodos** para esta clase:

- Un **constructor sin parámetros**.

*// Inicializa las cadenas a vacío y el n° de cuentas a cero.*

- **void ActualizarCliente(Cadena pNomb, Cadena pDir);**

*/\* Actualiza los atributos **nombre** y **dirección** con los parámetros **pNomb** y **pDir** respectivamente. El atributo número de cuentas (**NoCuentas**) se inicializa a 0. \*/*

- **void DameNombre(Cadena pNom);**

*// Devuelve el **nombre** del cliente mediante el parámetro **pNom**.*

- **void DameDireccion(Cadena pDir);**

*// Devuelve la **dirección** del cliente mediante el parámetro **pDir**.*

- **int BuscarCuenta(int pNoCuenta);**

*/\* Busca, en todas las **cuentas** del cliente, aquella cuyo n° de cuenta coincide con el valor del parámetro **pNoCuenta**. \*/*

▪ **bool CrearCuenta(Cuenta pCu) ;**

*/\* Crea una cuenta nueva al cliente siempre y cuando tenga espacio, en caso de no tenerlo el método devuelve **false**. El objeto **cuenta**, ya inicializado, es pasado al método mediante el parámetro **pCu**. El método buscará entre las cuentas del cliente aquella cuyo nº de cuenta coincida con el que posee la cuenta pasada por parámetro. Si no la encuentra, asignará el objeto **pCu** al final del vector de cuentas del cliente y devolverá **true** y en caso contrario el método solo devolverá **false**. \*/*

▪ **bool ActalizarCuenta(Cuenta pCu) ;**

*/\* Actualiza la cuenta del cliente con los datos de la **cuenta** pasada por parámetro. Para ello, el método buscará aquella cuenta del cliente cuyo nº de cuenta coincida con el que posee el objeto **cuenta pCu**. Si la encuentra la actualizará y devolverá **true**, en caso contrario devolverá **false**. \*/*

▪ **bool BorrarCuenta(int pNoCuenta) ;**

*/\* Elimina la cuenta del cliente cuyo nº de cuenta es pasado por parámetro. Si la encuentra, la eliminará del vector de cuentas y devolverá **true**, en caso contrario devolverá **false**. \*/*

▪ **int DameNoCuentas() ;**

*// Devuelve el número de cuentas que posee el cliente.*

▪ **Cuenta DameCuenta(int pos) ;**

*// Devuelve la **cuenta** del cliente que está en la posición **pos**.*

▪ **void Mostrar(char Campo) ;**

*/\* Muestra el **nombre** y todas las cuentas del cliente según indique el parámetro **Campo**. Si **Campo** contiene '**d**' mostrará el **nombre** y la **dirección** del cliente, si contiene '**c**' mostrará toda la información de sus cuentas (nº, saldo y si está o no bloqueada), y si contiene '**t**' mostrará el **nombre**, la **dirección** y toda la información de sus **cuentas**. \*/*



Diseñe las siguientes **funciones**:

- **int BuscarCliente(Cliente Ctes[MAX\_CLIENTES], int NCtes, Cadena Nombre);**

*/\* Busca un cliente en el vector **Ctes** cuyo nombre coincida con el nombre pasado por el parámetro **Nombre**. El parámetro **NCtes** indica el nº de elementos del vector **Ctes**. Devolverá la posición del cliente encontrado o **-1** si no lo encuentra. \*/*

- **int Menu();**

*/\* Muestra por pantalla el menú indicado y devolverá la opción seleccionada, que deberá ser correcta \*/*

```
--- Menú Principal ---
1 Añadir un cliente
2 Actualizar Dirección del Cliente
3 Mostar un cliente
4 Mostar todos los clientes
5 Submenú Gestión de Cuentas
6 Salir
Elige Opción:
```

- **int MenuCuentas();**

*/\* Muestra el menú que se indica y devuelve la opción seleccionada, que deberá ser correcta \*/*

```
--- Menú Gestión de Cuentas ---
1 Añadir una cuenta a un cliente
2 Mostrar las cuentas del cliente
3 Borrar una cuenta del cliente
4 Modificar Saldo de una cuenta
5 Modificar Estado de una cuenta
6 Salir
Elige Opción:
```

Diseñe un **main()** que compruebe el funcionamiento correcto de los métodos de esta clase **Cliente**. Para ello definirá las siguientes variables locales de contendrán los **clientes de un banco**:

```
Cliente Datos[MAX_CLIENTES];
int nClientes;
```

**Datos** es un vector de objetos de tamaño **MAX\_CLIENTES** y **nClientes** contiene el nº de objetos del vector que se están utilizando.

La función `main()` mostrará el menú y realizará las siguientes acciones según la opción elegida:

**Opción 1.**

Solicitará por teclado el nombre y la dirección del cliente a crear, a continuación, actualizará con estos datos el objeto correspondiente del vector de clientes.

**Opción 2.**

Solicitará por teclado el nombre del cliente a actualizar, lo buscará en el vector de clientes y si lo encuentra solicitará la dirección y lo actualizará con dicha información. Si no lo encuentra mostrará un mensaje de error.

**Opción 3.**

Solicitará por teclado el nombre del cliente a mostrar y después lo buscará en el vector de clientes. Si lo encuentra muestra toda su información, si no mostrará un mensaje de error.

**Opción 4.**

Mostrará todos datos de los clientes, así como todas sus cuentas corrientes.

**Opción 5.**

Solicitará el nombre del cliente cuyas cuentas van a ser gestionadas. Si lo encuentra dentro del vector de clientes mostrará un submenú con el mismo contenido que el ejercicio anterior (todas las operaciones descritas utilizarán el objeto del vector cliente que ha sido localizado), si no lo encuentra mostrará un mensaje de error.

**Opción 6.**

Termina el programa.

#### Ejercicio 4. Búsqueda del Tesoro

Implementar la clase **Tesoro**, que nos permita jugar a buscar un tesoro rodeado de minas a distinta profundidad. Para ello contará con:

##### Atributos privados:

- **Tablero**

- /\* Matriz de 5x5 que almacenará en cada casilla:*

- Una letra, indicando si el contenido es el tesoro ('T'), una bomba ('B') o arena ('A')
  - Un valor entero, indicando la profundidad de la bomba. Habrá 3 niveles de profundidad (1, 2 ó 3, de menos a más profundo). Las bombas más superficiales son las que más puntos quitan. \*/

- **Tiradas**

- /\* Matriz de 5x5 elementos de tipo char, que almacenará los distintos intentos (tiradas) realizados por el jugador para descubrir el tesoro. Los valores posibles para cada casilla serán:*

- '-' que indicará que la casilla no ha sido descubierta (seleccionada) aún por el jugador.
  - '1', '2' o '3' que indicarán que la casilla ha sido descubierta y que en ella había una bomba a dicha profundidad.
  - 'T' indicando que la casilla ha sido descubierta y en ella está el tesoro. \*/

- **Puntos**

- // Valor entero con los puntos conseguidos por el jugador.*

- **Intentos**

- // Valor entero con el nº de intentos que le quedan al jugador.*

##### Métodos privados:

- **void CasillaArena(int &f, int &c);**

- /\* Devuelve en f y c la fila y columna respectivamente de una casilla, generada aleatoriamente, en la que hay arena. \*/*

- **void InsertarBombas(int cant, int prof)**

- /\* Inserta la cantidad de bombas indicada por cant a la profundidad establecida por prof. Dichas bombas deben insertarse solo en casillas en las que haya arena, por lo tanto este método hará uso de CasillaArena(). \*/*

##### Métodos públicos:

- **Tesoro()**

- /\* Se establecen en 15 tanto los puntos iniciales como el nº de intentos, además se rellena todo el tablero con arena y se indica en tiradas que aún no se ha descubierto ninguna (se rellena con guiones). \*/*

- **bool Configurar(int nbp1, int nbp2, int nbp3)**

- /\* Se inicializa el tablero de juego insertando 1 tesoro, nbp1 bombas a profundidad 1, nbp2 bombas a profundidad 2 y nbp3 bombas a profundidad 3, en casillas con arena seleccionadas aleatoriamente.*

El número máximo de bombas que se pueden insertar es **12**, de manera que antes de insertar habrá que comprobar dicha cantidad. Si se sobrepasa el tablero no se inicia y el método devuelve **false**, en caso contrario se insertan el tesoro y las bombas y devuelve **true**. Este método debe hacer uso de **InsertarBombas()**.\*/

- **void MostarTiradas();**  
// Visualiza en formato 2D la matriz Tiradas.
- **int getPuntos();**  
// Devuelve el valor del atributo Puntos.
- **int getIntentos();**  
// Devuelve el valor del atributo Intentos.
- **bool Tirada(int f, int c);**  
/\* Recibe como parámetros de entrada las coordenadas **[f,c]** de la casilla seleccionada por el jugador para ser descubierta. Ambas coordenadas están en el rango **[1,5]**.  
Se decrementará el número de intentos que le quedan por utilizar al jugador y se actualizará el valor de la casilla **[f,c]** de la matriz **Tiradas** en función del contenido de dicha casilla en la matriz **Tablero**. Además se actualizarán los puntos conseguidos en función de la casilla descubierta, del siguiente modo:  
a) Si se ha encontrado el tesoro, el jugador suma **100** puntos.  
b) Si se ha encontrado arena, se le añade **1** punto a los ya acumulados.  
c) Si se ha encontrado una bomba, se le restará la siguiente cantidad de puntos a los ya acumulados **4-profundidad**.  
Si se hubiera encontrado el tesoro el método devolverá **true**, en caso contrario devolvería **false**. \*/
- **void DescubrirTablero()**  
/\* Mostrará por pantalla el contenido del tablero del siguiente modo:

A	B(2)	A	A	A
A	B(2)	A	A	A
A	B(1)	A	T	A
A	A	A	A	B(3)
A	A	B(3)	B(3)	A

Por lo tanto, se indicará:

- Una **A** para la arena.
- Una **T** para el tesoro.
- Una **B(profundidad)** para las bombas. \*/

Una vez implementada la clase, **codificar el programa principal** que permita su utilización para jugar a la **Búsqueda del Tesoro**.

El juego constará de los siguientes pasos:

- 1) Se **configurará** el juego solicitándole al jugador la cantidad de bombas que quiere insertar de cada profundidad.
- 2) Se **mostrará** la matriz **Tiradas** (que inicialmente estará con todas las casillas por descubrir), los **puntos** de partida y los **intentos** que le quedan.

```
- - - - -  
- - - - -  
- - - - -  
- - - - -  
- - - - -  
  
Puntos acumulados ... 15  
Intentos restantes .. 15
```

- 3) Se le **pedirá** al jugador la **fila** y la **columna** de la **casilla que quiere descubrir**. Si los valores no están en el rango **[1,5]** se ignorará y se volverán a solicitar (previo mensaje indicando la situación al jugador).

Para simplificar el ejercicio, el juego no controlará que la casilla indicada no haya sido descubierta previamente, por lo tanto si el jugador repite coordenadas, contará como una tirada y descontará los puntos correspondientes.

- 4) Se realizará la **tirada** indicada por el jugador y en función de la misma se actuará del siguiente modo:
  - a. Si **no se ha encontrado el tesoro** y aún **quedan intentos**, se mostrará la matriz **Tiradas**, los **puntos acumulados** y los **intentos restantes**, para posteriormente volver al paso 3 (pedir las coordenadas de otra casilla).

```
A - - - - -  
- - - - -  
- - - - -  
- - - - -  
- - - - -  
  
Puntos acumulados ... 16  
Intentos restantes .. 14
```

- b. Si **no se ha encontrado el tesoro** y ya **no quedan más intentos**, se le muestra al jugador un **mensaje** en el que se le indica que ha perdido y se **descubre el tablero**, para que el jugador sepa dónde estaban las bombas (con sus correspondientes profundidades) y el tesoro.
- c. Si **se ha encontrado el tesoro**, se muestra la matriz **Tiradas**, se le da la **enhorabuena** y se le indican los **puntos conseguidos** y los **intentos** que le han **sobrado**.

## Ejercicio 5. Tres en Raya

Se quiere realizar el juego del **Tres en Raya (tic tac toe)** para lo cual se le solicita al alumno las implementaciones que se detallan a continuación:

1) La clase **TicTacToe** con los siguientes atributos y métodos:

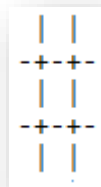
### Atributos privados:

- `char Tablero[3][3];`  
*/\* Contendrá la partida realizada. Los valores posibles para sus casillas serán: espacio en blanco, X y O \*/*

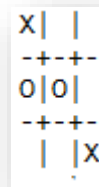
### Métodos públicos:

- `Tictactoe();`  
*/\* Hará una llamada a **LimpiarTablero()** de modo que cuando se cree el tablero quede preparado para jugar. \*/*
- `void LimpiarTablero();`  
*// Asignará a todas las casillas del tablero el valor ' '*
- `void Pintar();`  
*// Mostrará el contenido del tablero del siguiente modo:*

Tablero vacío:



Tablero con algunas tiradas:



- `bool PonerFicha(char ficha, int fila, int columna);`  
*/\* Si en la posición (fila, columna) hay un espacio en blanco, se le asignará a dicha posición el valor contenido en el parámetro ficha (X u O) y se devolverá true, en caso contrario no se podrá anotar la jugada y se devolverá false. \*/*
- `bool ComprobarFila(char ficha, int fila);`  
*/\* Devuelve true si se ha conseguido realizar 3 en raya con la ficha y en la fila indicadas como parámetros. En caso contrario devolverá false. \*/*
- `bool ComprobarColumna(char ficha, int columna);`  
*/\* Devuelve true si se ha conseguido realizar 3 en raya con la ficha y en la columna indicadas en los parámetros. En caso contrario devolverá false. \*/*

- **bool ComprobarDiagonales(char ficha, int fila, int columna);**

*/\* Devuelve **true** si en la diagonal o diagonales (en el caso de tratarse de la casilla central del tablero) en la que se encuentra la casilla marcada por los parámetros **fila** y **columna** se ha conseguido 3 en raya con la **ficha** indicada, y **false** en caso contrario. \*/*

- **bool TableroCompleto();**

*/\* Devuelve **true** si no se puede seguir jugando por no quedar espacios en blanco, y **false** en caso contrario. \*/*

- 2) La función **void PedirPosicion(char ficha, int &fila, int &columna);**

*/\* Muestra un mensaje al jugador indicado por el parámetro **ficha**, solicitándole que introduzca una coordenada del tablero. Dicha coordenada será devuelta a través de los parámetros **fila** y **columna**. Esta función no podrá devolver una coordenada incorrecta, por lo que si el usuario indica un valor para **fila** o **columna** erróneo deberá indicárselo con un mensaje y volverlas a solicitar. El alumno podrá optar por enumerar las casillas, para el usuario, desde el **1** o bien desde el **0**. \*/*

- 3) Un **programa principal** que nos permita jugar al **Tres en Raya** haciendo uso de la clase y función indicadas anteriormente.

El programa permitirá jugar tantas veces como quieran los jugadores, de modo que tras finalizar cada partida, preguntará si se desea volver a jugar, actuando en consecuencia.

Al comenzar la partida se mostrará el tablero (que estará en blanco), y la primera ficha en poner será siempre la **x**. Cada vez que un jugador indique una coordenada válida el proceso a seguir será el mismo:

- 1º Se comprueba si la ficha se puede colocar en el tablero, pues sólo estarán disponibles las casillas vacías.
- 2º Se muestra cómo queda el tablero con la tirada realizada por el jugador al que le corresponda.
- 3º Se comprueba si se ha conseguido realizar 3 en raya:
  - Si se ha conseguido, se le muestra un mensaje de enhorabuena al jugador que ha ganado y la partida finaliza.
  - Si no se ha conseguido, se comprueba si el tablero se ha completado:
    - Si estuviera completo, se mostrará un mensaje por pantalla indicando que los jugadores han empatado.
    - En caso contrario, se cambiará de jugador para realizar la siguiente tirada.
- 4º Si la partida ha finalizado se le preguntará a los jugadores si quieren volver a jugar, en caso de no haber finalizado se realizará otra tirada por parte del jugador al que le corresponda.