Universidad de Jaén
Departamento de Informática

Programación Orientada a Objetos

http://tiny.cc/poouja

Curso 2022/2023

José Ramón Balsas Almagro
Víctor Manuel Rivas Santos
Ángel Luis García Fernández
Juan Ruiz de Miras

Práctica 6. Relaciones de herencia

Objetivos

- Saber realizar generalizaciones-especializaciones de clases utilizando herencia
- Conocer las peculiaridades y saber implementar el constructor de copia y el operador de asignación en clases derivadas cuando existen relaciones de herencia en el diseño
- Conocer y saber manejar el concepto de flujo para el almacenamiento y recuperación de información desde dispositivos de almacenamiento masivo.
- Saber utilizar flujos en C++ para el almacenamiento de datos en archivos de texto.

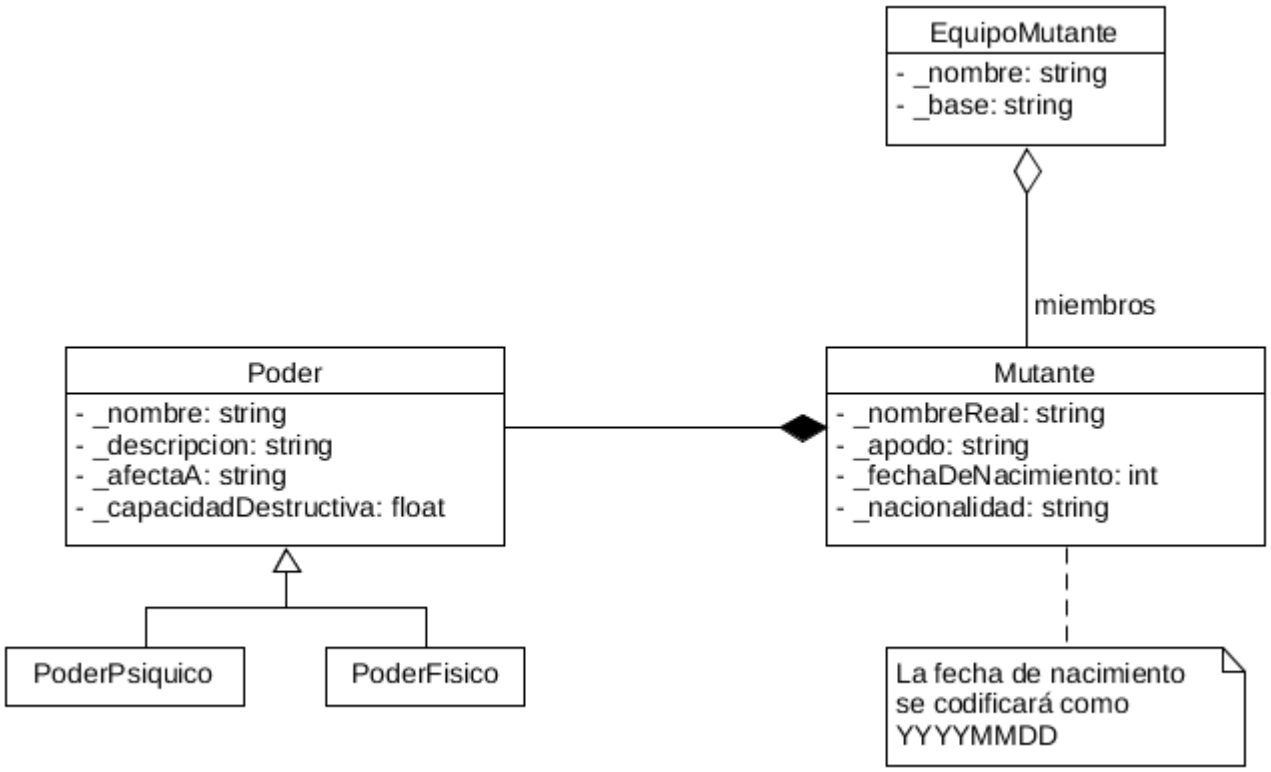
Índice

Contexto de la práctica

S.H.I.E.L.D. es una organización secreta que busca combatir el terrorismo internacional. Para ello, ha decidido sacar partido de los mutantes (personas que, debido a una serie de mutaciones genéticas, han desarrollado una serie de poderes, físicos o psíquicos, que les permiten realizar hazañas que no están al alcance de las personas corrientes) que viven de incógnito en distintos

lugares del planeta, reclutándoles e integrándoles en equipos, de forma que sus poderes se complementen y consigan así ser fuerzas de ataque y defensa poderosas ante las organizaciones que intenten atacar los países aliados de S.H.I.E.L.D.

Para poder gestionar de la mejor forma posible la información relativa a estos mutantes, los informáticos de S.H.I.E.L.D. han diseñado una aplicación, cuyo diagrama UML se puede ver a continuación:



A partir de la estructura de clases propuesta, se debe desarrollar la aplicación lo antes posible, ya que la amenaza terrorista en los últimos meses es más importante que nunca. Los programadores de S.H.I.E.L.D. no dan abasto con todo el trabajo pendiente, así que solicitan tu ayuda para terminar la aplicación en un tiempo récord.

Punto de partida

El punto de partida de esta práctica son las clases ya escritas por el equipo de programadores de S.H.I.E.L.D. El proyecto está disponible en el siguiente enlace: http://bit.ly/poouja1617pr7

Ejercicios

1. Define e implementa la clase **Poder**, con los atributos indicados en el diagrama UML. Incluye en la clase métodos observadores (*getters*) y modificadores (*setters*) para los

atributos, así como constructores de copia y parametrizado, operador de asignación y métodos *toCSV()*.

2. Modifica la clase **Mutante**, de forma que se represente la relación de composición entre esta clase y la que has creado en el ejercicio 1. Incluye en la clase Mutante los métodos **addPoder (string nombre, string descripcion, string afectaA, float capacidadD)** y **borraPoder (int cual)**, que permitirán, respectivamente, añadir y quitar poderes a un mutante. Implementa también el método **capacidadDestructivaTotal ()**, que devolverá la suma de la capacidad destructiva de los poderes del mutante.

3. Define e implementa las clases **PoderPsiquico** y **PoderFisico** como especializaciones de la clase Poder. Inicialmente, estas clases no incluyen atributos adicionales, pero es necesario implementar los constructores (por defecto, parametrizado y de copia) y el operador de asignación, tal y como se explica en la sección Contenidos. Añade a la clase Mutante los métodos **addPoderFisico** y **addPoderPsiquico**; cada uno de ellos recibirá los mismos parámetros que el método *addPoder* antes implementado, pero en lugar de crear objetos de clase Poder, crearán objetos de clase PoderFisico o PoderPsiquico, según corresponda.

4. Crea tres nuevos métodos sobrecargados de la clase Mutante: **addPoder(const Poder&), addPoder(const PoderFisico&) y addPoder(const PoderPsiquico&)** que añadan al mutante una copia del poder respectivo suministrado. La ventaja de estos métodos respecto a los ya existentes en la clase es que, de esta forma, si se añaden nuevos atributos a cualquiera de los tipos de poder, no habrá que cambiar los parámetros de los métodos ya existentes, puesto que se delega en los constructores de copia de dichas clases la tarea de copiar la información de un objeto a otro.

5. Modificar las funciones **Visualiza(Mutante&)** y **Visualiza(EquipoMutante&)** del archivo *main.cpp* para que muestren todos los poderes de un mutante en formato CSV y todos los miembros de un equipo respectivamente.

6. Implementar en el archivo *main.cpp* la función **almacenaMutantesCSV(Mutante* v[], int tamv, std::string nombreArchivo)** que permita almacenar en un archivo, en formato CSV, la información de todos los mutantes del vector.

7. Modifica la función **main**, de forma que se realicen las siguientes tareas:
   - Crear un vector de 5 punteros a mutantes y los objetos correspondientes, asignándoles los valores a sus atributos directamente en el código fuente. Añadir a los mutantes diferentes poderes.
   - Crear dos equipos mutantes, asignando al primero los mutantes que ocupan las posiciones pares del vector, y al segundo los de las posiciones impares.
   - Se muestre por la consola la información de cada equipo mutante
   - Utilice la función del ejercicio 6 para almacenar todos los datos de los mutantes del vector en un archivo en formato CSV.
   - Finalmente, se destruyan todos los objetos creados en memoria dinámica antes de la finalización del programa

Contenidos

Relación de generalización-especialización (herencia)

Una de las relaciones más características dentro de la POO es la de generalización-especialización, que se traduce a la hora de programar en la utilización de herencia para derivar nuevas **subclases** a partir de una **superclase**.

Dado que el concepto de herencia lleva asociado otro conjunto de conceptos más complejos, haremos en esta práctica una primera aproximación que consistirá exclusivamente en la derivación de nuevas subclases.

Cuando hablamos de especialización de una clase nos referimos al hecho de que a la hora de plantear nuestro diseño, por algún motivo en particular, observamos que existen objetos de la clase que tienen características y/o comportamientos específicos además de las características y comportamiento de la clase a la que pertenecen. En esta situación, podemos utilizar la relación de herencia para derivar una clase "especializada" que añada estado y/o comportamiento al de la clase padre. De esta forma, reutilizamos el código de esa clase padre y añadimos nuevos atributos o métodos que complementen o sobrecarguen el comportamiento de los objetos de la nueva clase hija.

La derivación de una subclase se realiza fácilmente de la siguiente forma:

```
/* ---
Ejemplo de derivación de clases
--- */

// Definición de la superclase
class ComponenteOrdenador
{  private:
      float _precio = 0;
   public:
      ComponenteOrdenador();
      ComponenteOrdenador( const float precio );
      ~ComponenteOrdenador();
};
```

```
// A continuación especializamos la superclase para
// identificar un tipo específico de componentes

// Definición de la subclase
class Microprocesador: public ComponenteOrdenador
{  private:
      int _nucleos = 1;
```

```
   public:
      Microprocesador();
      Microprocesador( const float precio, const int nucleos);
      ~Microprocesador();
};
```

Hay que recordar que la clase derivada únicamente tendrá acceso a los atributos y métodos públicos o protegidos de la superclase, pero no tendrá acceso a los privados. Existen varios motivos por los que se hace necesario que la clases hijas no accedan a los métodos y atributos privados: evitar que la subclase tenga dependencias con detalles de implementación que puedan variar en la superclase, simplificar la interfaz de uso de la superclase, garantizar que la subclase no altera la integridad de la información que gestiona la superclase, etc.

Así, en el ejemplo anterior, ningún objeto de tipo *Microprocesador* podrá acceder (ni para consultar ni para modificar) directamente al atributo definido como *precio*, sino que deberá hacerlo a través de los métodos GET y SET que podamos incorporar como métodos públicos de la superclase *ComponenteOrdenador*.

Es especialmente interesante el hecho de saber que todo objeto de la Subclase **es también un objeto de la Superclase**. Entendamos lo que esto significa: cada vez que instanciamos un objeto *Microprocesador* se llama a su constructor, pero también se deberá hacer una llamada al constructor de la superclase, es decir, de *ComponenteOrdenador*.

El siguiente código indica cómo es posible hacer una **llamada a un constructor específico de la superclase** cuando ésta tiene los constructores sobrecargados.

```
/* ---
Ejemplo de llamada a constructor de la superclase desde constructor de
la subclase
--- */
Microprocesador::Microprocesador ( float precio, int nucleos):
                                  ComponenteOrdenador ( precio )
                                   , _nucleos ( nucleos )
{ ... }
```

Por supuesto, si en el constructor de la subclase no se llama a ninguno de los constructores de la superclase, se ejecutará el constructor por defecto de esta última. No obstante, **se aconseja llamarlo expresamente** por motivos de claridad en el código.

Hay que tener en cuenta que, como el constructor de la clase padre podría lanzar cualquier tipo de excepción, en caso de que ésta quisiera capturarse, debería utilizarse una versión especial de la construcción **try/catch para la secuencia de inicialización** como se estudió en el Tema 2:

```
/* ---
Ejemplo de captura de excepción en secuencia de inicialización
--- */
```

```
Microprocesador::Microprocesador ( float precio, int nucleos)
try: ComponenteOrdenador ( precio ),
     _nucleos ( nucleos )
{

     … //código del constructor

} catch ( ComponenteException &e ) {
  std::cerr << " Error al construir un microprocesador: "
            << e.mensaje();
  //Se relanza automáticamente e aunque no se indique:  throw e;
}
```

Sin embargo, también debe tenerse en cuenta que si el constructor de la clase padre lanza una excepción, como no se ha podido inicializar el estado heredado de dicha clase, el objeto de la clase derivada que se está tratando de inicializar ya **no se podrá construir** finalmente. Por lo tanto, debe recordarse que, en este caso en particular de construcción try/catch, **se relanzará** (*throw e;*) **automáticamente** la excepción aunque hubiera sido capturada en el catch. Por lo tanto, si no es necesario capturar esta excepción en el constructor de la subclase[1], es preferible simplemente evitar el uso de esta construcción try/catch con lo que la implementación del constructor quedaría más sencilla.

Constructor de copia en clases derivadas

El constructor de copia es el encargado de construir una copia de un objeto a partir de otro de la misma clase. Su funcionamiento consiste en realizar la copia de cada uno de los atributos del objeto original en el nuevo objeto copiado. En el caso de clases derivadas, es necesario redefinir el constructor de copia heredado de la superclase siempre y cuando se añadan nuevos atributos a la superclase que también deban ser copiados. En este caso, al igual que ocurre con los constructores de clases derivadas, debemos hacer uso de la implementación del constructor de copia de la superclase y posteriormente realizar la copia de los nuevos atributos de la subclase. Al igual que ocurre con el resto de constructores, la llamada a otros constructores se realiza en la lista de inicialización del constructor:

```
/* ---
 Ejemplo de llamada al constructor de copia de la superclase Poligono
desde el constructor de copia de la subclase Triangulo
NOTA: en este ejemplo, los métodos se implementan inline para reducir el
espacio ocupado por el código. En las clases que implementéis, tendréis
que escribir la cabecera en la definición de la clase (archivo .h), y la
implementación en el archivo .cpp, como hasta ahora habéis hecho
--- */
class Poligono
```

[1] Por ejemplo: un uso para este tipo de captura podría ser lanzar otra excepción con información relativa a la clase derivada en vez de propagar hacia atrás en la pila de llamadas la excepción original que hace referencia a la clase padre. Solo relanzar podría confundir a quien la capturase posteriormente.

```
{  private:
      int _numLados = 0;
   public:
      Poligono ( const Poligono &orig ):
                  _numLados(orig._numLados) //constructor de copia del
atributo
     { ... }
};
```

```
class Triangulo: public Poligono
{  private:
      float _altura = 0;
   public:
      Triangulo ( const Triangulo &orig ):
          Poligono(orig)  //constructor de copia de la clase Poligono
           , _altura(orig._altura) //constructor de copia del atributo
altura
     { ... }
};
```

Ejemplos de utilización del constructor de copia en la clase derivada:

```
Triangulo tri1;   // Creación del objeto original
Triangulo tri2 ( tri1 ), tri3 = tri1;   // Creación de dos copias
```

Operador de asignación en clases derivadas

El operador de asignación es el encargado de copiar los valores de los atributos de un objeto en otro cuando se asigna el primero al segundo. Puesto que sólo se encarga de copiar los atributos de la clase para los que está definido, **debe redefinirse** si la subclase tiene atributos adicionales que deban ser también copiados. El motivo es para evitar que, en caso de que no se implementara en la subclase, no se puedan hacer asignaciones “parciales” de objetos de la subclase en los que sólo se copiaran los atributos de la superclase, dejando los de la subclase sin inicializar. Además, es necesario llamar al operador de asignación de la superclase, puesto que en caso que esta tuviera atributos privados, sería la única forma de copiarlos en el objeto de la subclase.

```
/* ---
Ejemplo de utilización del operador asignación de la clase padre
Poligono desde el operador de asignación de la clase hija Triangulo
NOTA: en este ejemplo, los métodos se implementan inline para reducir el
espacio ocupado por el código. En las clases que implementéis, tendréis
que escribir la cabecera en la definición de la clase (archivo .h), y la
implementación en el archivo .cpp, como hasta ahora habéis hecho
--- */
```

```
class Poligono
{  private:
      int _numLados = 0;
   public:
      Poligono &operator= ( const Poligono &orig )
      {  //sólo copiamos si asignamos a un objeto diferente!
         if ( this != &orig )
         {  _numLados = orig._numLados;
         }
         //Devolver ref. a objeto actual para asignaciones en cascada
         return *this;
      }
};
```

```
class Triangulo: public Poligono
{  private:
      float _altura = 0;
   public:
      Triangulo &operator= ( const Triangulo &orig )
      {  if (this!=&orig)
         {  //Llamada expresa al operador de asignación de la superclase
            //¡Aquí no hay acceso al atributo heredado numLados!
            this->Poligono::operator= ( orig );
            //Copiamos atributos locales
            _altura = orig._altura;
         }
         return *this;
      }
};
```

Ejemplo de uso del operador asignación:

```
Triangulo tri1, tri2, tri3;
tri1 = tri2;
tri3 = tri1 = tri2;
```

Manejo de Flujos

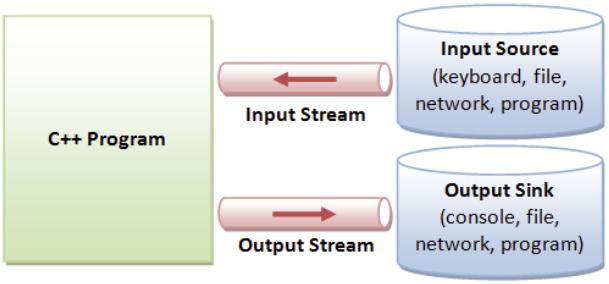
Hasta este momento, en las prácticas anteriores siempre hemos creado los objetos con información introducida por el usuario a través de la consola o directamente asignando valores en el propio código a través de parámetros en constructores o métodos de tipo *set*. Sin embargo, en una aplicación real, debido a la cantidad de datos con los que normalmente se trabaja, lo normal es recurrir a algún dispositivo o servicio de almacenamiento donde depositar de forma persistente o

desde el que recuperar la información de nuestras entidades.

En esta práctica vamos a comenzar con el estudio del sistema más sencillo de almacenamiento persistente: **los archivos**. Los archivos (o ficheros) son el mecanismo que nos proporciona el sistema operativo del ordenador para el almacenamiento de información en un dispositivo de almacenamiento secundario, como puede ser el caso de un disco duro, una memoria USB, un disco SSD, un DVD, etc. El concepto de archivo trata de abstraer la complejidad y las diferencias específicas del hardware que tienen los distintos tipos de dispositivos de almacenamiento secundario, proporcionando una forma homogénea a los programas y usuarios de agrupar y acceder a la información.

Los **flujos** (*Streams*) son una abstracción habitual existente en lenguajes de programación orientada a objetos para el almacenamiento o acceso de información en archivos. Los flujos establecen una interfaz de uso homogénea que permite simplificar manipulación de archivos ocultando los detalles específicos de la comunicación con el dispositivo.

En un programa, podemos entender un flujo como un **objeto** con el que podemos intercambiar información utilizando ciertos **métodos y operadores** disponibles. Estos métodos, dependiendo del tipo de flujo, permiten adaptar la información intercambiada haciendo las transformaciones de tipo que sean necesarias. Para utilizar un flujo en C++, normalmente debemos construir un objeto del tipo específico y, en ocasiones, pasar a su constructor información sobre el tipo de dispositivo al que va asociado. No obstante, en el caso de la consola en modo texto de un programa, existen dos objetos creados e inicializados para su uso: ***cin*** y ***cout***.



Flujos de datos. Fuente: Hock, Chua (2013)

También es posible crear flujos de entrada o salida que trabajen directamente sobre ficheros del ordenador, para así leer o escribir respectivamente información de los mismos. Para este tipo de operaciones, debemos crear objetos de las clases ***ifstream***, ***ofstream*** o ***fstream***, dependiendo del tipo de operaciones que realicemos sobre los mismos (lecturas, escrituras o ambas, respectivamente). Cuando creamos un objeto de alguna de estas clases, debemos indicarle la ruta del fichero sobre el que realizaremos las operaciones. A partir de este momento, el objeto flujo construido permite acceder al contenido del fichero asociado utilizando los métodos y operadores que definen las clases *fstream*, *ifstream* y *ofstream*.

En el caso de que trabajemos con **ficheros de texto**, podemos hacer una analogía entre el uso de los flujos asociados y el manejo que ya conocemos de los flujos de entrada/salida estándar: *cin* y

cout. Es decir, **podemos hacer uso de los métodos y operadores conocidos** (*get*, *getline*, *put*, >>, <<, *flush*, etc.) para recibir y enviar información desde y hacia el fichero, **tal y como se haría con la consola de texto** de la aplicación. De hecho, una forma de familiarizarse con el manejo de flujos de texto es utilizar la salida estándar *cout* para mostrar los datos de la forma en que nos gustarían que aparecieran en el fichero. Una vez que conseguimos el resultado deseado, podemos sustituir el objeto *cout* por un flujo de salida (*ofstream*) asociado a un fichero donde queremos que se almacene la información.

No obstante, aunque el uso de flujos asociados a ficheros es similar a al uso de los objetos cin y cout, hay que tener en cuenta algunos aspectos específicos para su utilización:

- Debe indicarse la **ruta del fichero** al que estarán asociados. Esto puede hacerse en el constructor del objeto o, posteriormente, mediante el método *open*(). En ambos casos, la cadena de caracteres debe tener la sintaxis de C; es decir: "nombre del fichero" o, si se trata de una variable string, utilizar el método *c_str*()[2]. Si sólo se indica el nombre del fichero, se considera que estará ubicado **en la carpeta donde se encuentre el ejecutable**. Por ejemplo:

```
fstream fdatos ( "misdatos.txt" ); // usando el constructor
fstream f;
f.open ( nomArchivo.c_str() );      // usando open y c_str
```

- Las operaciones sobre flujos asociados a ficheros **no siempre pueden realizarse** debido a diferentes motivos: falta de espacio en disco, error de lectura/escritura en disco, no hay más datos en el fichero (al leer datos), etc. Por lo tanto, es necesario comprobar si cada operación se ha realizado correctamente utilizando alguno de estos métodos:
  - ***good*()**, devuelve *true* si no ha ocurrido ningún problema
  - ***eof*(),** devuelve *true* cuando se ha llegado al final de un archivo (en una operación de lectura)
- Finalmente, cuando un flujo deja de utilizarse, es necesario cerrarlo utilizando el método *close*() para evitar que pueda quedarse información sin guardarse físicamente en el disco o para que el archivo pueda utilizarse desde otros programas en el sistema operativo.

Por ejemplo, podríamos utilizar el siguiente fragmento de código para crear un fichero de texto con varias líneas precedidas por un número:

```
#include <fstream>

int main ()
{  string lineas[] = { "primera línea", "segunda línea"
                     , "tercera línea"
                    };
   ofstream f;
```

[2] A partir de C++11 también es posible pasar únicamente un objeto string al método open de un stream sin necesidad de utilizar su método c_str()

```
    string nombreArchivo = "prueba.txt";

    // el nombre de archivo se pasa como una cadena en C
    f.open ( nombreArchivo.c_str() ); // c_str() opcional en C++11
    //Comprobamos que se ha podido abrir el fichero
    if ( f.good() )
    {  for ( int i = 0; i < 3; i++ )
       {  //El stream sabe como transformar un entero,
          // i+1, en una cadena de caracteres (igual que cout)
          f << i+1 << ".- " << lineas[i] << endl;
       }
       f.close();
    }
    else
    {  cerr << "Error de apertura de archivo";
    }
    return 0;
}
```

Que generaría un fichero con el nombre *prueba.txt* con el contenido:

```
1.- primera línea
2.- segunda línea
3.- tercera línea
```

Para más información sobre utilización de archivos pueden consultarse el tema 10 de los apuntes de la asignatura de Fundamentos de programación y la conocida página http://www.cplusplus.com/doc/tutorial/files/

Almacenamiento de objetos en formato CSV

Como comentábamos en las prácticas anteriores, CSV es un formato simple de representación de información estructurada que puede ser útil para almacenar o recuperar nuestros objetos desde un dispositivo de almacenamiento. Por dicho motivo, se estudió cómo dotar a los objetos de métodos *toCSV()* y *fromCSV(string)* para, respectivamente, obtener la representación en CSV del estado de un objeto y, posteriormente, poder usar dicha información para recuperar el objeto original.

Dado que CSV permite representar la información de un objeto como una cadena de caracteres, en este apartado veremos cómo utilizar ficheros de texto (con extensión .csv) para almacenar de forma simple los objetos de nuestro programa, de forma que podamos recuperarlos en ejecuciones posteriores. Además, al utilizar este formato, podremos también utilizar otras aplicaciones (Microsoft Excel, LibreOffice Calc, Bloc de notas) para procesar estos contenidos.

Continuando con el ejemplo anterior de escritura en un fichero, podríamos adaptarlo fácilmente de la siguiente forma para almacenar en formato CSV una colección de objetos en un fichero de texto. De esta forma, obtendremos un fichero donde en cada línea almacenaremos la representación CSV

de cada objeto:

```
#include <fstream>

int main ()
{  Plato primeros[10];

   //... Asignamos información a cada plato con métodos set

   // Almacenamos los platos en un fichero en formato CSV
   ofstream f;
   f.open ( "platos.csv" );

   if ( f.good() )
   {  for ( int i = 0; i < numPrimeros; i++ )
      {  //cada objeto se almacena en una línea
         f << primeros[i].toCSV() << endl;
      }
      f.close();
   }

   return 0;
}
```

Una vez creado el fichero, podemos consultar la información almacenada utilizando algún editor de texto simple, p.e. el editor de Clion[3], el bloc de notas de Windows, o abrirlo utilizando una hoja de cálculo como Microsoft Excel o LibreOffice Calc.

En la siguiente sesión estudiaremos cómo realizar la operación inversa y recuperar la información de dicho fichero y usarla para inicializar adecuadamente objetos de nuestro programa.

[3] CLion permite editar los ficheros .csv tanto en formato texto como en formato tabla +información