Programación Orientada a Objetos

Práctica 2: Relaciones de asociación, y contenedores de la STL
Implementación del caso de uso 1 «Gestión de usuarios y tarjetas» y
del caso de uso 2 «Gestión del carrito de la compra» del S. G. L.

Curso 2024–2025

Índice

1. Introducción

En esta práctica se va a llevar a cabo la implementación y prueba del caso de uso 1 «Gestión de usuarios y tarjetas» y del caso de uso 2 «Gestión del carrito de la compra» del Sistema de Gestión de Librería (SGL) descrito de manera general en la sección de prácticas del campus virtual de la asignatura.

Se partirá de los requisitos correspondientes a estos casos de uso y a partir de su diagrama de clases se realizará su implementación en el lenguaje de programación C++. Después se deberán realizar una serie de pruebas para asegurarse de que la aplicación, para los casos de uso 1 y 2, funciona correctamente y satisface los requisitos especificados.

2. Objetivos

Implementar en C++ el modelo de clases para los casos de uso 1 y 2 del SGL cumpliendo los principios de la programación orientada a objetos y todas las pruebas necesarias para que estemos seguros en un alto grado de que la implementación tiene el comportamiento esperado.

3. Descripción de los requisitos de los casos de uso

Comenzaremos la implementación del SGL por la gestión de los usuarios que van a realizar las compras y de las tarjetas de crédito con las que realizarán sus pagos. El SGL tendrá, por tanto, que permitir la creación de nuevas cuentas de usuario. De cada usuario se deben guardar nombre y apellidos, así como dirección postal. El usuario elegirá un identificador o nombre que debe ser único, y una contraseña de al menos 5 caracteres y que habrá que almacenar cifrada. También se deberá guardar información de una o más tarjetas de pago que el usuario podrá elegir para pagar los diferentes pedidos que haga. De las tarjetas se guardará el número, el nombre del titular, si está activa, y la fecha de expiración.

Cada cuenta de usuario va a disponer de un «carrito de la compra virtual», en el que puede ir añadiendo los artículos que desee comprar y quitando aquellos en los que ya no esté interesado. Puede adquirir varias unidades de un mismo artículo, especificando la cantidad al introducirlo en el carrito. No se comprobará el número de unidades almacenadas, pudiéndose añadir todos los ejemplares que se quiera. Los artículos tienen un número de referencia, título, fecha de publicación, precio de venta y cantidad disponible.

4. Diagrama de clases de los casos de uso 1 y 2

En la figura 1 aparece el diagrama de clases asociado con los casos de uso 1 y 2. Se trata de tres clases principales —*Usuario*, *Tarjeta* y *Articulo*— y otras auxiliares —*Clave* y *Numero*, *Cadena* y *Fecha*—, que se relacionan con las primeras por composición, es decir, las principales contienen varios atributos que son objetos de las clases auxiliares. Existe una relación de asociación calificada entre las clases *Usuario* y *Tarjeta* y una asociación entre *Usuario* y *Articulo* con un atributo de enlace, *cantidad*.

Nota importante: Aunque en circunstancias normales o «reales» se debería, no se podrá usar la clase estándar *std*::*string*, sino *Cadena* en su lugar, que por algo debe estar hecha antes de comenzar esta práctica.

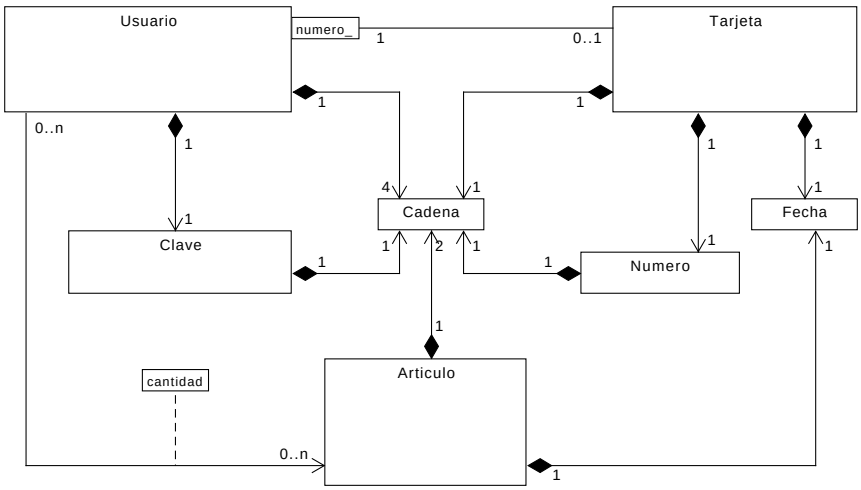

Figura 1: Diagrama de clases de los casos de uso 1 y 2

5. Implementación de las clases

La clase *Articulo* se escribirá en los ficheros `articulo.[ch]pp`. Las clases *Usuario* y *Clave* se escribirán en los ficheros `usuario.[ch]pp`. Las clases *Tarjeta* y *Numero* se escribirán en los ficheros `tarjeta.[ch]pp`.

5.1. La clase Articulo

Sesión 1

- Tiene cinco atributos, que corresponden al código de referencia, título, fecha de publicación, precio y número de ejemplares a la venta. Los tres primeros son inmodificables, una vez creado el objeto. No se establecen restricciones de formato sobre el código de referencia, se admitirá cualquier *Cadena*.
- Un artículo se construye únicamente con cinco parámetros dados en el orden: referencia, título, fecha de publicación, precio y existencias. Este último parámetro es opcional: si no se suministra, se toma como cero.
- Los atributos serán devueltos por métodos observadores de nombre *referencia*(), *titulo*(), *f_publi*(), *precio*() y *stock*(). Los dos últimos estarán sobrecargados para devolver una referencia al atributo correspondiente, con el fin de permitir su modificación.
- El operador de inserción en un flujo de salida, **operator** <<, imprimirá referencia, título, año de publicación y precio con el formato: «referencia entre corchetes, espacio, título entre comillas dobles, coma, espacio, año de publicación, punto, espacio, precio con 2 decimales, espacio, símbolo del euro», como se muestra en el ejemplo:
`[110] "Fundamentos de C++", 1998. 29,95 €`

5.2. La clase Clave

Sesión 1

- Su atributo es una *Cadena* que aloja la contraseña cifrada.
- Se construye a partir de una cadena de caracteres de bajo nivel que contendrá la contraseña «en claro», sin cifrar. La contraseña puede ser incorrecta por ser demasiado corta (menos de

5 caracteres, como se ha dicho), o por haber ocurrido algún error al cifrarse. En esos casos, el constructor elevará la excepción *Clave*::*Incorrecta*. Esta clase almacenará como atributo la razón de la incorrección, que será una enumeración llamada *Clave*::*Razon* con las constantes *CORTA* y *ERROR_CRYPT*. Su método observador *razon*() devolverá dicho atributo.

- El método observador *clave*() devolverá la contraseña cifrada.
- El método *verifica* () recibe como parámetro una cadena de bajo nivel con una supuesta contraseña *en claro* y devuelve el valor booleano verdadero si se corresponde con la contraseña almacenada, o falso si no. Vea la sección 5.6 para saber cómo cifrar y descifrar la contraseña.

5.3. La clase Numero

Sesión 1

- Esta clase representa el número troquelado en el anverso de una tarjeta de crédito. Se almacenará este dato como un atributo de tipo *Cadena*, ya que este «número» puede tener espacios de separación al principio, al final o, más normal, en medio.
- Por tanto su constructor recibirá como parámetro esa *Cadena* con el número. Tendrá que quitarle los blancos y comprobar que es un número válido. Si no lo fuera, lanzará la excepción *Numero*::*Incorrecto*. Vea la sección 5.6 para saber cómo averiguar si un número de tarjeta de crédito es válido.
  - Dentro de *Numero* se definirá la enumeración *Razon*, con los elementos *LONGITUD*, *DIGITOS* y *NO_VALIDO*, para representar por qué un *Numero* no es válido,
  - y la clase *Incorrecto*, con un atributo de tipo *Numero*::*Razon*, el constructor que recibe una *Razon* como parámetro, y el método observador *razon*() que devuelve el atributo.
- La clase *Numero* tendrá también un operador de conversión a cadena de bajo nivel de caracteres constantes, que será de utilidad para imprimir números de tarjeta.
- Deberá definirse el operador «menor-que» para comparar dos objetos de la clase.

5.4. La clase Usuario

Sesión 2

- Sus atributos son los siguientes:
  - Cuatro *Cadena* que representarán, por este orden: identificador, nombre, apellidos y dirección del usuario. Estos datos no son modificables, una vez creados.
  - La contraseña, de tipo *Clave* (V. § 5.2).
  - Las tarjetas de pago que posea: un diccionario ordenado de números de tarjeta y punteros a tarjetas: (*std*::*map*<*Numero*, *Tarjeta*∗>. Vea § 5.5). Se definirá un sinónimo público para este tipo, llamado *Tarjetas*.
  - El contenido del carrito, representado por un diccionario desordenado de punteros a artículos y el atributo de enlace de la asociación con la clase *Articulo*, que es la cantidad: *std*::*unordered_map*<*Articulo*∗, **unsigned int**> (V. § 5.6)). Igualmente, se definirá un sinónimo público para este tipo de nombre *Articulos*.
- Un *Usuario* se construirá únicamente con 5 parámetros, que serán por este orden: identificador, nombre, apellidos, dirección y clave.

  El constructor debe comprobar que el usuario que se va a construir sea correcto; es decir, que el identificador de usuario no esté repetido (V. § 5.6).

  En caso de que el identificador ya exista, el constructor elevará una excepción del tipo *Usuario*::*Id_duplicado*, cuyo constructor recibirá una *Cadena* representando ese identificador duplicado; su método observador *idd*() devolverá dicho identificador, guardado como atributo en esta clase de excepción.

- Un *Usuario* no podrá crearse por copia de otro, no podrá pues copiarse ni asignarse a otro; esto es lógico, pues entonces se crearía un *Usuario* con el mismo identificador, lo cual está prohibido, según se ha dicho ya en el punto anterior.
- Un *Usuario* poseerá métodos observadores que devolverán algunos de los atributos. Estos métodos se llamarán *id*(), *nombre*(), *apellidos*(), *direccion*(), *tarjetas* () y *compra*(). Ningún método devuelve la clave.
- De la asociación con la clase *Tarjeta* se encargarán los métodos de nombre *es_titular_de*() y *no_es_titular_de*(), que recibirán como parámetro una *Tarjeta* con la que el *Usuario* se asociará o de la que se desligará.
- El destructor tendrá que desligar sus tarjetas con el método *Tarjeta* :: *anula_titular*() sobre cada una de ellas. Solamente la clase *Usuario* podrá llamar a este método de *Tarjeta*. Vea § 5.5.
- La asociación unidireccional con *Articulo* se establecerá con un método llamado *compra*() (método sobrecargado) que recibirá dos parámetros, artículo y cantidad (por omisión es 1). Si la cantidad es 0, el artículo se sacará del carrito; es decir, el enlace con el artículo será eliminado; si es mayor que 0, esta será la nueva cantidad de dicho artículo en el carrito. La cantidad no está limitada, ya que no se requiere comprobar el *stock* del artículo.
- Con el método *compra*() se podrá vaciar el carrito sacando los artículos uno por uno, sin embargo, se dispondrá también de un método llamado *vaciar_carro*().
- El número de artículos diferentes que hay en el carrito será devuelto por un método llamado *n_articulos*().
- Se sobrecargará el operador de inserción en flujo (<<) para mostrar o imprimir un *Usuario* en un flujo de salida. El formato será:

  identificador [*clave cifrada*] *nombre apellidos*
  dirección
  `Tarjetas:`
  <lista de tarjetas>

  Ejemplo:

```
miguel [2456DJyasw0iY] Miguel Mares Miramontes
C/ del Paseo, 59 (El Ronquillo)
Tarjetas:


 -------------------------
/                         \
| Mastercard              |
| 5555555555554444        |
| MIGUEL MARES MIRAMONTES |
| Caduca: 01/27           |
\_________________________/


 -------------------------
/                         \
| Mastercard              |
| 5610591081018250        |
| MIGUEL MARES MIRAMONTES |
| Caduca: 09/26           |
\_________________________/
```

- Por último, se definirá una función externa *mostrar_carro*(), que deberá mostrar o imprimir en un flujo de salida dado como primer parámetro el contenido del carro de la compra de un usuario, pasado este como segundo parámetro, con el formato de este ejemplo:

```
Carrito de compra de sabacio [Artículos: 2]
 Cant. Artículo
================================================================
   1   [111] "The Standard Template Library", 2002. 42,10 €
   3   [110] "Fundamentos de C++", 1998. 35,95 €
```

En el ejemplo, *sabacio* es el identificador del usuario, no su nombre (observe que la inicial está en minúscula), y se imprime la cantidad de cada artículo del carro, seguido del artículo correspondiente.

5.5. La clase Tarjeta

Sesión 2

- Se definirá una enumeración pública de nombre *Tipo* con los valores *Otro*, *VISA*, *Mastercard*, *Maestro*, *JCB* y *AmericanExpress*.
- Sus atributos son:
  - un *Numero* constante, que es el número de la tarjeta tal como viene troquelado;
  - un puntero a *Usuario* constante, que es su titular;
  - una *Fecha* constante, que es la de caducidad,
  - y un booleano, que representa si la *Tarjeta* está activa o no.
- Se construye únicamente a partir del *Numero*, el *Usuario* y la *Fecha* de caducidad. Si esta fecha es anterior a la actual, se lanzará la excepción *Tarjeta*::*Caducada*. Esta clase de excepción tendrá un atributo que almacenará la fecha caducada, un constructor que la reciba como parámetro y un método observador *cuando*(), que la devolverá.

  Una *Tarjeta* se crea siempre activada.

  El constructor deberá asociar la tarjeta que se está creando con su *Usuario* correspondiente, llamando a *Usuario*::*es_titular_de*() sobre su titular.

  Al igual que en el caso de *Usuario*, no puede haber 2 tarjetas con el mismo número, por lo que habrá de seguirse la misma estrategia, como se describe en 5.6. En el caso de que el constructor reciba como parámetro un *Numero* ya existente, se lanzará la excepción *Tarjeta*::*Num_duplicado*, cuyo constructor recibe el *Numero* en cuestión. Esta clase de excepción tiene un atributo que es el *Numero*, y un método observador *que*() que lo devuelve.
- Dos tarjetas no se pueden copiar, ni al crearse ni por asignación. Esto sería ilegal, no puede haber dos tarjetas iguales.
- Habrá un método observador para cada atributo, que lo devuelva. Se llamarán *numero*(), *titular*(), *caducidad*() y *activa*().
- El método observador *tipo*() devolverá el *Tipo* de la *Tarjeta*. Este está determinado por los primeros dígitos del *Numero*. Para simplificar, aunque no sea muy correcto, supondremos que el tipo es:

  AmericanExpress Si el *Numero* empieza por 34 o 37,

  JCB Si el *Numero* empieza por 3, salvo 34 o 37,

  VISA Si el *Numero* empieza por 4,

  Mastercard Si el *Numero* empieza por 5,

  Maestro Si el *Numero* empieza por 6, y

  Otro En cualquier otro caso.
- Habrá también un método modificador para activar o desactivar una *Tarjeta*. Se llamará *activa*() (siendo por tanto una sobrecarga del método observador anterior), recibirá el nuevo estado de la *Tarjeta* en un parámetro booleano y devolverá dicho estado después de la activación o desactivación.

- Cuando un *Usuario* se destruya, sus tarjetas asociadas seguirán «vivas», aunque obviamente ya no pertenezcan a nadie. Por lo tanto, la clase *Tarjeta* tendrá un método modificador llamado *anula_titular*() que dará al puntero que representa al titular el valor nulo y desactivará la *Tarjeta* poniendo a **false** el atributo correspondiente. El destructor de la clase *Usuario* llamará a este método para cada una de sus tarjetas.

- Al destruirse un objeto de tipo *Tarjeta* deberá desasociarse de su *Usuario*, llamando al método *Usuario*::*no_es_titular_de*() sobre su titular, en caso de que este no haya sido destruido previamente; de lo contrario, la *Tarjeta* ya habrá sido desligada de su *Usuario* al ser destruido este (vea el punto anterior).

- Se sobrecargará el operador de inserción en flujo (**operator** <<) para mostrar o imprimir una *Tarjeta* en un flujo de salida. El formato será:

  tipo
  número
  titular facial
  `Caduca:` *MM/AA*

  donde *MM* es el mes de la fecha de caducidad, expresado con dos dígitos y *AA* son los dos últimos dígitos del año; por ejemplo: 11/28 sería noviembre de 2028.

  El *titular facial* es el nombre y apellidos del propietario de la tarjeta, en mayúsculas.

  Si quiere, por estética, puede dibujar líneas rodeando la información impresa de la tarjeta, simulando, aun pobremente, su aspecto. Esto es opcional.

  Para imprimir el nombre del tipo de la tarjeta (VISA, American Express...), deberá sobrecargar también el operador de inserción para *Tarjeta*::*Tipo*, el cual imprimirá el texto «Tipo indeterminado» cuando el valor sea *Tipo*::*Otro*.

  Ejemplo de impresión de una *Tarjeta* (las líneas son opcionales):

```
 --------------------
/                    \
| American Express   |
| 378282246310005    |
| SISEBUTO RUSCALLEDA|
| Caduca: 11/28      |
\--------------------/
```

- Dos *Tarjeta* podrán ordenarse por sus números. Para ello tendrá que definir el operador *menor-que* de dos tarjetas.

5.6. Pistas e indicaciones

Cifrado de la contraseña en la clase *Clave* Lo más simple, aparte de otros métodos triviales, pero aún manteniendo cierta dificultad para el descifrado «ilícito», puede ser el viejo algoritmo DES, implementado en la función de C (estándar POSIX.1) *crypt*(), declarada en <*unistd.h*>. Vea en el Manual de UNIX *crypt(3)*; p. ej., en consola, con la orden `man crypt`, o en https://man7.org/linux/man-pages/man3/crypt.3.html.

El prototipo de la función es: **char*** *crypt*(**const char*** *contraseña*, **const char*** *sal*)

El primer parámetro es la contraseña a encriptar. La *sal*, segundo parámetro, debe ser una cadena de dos caracteres por lo menos, sin contar el terminador. No importa si esta cadena es más larga, pues crypt() solo necesita los dos primeros caracteres e ignora el resto. Estos tienen que ser dos cualesquiera elegidos entre los 64 del conjunto {*a*−*zA*−*Z0*−9*./*} (26 letras minúsculas, 26 mayúsculas, 10 dígitos y 2 caracteres más, `'.'` y `'/'`). La función *crypt*() los emplea para variar el algoritmo en uno de $4\,096$ modos diferentes.

El resultado es un puntero a la contraseña cifrada, una serie de 13 caracteres imprimibles (los dos primeros corresponden a la *sal* empleada en la encriptación). El valor devuelto apunta a datos estáticos cuyo contenido se sobrescribe en cada llamada a *crypt*(), por lo que, para no perderlos, deberán ser copiados en el atributo de tipo *Cadena* de la clase *Clave* destinado a guardar la contraseña cifrada. En caso de error, *crypt*() devuelve un puntero nulo.

En la clase *Clave* añadiremos un atributo de clase privado en el que se almacenará la cadena formada por el conjunto de caracteres válidos para la *sal*:

```
class Clave {
private:
  // Conjunto de caracteres entre los que escoger la «sal» para encriptar las claves.
  static const char caracteres_validos[];
// ...
};
```

Este atributo añadido se deberá inicializar en el lugar adecuado con la cadena literal

```
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789./"
```

Para obtener una *sal* aleatoria en cada encriptación, es necesario hacer uso de otras funciones de C como (*s*)*rand*() (de la cabecera <*cstdlib*>, vea *rand(3)* como antes) o, mejor aún, usar generadores de números pseudoaleatorios (GNA) de la cabecera <*random*> de C++11, preferentemente *random_device* y *uniform_int_distribution* (vea: https://en.cppreference.com/w/cpp/numeric/random), como en el siguiente fragmento de código:

```
static std::random_device rd;  // Para obtener una semilla aleatoria para el GNA.
static std::mt19937 gna{rd()}; // Usamos un buen GNA, el de Matsumoto y Nishimura,
                               // por ejemplo, inicializado con el valor rd().
static const std::size_t n {sizeof(caracteres_validos) - 1};       // n = 64
std::uniform_int_distribution<std::size_t> uniforme(0, n - 1);    // Genera números
                                 // pseudoaleatorios equiprobables en el intervalo [0, 63].
const char sal[3] = {caracteres_validos[uniforme(gna)],
                     caracteres_validos[uniforme(gna)], '\0'};    // 2 aleatorios de
                                                  // caracteres_validos y el terminador.
```

Con esta variable *sal* generada aleatoriamente ya se puede usar la función *crypt*() para cifrar el parámetro que recibe el constructor de la clase *Clave*.

Verificación de contraseña Es tan fácil como cifrar la supuesta contraseña válida y comprobar si coincide con la que está guardada, la verdadera. Evidentemente, ahora se deberá usar la misma *sal* que cuando se encriptó la contraseña real. No es problema, porque, como se ha dicho anteriormente, los dos primeros caracteres de la clave ya cifrada se corresponden con la *sal* que se usó para cifrarla. Por lo tanto, simplemente se utilizará como segundo parámetro de *crypt*() la propia clave cifrada que está guardada.

Validez del número de la tarjeta Un número de tarjeta de pago válido debe contener solo dígitos, y esta cadena de solo dígitos debe tener una longitud comprendida entre 13 y 19, incluidos. Así que primero, en el constructor de *Numero*, hay que quitar los espacios en blanco y ver si hay algún carácter no dígito. Después hay que comprobar la longitud, y por último la validez, ya que el último dígito de la tarjeta es de control. Para esta tarea se debe usar el algoritmo de Luhn, que puede consultar por ejemplo en la *Wikipedia*: https://en.wikipedia.org/wiki/Luhn_algorithm

Se proporciona el fichero *luhn.cpp* con una implementación en C++.

Comprobación de usuario duplicado La duplicidad de usuarios la controlaremos guardando los identificadores en un conjunto desordenado (*unordered_set*) estático privado. Como un conjunto no admite elementos repetidos, al insertar en él un identificador que ya exista, el método *unordered_set*::*insert*() —que devuelve un *par* (clase *pair*) formado por un iterador apuntando al sitio de la inserción y un booleano— devolverá el par cuyo primer componente

(*first*) será un iterador al elemento con igual identificador y el segundo (*second*) será el valor **false**, indicando que la inserción no ha sido posible. Así que, tras un intento de inserción, se puede comprobar el segundo componente del par devuelto: si es **false**, es que el identificador ya existe y, por tanto, se trata de un duplicado. El siguiente fragmento de código refleja este comportamiento de la inserción:

```
std::unordered_set<Cadena> ids; // Conjunto de identificadores
typedef unordered_set<Cadena>::iterator tipoIt;
std::pair<tipoIt, bool> res = ids.insert(id); // Insertar el identificador id
if (res.second) // Inserción con éxito
  // id no estaba en ids, ahora es *res.first ==> id no está duplicado
else // Inserción fallida
  // id se ha encontrado en res.first ==> id es un identificador duplicado
```

Comprobación de tarjeta duplicada Para evitar la duplicidad de tarjetas, guardaremos los objetos *Numero* en un conjunto ordenado (*set*) estático privado. Al insertar en él un *Numero*, detectaremos un posible duplicado siguiendo el procedimiento descrito en el párrafo anterior para comprobar la duplicidad de un identificador de usuario.

Contenedores asociativos desordenados Los contenedores de nombre *unordered_...* requieren una función *hash* sobre su clave, así como que dos valores de la misma se puedan comparar con el operador de igualdad ==. Ambos, función y operador, se utilizan internamente para organizar el contenido y realizar búsquedas muy eficientemente. Por defecto, estos contenedores usan la función *std::hash<Key>()*, siendo *Key* el tipo de la clave, es decir, una especialización para el tipo *Key* de una plantilla de clase (o clase genérica) de funciones *hash* definida en la Biblioteca Estándar. En ella también existen especializaciones de esta plantilla para los tipos básicos, incluidos los punteros, y algunos tipos estándar como *std::string*.

Como el carro de la compra se representa mediante un diccionario desordenado de tipo *std::unordered_map<Articulo*, **unsigned int**>*, donde la clave es de tipo puntero (a *Articulo*), este contenedor hará uso del operador de igualdad de C++ y de la función *hash* estándar para punteros (a *Articulo*), *std::hash<Articulo*>*.

Sin embargo, en el apartado anterior sobre comprobación de usuario duplicado, se ha dicho que se cree un conjunto desordenado de identificadores y estos son de tipo *Cadena*, por lo que se usará el operador == de *Cadena* (implementado en la primera práctica) y una función *hash* que debemos definir para este tipo nuestro. La forma más fácil de lograrlo es aprovechar la función *hash* estándar de *string* por medio de conversiones. Esto es, añadiendo al fichero `../P1/cadena.hpp` el siguiente código:

```
// Para P2 y ss.
// Especialización de la plantilla std::hash<Key> para definir la función hash
// a usar con contenedores asociativos desordenados con claves de tipo Cadena,
// unordered_[set|map|multiset|multimap].
namespace std {                    // Estaremos dentro del espacio de nombres std.
  template <>                      // Es una especialización de una plantilla para Cadena.
  struct hash<Cadena> {            // Es una clase con solo un operador público.
    size_t operator() (const Cadena& cad) const          // El operador función.
    {
      hash<string> hs;                          // Creamos un objeto hash de string.
      auto p{(const char*)(cad)};     // Convertimos cad a cadena de bajo nivel.
      string s{p};                  // Creamos un string desde la cadena de b. nivel.
      size_t res{hs(s)};              // El hash del string. Como hs.operator()(s);
      return res;                                 // Devolvemos el hash del string.
      // En forma abreviada:
      // return hash<string>{}((const char*)(cad));
    }
  };
}
```

Debe incluirse también en dicho fichero la cabecera <*string*>, que es donde se define la especialización de la plantilla *hash*<*string*>. Excepcionalmente, este es el único sitio de la práctica donde se permite el uso de la palabra *string*.

6. Estructura, Makefile y pruebas

La estructura de directorios del proyecto del hipotético (más vale que lo sea) alumno de nombre Pánfilo Pancracio y de apellidos Povedilla Putiérrez debe ser la siguiente:

```
.../Povedilla_Putierrez_PanfiloPancracio/dsl-comprobaciones/...
   |                                    |Tests-auto/...
   |                                    |P1/cadena.[ch]pp
   |                                    |  |fecha.[ch]pp
   |                                    |  |Make_check.mk
   |                                    |  |Makefile
   |                                    |  | ...
   |                                    |P2/articulo.[hc]pp
   |                                    |  |tarjeta.[hc]pp
   |                                    |  |usuario.[hc]pp
   |                                    |  |libreria_check.cpp
   |                                    |  |Make_check.mk
   |                                    |  |Makefile
   |                                    |  |test-consola.cpp
   |                                    |luhn.cpp
```

De todos estos, se proporcionarán a través del campus virtual:

- El *Makefile* para la compilación.
- El fichero `luhn.cpp`, que contiene el código de la función *luhn*() para comprobar si un número de tarjeta es válido. Estará al mismo nivel de los directorios de prácticas Px, para que no haya que copiarlo en cada práctica, ya que siempre es el mismo.
- Los ficheros de pruebas de consola, `test-consola.cpp`, y de comprobaciones estáticas de código, `libreria_check.cpp` y *Make_check.mk*.
- Los directorios `Tests-auto` (pruebas unitarias automáticas) y `dsl-comprobaciones` (comprobación estática de código), de los que ya debería disponer de prácticas previas.

Al igual que en las prácticas precedentes, el *Makefile* incluye reglas para la construcción y ejecución de los cuatro tipos de pruebas. Concretamente:

- Los objetivos *test-consola* y *test-auto* permiten ejecutar (y compilar si fuera necesario) los programas de prueba de consola y de pruebas unitarias automáticas, respectivamente (también se puede hacer de una vez con *make tests*).
- El objetivo *check* realiza las comprobaciones sobre el código fuente, y el objetivo *valgrind* realiza el análisis de la gestión de la memoria.

Aparte de estas reglas, se dispone del objetivo *clean* para limpiar el directorio de ficheros sobrantes (módulos objeto, respaldos del editor, ejecutables, etc.), así como *distclean*, que limpia a fondo dejando nada más que los fuentes.

Nota: Siempre que pase de una práctica a otra (y por tanto del directorio Px al P$x+1$), es muy recomendable ejecutar la regla *distclean* para así limpiar el nuevo directorio, eliminando todos los objetos que hubieran sido generados en la construcción con la opción `-DP`x.

Para más indicaciones y objetivos, ejecute *make help* o lea los comentarios en el *Makefile*.