

**Programación con estructuras lineales**



**Actividad colaborativa 1**

04/10/2021

*Marcos Eladio Somoza*

*Fernando Moreno López*

*Alex Tensa Jiménez*

*Carlos de Martin Juan*

**ÍNDICE**

[1. Enunciado 4](#_Toc84330237)

[2. Código 5](#_Toc84330238)

[2.1 Book 5](#_Toc84330239)

[2.2 Person 6](#_Toc84330240)

[2.3 User 7](#_Toc84330241)

[2.4 Library 9](#_Toc84330242)

[2.5 LibraryApp 15](#_Toc84330243)

# Enunciado

Una Biblioteca pretende crear un software para gestionar la entrada/salida de sus libros ya que, su logística todavía se escribe en ficheros Excel deWindowsXP (lo cual en si ya es un avance a nivel logístico, pero resulta difícil gestionar los datos de entrada y salida y el ordenador de la biblioteca no da más de sí. Así pues, la biblioteca nos pide las siguientes funcionalidades:

* Tener una "biblioteca" con al menos 20 ejemplares separados por categorías (acción, aventuras, drama, comedia, romanticismo...)
* Tener control de qué libros están o no disponibles y en caso negativos a ver quién lo tiene.
* Tener un registro de usuarios con sus datos y el historial de lectura:
* Nombre, apellidos, DNI....
* Libro que tiene ahora
* Historial...

Todo ese software pretende facilitarle la vida a los bibliotecarios que hacen las gestiones de reservas de libros y les tiene que permitir realizar las siguientes acciones:

• Comprobar qué libros hay disponibles

• Ver el historial de libros de una persona

• Ver si una persona tiene un libro o no

• Sacar un libro (asignárselo a una persona)

• Devolver un libro (desasignárselo a dicha persona

{\displaystyle \log ,}

# Código

La arquitectura se basa en una librería que contiene usuarios y libros, donde se pueden realizar diferentes tareas. Así pues, debe haber una clase usuario, una libro y una librería como mínimo. Aunque, como veremos a continuación, es conveniente crear alguna más. Para mantener un código limpio y seguir los principios ***S.O.L.I.D***, se ha separado cada clase en su fichero ***.h*** y ***.cpp***; a fin de guardar las declaraciones de atributos, constructores y métodos en el ***.h*** y sus respectivas funciones en el ***.cpp***. A fin de mantener este documento lo menos extenso posible, se ha omitido comentar los ficheros ***header*** (son auto explicativos y están disponibles entre los ficheros adjuntos).

## 2.1 Book

Cuenta con tres constructores, dependiendo de cómo se quiera crear un nuevo objeto ***Book***. El primero de ellos (el predeterminado ***Book()***) crea un libro con nombre y género vacíos (es utilizado cuando se quiere decir que un usuario no tiene ningún libro actualmente, a fin de no usar objetos ***null***).

Book::Book() {

this->name = "";

this->genre = "";

}

El siguiente constructor crea un objeto libro con un nombre y un género aleatorio, llamando al método ***set\_random\_book()*** de la misma clase. Este método selecciona un nombre y género aleatorio usando ***rand()***, mirando en los vectores de ***string*** ***names*** y ***genres*** ( un almacén de nombres y géneros aleatorios guardados como atributos en ***Book***).

Book::Book(bool random) { if (random)set\_random\_book(); }

void Book::set\_random\_book()

{

    this->name = this->names[rand() % 20];

    this->genre = this->genres[rand() % 8];

}

Finalmente, el ultimo constructor utiliza el ***id\_nombre*** y el ***id\_género*** pasados como parámetros para crear el objeto, accediendo a las posiciones correspondientes de ***names*** y ***genres***.

Book::Book(int id\_name, int id\_genre) {

    this->name = this->names[id\_name];

    this->genre = this->genres[id\_genre];

}

Para utilizar los atributos tanto del nombre como del género del libro, dado que son atributos privados, se han creado métodos ***getter*** para cada una de ellas ***get\_name()*** y ***get\_genre().***

string Book::get\_name() { return name; }

string Book::get\_genre() { return genre; }

Para ver el contenido del Book, se han creado dos métodos que o bien devuelven en formato ***string*** el nombre y el género en paréntesis, o bien lo imprime por consola; ***to\_string()*** y ***print()*** respectivamente.

std::string Book::to\_string() { return this->name + "(" + this->genre + ")"; }

void Book::print() { std::cout << "Book: " << Book::to\_string() << endl; }

Finalmente el método ***are\_equal()*** recibe como parámetro un objeto ***Book***. Compara el nombre del libro pasado como parámetro con el nombre actual así como sus géneros, en caso de que ambos sean iguales devuelve ***true***, sino, ***false***. Cabe destacar que el libro pasado por parámetros es una ***referencia*** a un objeto libro (necesario para su uso posterior en la clase ***Library***).

bool Book::are\_equal(const Book &book)

{

    bool equal = false;

    if (this->name == book.name && this->genre == book.genre)

        equal = true;

    return equal;

}

## 2.2 Person

***Person*** será la clase padre de ***User***, dado que todo usuario es una persona, nos permite encapsular la información base del ***name***, ***surname*** y ***dni*** manteniendo un código limpio. Constará de cuatro constructores distintos. El predeterminado asignará un ***name***, ***surname*** y ***dni*** vacíos (al igual que pasaba con ***Book***, para su posterior uso en ***Library***).

Person::Person()

{

    this->name = "";

    this->surname = "";

    this->dni = "";

}

Al igual que con ***Book***, el siguiente constructor genera un objeto Persona aleatorio llamando a ***set\_random\_person()***. Este método selecciona un nombre, apellido y dni aleatorios usando ***rand()***, mirando en los vectores de ***string*** ***names***, ***surnames*** y ***dnis***( nombres, apellidos y dnis aleatorios guardados como atributos en ***Person***).

Person::Person(bool random) { set\_random\_person(); }

void Person::set\_random\_person()

{

    this->name = this->names[rand() % 20];

    this->surname = this->surnames[rand() % 20];

    this->dni = this->dnis[rand() % 20];

}

El siguiente constructor usa al igual que ***Book*** *ids* como parámetros de entrada para crear el objeto, usándolos como posiciones de los vectores ***string names, surnames*** y ***dnis***. Estos *ids* son ***id\_name***, ***id\_surname***, ***id\_dni***.

Person::Person(int id\_name, int id\_surname, int id\_dni)

{

    this->name = names[id\_name];

    this->surname = surnames[id\_surname];

    this->dni = dnis[id\_dni];

}

Finalmente, el último constructor utiliza parámetros de entrada de tipo ***string*** para asignarlos directamente a los atributos ***name***, ***surname*** y ***dni***.

Person::Person(string name, string surname, string dni)

{

    this->name = name;

    this->surname = surname;

    this->dni = dni;

}

Dado que los atributos de nombre, apellido y dni son privadas, se han creado ***getters*** para cada una de ellas.

string Person::get\_name() { return this->name;

string Person::get\_surname() { return this->surname; }

string Person::get\_dni() { return this->dni; }

Finalmente, el método ***print\_info()*** muestra por consola la información del objeto.

void Person::print\_info() { cout << Person::get\_name() << " " << Person::get\_surn

ame() << " (" << Person::get\_dni() << ")" << endl; }

## 2.3 User

Como se ha mencionado anteriormente, ***User*** hereda de ***Person***, lo que quiere decir que se crea en base a ***Person***, generando una jerarquía de clases lógica heredando los atributos y métodos previos. Además, la clase ***User*** sigue la misma estructura en cuanto a constructores se refiere que las clases previamente analizadas, exceptuando que además crea una nueva instancia de ***Book*** vacía en el caso del constructor predeterminado.

User::User()

{

    Book b;

    this->current\_book = b;

}

Otro constructor que aleatoriamente genera un nuevo ***Book***, un nuevo ***Person***, y una lista de historial aleatoria, haciendo uso de sus métodos correspondientes ***set\_random\_book()***, ***set\_random\_person()*** y el constructor que genera un libro aleatorio de ***Book.***

User::User(bool random)

{

    if (random)

    {

        this->set\_random\_person();

        this->current\_book.set\_random\_book();

        for (int i = 0; i < rand() % 6; i++)

        {

            Book b(true);

            this->history\_books.push\_back(b);

        }

    }

}

y otro que asigna el ***Book*** pasado por referencia.

User::User(Book current\_book)

{

    this->current\_book = current\_book;

}

Además de tener un atributo de tipo ***Book current\_book***, tiene otro de tipo vector ***Book history\_books***, donde guarda una lista de los libros previamente leídos. En el último constructor, además de pasar un ***Book*** por referencia pasa también un ***vector Book*** para asignar al ***history\_books***.

User::User(Book current\_book, vector<Book> history\_books)

{

    this->current\_book = current\_book;

    this->history\_books = history\_books;

}

Esta vez, además de los ***getters*** para los atributos privados de ***current\_book*** y ***history\_books***, se ha creado un ***setter*** para asignar un libro al ***current\_book*** (más concretamente, asignar el objeto guardado en la dirección de memoria a la que apunta el puntero ***Book\**** dado).

Book User::get\_current\_book() { return this->current\_book; }

vector<Book> User::get\_history\_books() { return this->history\_books; }

void User::set\_current\_book(Book\* current\_book) { this->current\_book = \*current\_book; }

En cuanto a punteros se refiere, se ha creado un método más que añade el objeto está alocado en la dirección de memoria a la que apunta el puntero ***Book\**** dado al vector de ***history\_books***.

void User::add\_book\_to\_history(Book\* book) { history\_books.push\_back(\*book); }

También se encuentran los métodos para mostrar por consola la información del usuario. ***Print\_info*** devuelve la información de la persona mediante ***Person::print\_info()*** así como la del ***current\_book*** mediante ***User::current\_book.print()***.

void User::print\_info()

{

    Person::print\_info();

    User::current\_book.print();

}

O ***print\_history()***, que recorre el vector de ***history\_books*** y va llamando al método ***to\_string()*** de cada uno para mostrarlos por consola.

void User::print\_history()

{

    int counter = 1;

    for(Book b  : User::get\_history\_books())

    {

        cout << counter << "." << b.to\_string() << endl;

        counter++;

    }

}

## 2.4 Library

Esta clase es la encargada de toda la gestión de la librería. Cuenta con tres atributos clave: un vector de ***User***, un vector de ***Book*** y un map de ***Book\*, bool***. La primera será la lista de usuarios en la biblioteca, la segunda la lista de libros y la tercera un diccionario que asocia uno de dichos libros con su disponibilidad de recogida.

En cuanto a los constructores, cuenta con un único constructor que recibe dos enteros como parámetos ***books\_amount*** y ***users\_amount*** (el número de libros y el número de usuarios a crear). Con estos enteros, llama a sus métodos ***fill\_random\_books()*** y ***fill\_random\_users()*** así como a ***setup\_availability().***

Library::Library(int books\_amount, int users\_amount)

{

    Library::fill\_random\_books(books\_amount);

    Library::fill\_random\_users(users\_amount);

    Library::setup\_availability();

}

***fill\_random\_books()*** recorre de cero a ***books\_amount*** para ir creando un nuevo libro con el constructor de Book que recibe ***id\_name*** e ***id\_genre*** como parámetro (usando el contador del for como el primero y ***rand() % 8*** como el segundo). Una vez creado, llama al método ***add\_book()***, que añade el libro creado al vector de libros.

void Library::fill\_random\_books(int cuantity)

{

    for (int i = 0; i < cuantity; i++)

    {

        Book book(i,rand()%8);

        add\_book(book);

    }

}

void Library::add\_book(Book book) { this->books.push\_back(book); }

***fill\_random\_users()*** funciona de manera similar, recorriendo de cero a ***users\_amount*** para ir creando un nuevo ***User*** con su constructor que recibe tres ***id\_name***, ***id\_surname*** e ***id\_dni*** como parámetros (usando el contador del for con los tres). Una vez creado, llama al método ***add\_user()***, que lo añade al vector de usuarios.

void Library::fill\_random\_users(int cuantity)

{

    for (int i= 0; i < cuantity; i++)

    {

        User user(i,i,i);

        add\_user(user);

    }

}

void Library::add\_user(User user) { this->users.push\_back(user); }

Como se ha mencionado sobre el constructor, al final de este llama a ***setup\_availability()***. Este método se encarga de rellenar el diccionario de ***Book\*,bool*** con la cantidad de libros adecuada (la longitud del vector ***books***), usando punteros a la dirección de memoria de cada libro de ***books*** y ***setteando*** todos a ***true*** (es decir, que en primera instancia, todos se encuentran disponibles).

void Library::setup\_availability()

{

    for (size\_t i = 0; i < books.size(); ++i)

    {

        Book\* b = nullptr;

        b = &books[i];

        this->availability[b] = true;

    }

}

Siguiendo con la disponibilidad de los libros, se han creado métodos que actualizan la disponibilidad del libro dado (pasando el puntero que apunte a la dirección de memoria de dicho libro). ***set\_availability\_available()*** y ***set\_availability\_notavailable().*** Recorren el diccionario hasta encontrar el libro que coincida con el dado, y si lo encuentra, actualizan su disponibilidad.

void Library::set\_availability\_available(Book\* book)

{

    for (map<Book\*, bool>::const\_iterator it = this->availability.begin(); it != this->availability.cend(); it++)

    {

        if(book->to\_string() == it->first->to\_string())

        {

            this->availability[it->first] = true;

            break;

        }

    }

}

void Library::set\_availability\_notavailable(Book\* book)

{

    for (map<Book\*, bool>::const\_iterator it = this->availability.begin(); it != this->availability.cend(); it++)

    {

        if (book->to\_string() == it->first->to\_string())

        {

            availability[it->first] = false;

            break;

        }

    }

}

Además, por si se quisiera comprobar la disponibilidad de un libro en concreto (pasando un puntero a su dirección de memoria como parámetro) se ha creado un método que devuelve si dicho libro está o no disponible llamado ***check\_availability().***

bool Library::check\_availability(Book\* book)

{

    bool available = true;

    for (map<Book\*, bool>::const\_iterator it = this->availability.begin(); it != this->availability.cend(); it++)

    {

        if (book->to\_string() == it->first->to\_string())

        {

            available = availability[it->first];

            break;

        }

    }

    return available;

}

También, si se quisiera un vector con todos los libros disponibles, o uno con todos los no disponibles, se han creado los métodos ***get\_available\_books()*** y ***get\_notavailable\_books()*** que devuelven dicho vector respectivamente.

std::vector <Book> Library::get\_available\_books()

{

    std::vector <Book> returner;

    for (map<Book\*, bool>::const\_iterator it = this->availability.begin(); it != this->availability.cend(); it++)

    {

        if (it->second)

        {

            Book\* b = it->first;

            returner.push\_back(\*b);

        }

    }

    return returner;

}

std::vector <Book> Library::get\_notavailable\_books()

{

    std::vector <Book> returner;

    for (map<Book\*, bool>::const\_iterator it = this->availability.begin(); it != this->availability.cend(); it++)

    {

        if (!it->second)

        {

            Book\* b = it->first;

            returner.push\_back(\*b);

        }

    }

    return returner;

}

Se han creado dos ***getters*** para devolver el vector de ***books*** y el vector de ***users***. ***get\_books()*** y ***get\_users()*** respectivamente.

std::vector <Book> Library::get\_books() { return this->books; }

std::vector <User> Library::get\_users() { return this->users; }

Además, otros dos métodos que devuelven un puntero a un libro o a un usuario cuyo nombre haya sido pasado como referencia, ***get\_book()*** y ***get\_user().*** En caso de no haberlo, devuelve un puntero vacío ( un ***nullptr***).

Book\* Library::get\_book(std::string name)

{

    Book\* returned\_book = nullptr;

    for (size\_t i = 0; i < books.size(); ++i)

    {

        if (books[i].get\_name() == name)

            returned\_book = &books[i];

    }

    return returned\_book;

}

User\* Library::get\_user(std::string name)

{

    User\* returned\_user = nullptr;

    for (size\_t i = 0; i < users.size(); ++i)

    {

        if (users[i].get\_name() == name)

            returned\_user = &users[i];

    }

    return returned\_user;

}

Como se pide en el enunciado, para comprobar si un usuario dado tiene un libro dado, se ha creado el método ***check\_user\_has\_book(),*** que recibe dos punteros ***Book\**** y ***User\**** como parámetros. Comprueba que el libro dado sea igual al ***current\_book*** del usuario dado y en caso de que lo sea, devuelve un mensaje de que el nombre del usuario tiene el nombre del libro, y de que no lo tiene en caso de que no coincidan.

void Library::check\_user\_has\_book(User\* user, Book\* book)

{

    if (book->are\_equal(user->get\_current\_book()))

        cout << "user " << user->get\_name() << " has "<< book->get\_name();

    else

        cout << "user " << user->get\_name() << " don't have "<< book->get\_name();

}

Para ver el historial de libros de un usuario dado, se ha creado un método ***check\_user\_history()*** que recibe un puntero ***User\**** como parámetro. Este método simplemente llama a los métodos del usuario dado ***print\_info()*** y ***print\_history()*** explicados previamente en el apartado de ***User***.

void Library::check\_user\_history(User\* user)

{

    user->print\_info();

    user->print\_history();

}

Siguiendo con las funcionalidades de Library, cuenta con dos métodos clave ***take\_book()*** y ***leave\_book()*** encargados de gestionar el “movimiento” de los libros por parte de los usuarios.

***take\_book()*** recibe como parámetro dos punteros ***User\**** y ***Book\****, siendo el usuario que quiere sacar un libro de la biblioteca y el libro en cuestión que desea retirar. El método comprueba si dicho está disponible actualmente mediante ***check\_availability()*** y si lo está, usa ***set\_availability\_notavailable()*** para dicho libro. Después, comprueba que el usuario tenga un libro actualmente y si es así, lo añade a su historial mediante el método ***add\_book\_to\_history()*** del ***User*** y vuelve a marcar dicho libro como disponible con ***set\_availability\_available()***. Finalmente, asigna el libro pasado como parámetro al principio como ***current\_book*** del usuario dado.

void Library::take\_book(User\* user, Book\* book)

{

    if (check\_availability(book))

    {

        Library::set\_availability\_notavailable(book);

        Book current = user->get\_current\_book();

        if (current.get\_name() != "" && current.get\_genre() != "")

        {

            user->add\_book\_to\_history(&current);

            Library::set\_availability\_available(&current);

        }

        user->set\_current\_book(book);

    }

}

Por otro lado, ***leave\_book()*** recibe únicamente un puntero ***User\**** como parámetro. Comprueba que el ***current\_book*** de dicho usuario no esté disponible y si es así, llama a ***set\_availability\_available()*** para dicho libro. También, lo añade al historial de libros del usuario mediante su método ***add\_book\_to\_history(),*** y asigna un libro vacío como su ***current\_book***.

void Library::leave\_book(User\* user)

{

    Book current = user->get\_current\_book();

    if(!check\_availability(&current))

        Library::set\_availability\_available(&current);

    Book empty\_book;

    Book\* bp = &empty\_book;

    user->set\_current\_book(bp);

    user->add\_book\_to\_history(&current);

}

Para terminar, se han creado diferentes métodos que imprimen información relevante. ***print\_books()*** muestra todos los libros de la biblioteca, ***print\_users()*** muestra todos los usuarios, ***print\_available\_books()*** muestra todos los libros disponibles actualmente de la biblioteca, ***print\_notavailable\_books()*** muestra todos los libros no disponibles actualmente en la biblioteca y ***print\_availability()*** muestra todos los libros con su actual disponibilidad.

void Library::print\_books()

{    for (Book b : this->books)

        b.print(); }

void Library::print\_users()

{    for (User u : this->users)

        u.print\_info(); }

void Library::print\_available\_books()

{

    int counter = 1;

    for (map<Book\*, bool>::const\_iterator it = this->availability.begin(); it != this->availability.cend(); it++)

    {

        if (it->second == 1)

        {

            cout << counter << "." << it->first->to\_string() << endl;

            counter++;

        }

    }

}

void Library::print\_notavailable\_books()

{

    int counter = 1;

    for (map<Book\*, bool>::const\_iterator it = this->availability.begin(); it != this->availability.cend(); it++)

    {

        if (it->second != 1)

        {

            cout << counter << "." << it->first->to\_string() << endl;

            counter++;

        }

    }

}

void Library::print\_availability()

{

    int counter = 1;

    for (map<Book\*, bool>::const\_iterator it = this->availability.begin(); it != this->availability.cend(); it++)

    {

        if(it->second==1)

        cout << counter << "." << it->first->to\_string() << "-> " << " AVAILABLE" << endl;

        else

            cout << counter << "." << it->first->to\_string() << "-> " << " NOT AVAILABLE" << endl;

        counter++;

    }

}

Finalmente, y para darle algo de brillo a la consola final, se ha creado un método ***title()*** que devuelve un ***string*** con el diseño de la portada de la app.

std::string Library::title()

{

    std::stringstream ss;

    ss << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << "\*\*\*\*\*\*\*\*\*\*\*\*\*" << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl

        << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << "\*\*\*\*\*\*\*\*\*\*\*\*\*" << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl

        << "\*\*\*                                                 \*\*\*" << endl

        << "\*\*\*                  " << "~LIBRARY APP~" << "                  \*\*\*" << endl

        << "\*\*\*                                                 \*\*\*" << endl

        << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << "\*\*\*\*\*\*\*\*\*\*\*\*\*" << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl

        << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << "\*\*\*\*\*\*\*\*\*\*\*\*\*" << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl << endl;

    std::string title = string(ss.str());

    return title;

}

## 2.5 LibraryApp

Por último se encuentra el script ***LibraryApp.cpp***, que actúa de ***main***. En él (y como se puede ver en el código) se muestran todos los métodos disponibles así como una pequeña explicación de que va haciendo en cada momento. Este script es temporal, puesto que sirve como demostración de funcionalidades y no como visor de consola para una aplicación funcional. Principalmente, crea una biblioteca con 20 libros y 10 usuarios y va probando todos los métodos previamente explicados.