



PRÁCTICA 3

Análisis Estático Automatizado de Código

Autores:

Andrés Rojas Ortega 77382127F

Esteban Jódar Pozo 26003112W

Índice

1. Introducción	3
2. Selección de los estándares a seguir	4
3. Aplicación de estándares sobre el proyecto	4
3.1. DCL51-CPP. No declare ni defina un identificador reservado . . .	4
3.1.1. Enlace al sitio web del estándar seleccionado	4
3.1.2. Explicación sobre su utilidad	4
3.1.3. Aplicación del estándar al proyecto	6
3.2. DCL52-CPP. Nunca califique un tipo de referencia con constante o volátil	6
3.2.1. Enlace al sitio web del estándar seleccionado	6
3.2.2. Explicación sobre su utilidad	6
3.2.3. Aplicación del estándar al proyecto	7
3.3. DCL59-CPP. No defina un espacio de nombres sin nombre en un archivo de encabezado.	8
3.3.1. Enlace al sitio web del estándar seleccionado	8
3.3.2. Explicación sobre su utilidad	8
3.3.3. Aplicación del estándar al proyecto	8
3.4. MEM51-CPP. Desasignar correctamente la memoria asignada a los objetos dinámicamente.	9
3.4.1. Enlace al sitio web del estándar seleccionado	9
3.4.2. Explicación sobre su utilidad	9
3.4.3. Aplicación del estándar al proyecto	9
3.5. MEM52-CPP. Detectar errores de asignación de memoria	10
3.5.1. Enlace al sitio web del estándar seleccionado	10
3.5.2. Explicación sobre su utilidad	10
3.5.3. Aplicación del estándar al proyecto	11
3.6. FIO50-CPP. No ingrese y elabore alternativamente desde un flujo sin una llamada de posicionamiento.	11
3.6.1. Enlace al sitio web del estándar seleccionado	11
3.6.2. Explicación sobre su utilidad	11
3.6.3. Aplicación del estándar al proyecto	12
3.7. FIO51-CPP. Cerrar los archivos cuando ya no sean necesarios. . .	12
3.7.1. Enlace al sitio web del estándar seleccionado	12
3.7.2. Explicación sobre su utilidad	12
3.7.3. Aplicación del estándar al proyecto	13
3.8. ERR51-CPP. Manejar todas las excepciones.	14
3.8.1. Enlace al sitio web del estándar seleccionado	14

3.8.2.	Explicación sobre su utilidad	14
3.8.3.	Aplicación del estándar al proyecto	14
3.9.	OOP53-CPP. Escribir los inicializadores de miembros de los constructores en el orden canónico.	16
3.9.1.	Enlace al sitio web del estándar seleccionado	16
3.9.2.	Explicación sobre su utilidad	16
3.9.3.	Aplicación del estándar al proyecto	17
3.10.	MSC52-CPP. Las funciones que devuelven un valor deben devolver un valor desde todas las rutas de salida.	28
3.10.1.	Enlace al sitio web del estándar seleccionado	28
3.10.2.	Explicación sobre su utilidad	28
3.10.3.	Aplicación del estándar al proyecto	28
4.	Análisis estático automatizado de código	31
4.1.	Error nº1: Memory leak	31
4.1.1.	Origen/explicación del error detectado	31
4.1.2.	Corrección realizada para solventar el error	31
4.2.	Error nº2: Memory leak	32
4.2.1.	Origen/explicación del error detectado	32
4.2.2.	Corrección realizada para solventar el error	33
4.3.	Error nº3: Mismatching allocation and deallocation	34
4.3.1.	Origen/explicación del error detectado	34
4.3.2.	Corrección realizada para solventar el error	35
4.4.	Error nº4: Class has a constructor with 1 argument that is not explicit	35
4.4.1.	Origen/explicación del error detectado	35
4.4.2.	Corrección realizada para solventar el error	36
4.5.	Error nº5: Parameter can be declared with const	37
4.5.1.	Origen/explicación del error detectado	37
4.5.2.	Corrección realizada para solventar el error	37

1. Introducción

La aplicación seleccionada para el desarrollo de la primera práctica de calidad del software se trata de una aplicación destinada a la gestión de una biblioteca.

Esta aplicación se desarrollo para una de las prácticas de la asignatura Estructuras de datos, impartida en el segundo curso del grado de Ingeniería informática ofertado en la Universidad de Jaén. Los alumnos que realizaron esta práctica fueron Esteban Jódar Pozo (integrante de este grupo de prácticas) y Julian Yopis Ruiz.

Esta aplicación permite dar de alta a usuarios, los cuales pueden registrarse y hacer pedidos de libros tanto por temática, nombre o ISBN. Además, tiene un esquema de administrador, en el cual se podrá controlar aspectos relativos a pedidos, usuarios, libros que han solicitado los usuarios, etc.

Por tanto, tiene dos esquemas de entrada, de Administrador y de Usuario.

El administrador, una vez introducida su clave, tendrá acceso a:

- Crear pedidos para la biblioteca y tramitarlos.
- Cerrar dichos pedidos una vez finalizados.
- Ver los pedidos que tenga pendiente un usuario en concreto y tramitarlos.

Y un usuario, si no está registrado lo puede hacer, y si ya lo está:

- Puede introducir login y contraseña.
- Consultar un libro.
- Realizar un pedido.

Las estructuras de datos principales que sirven de soporte a la aplicación son listas simples enlazadas tanto de usuarios, como de libros, como de pedidos de usuario y pedidos de biblioteca.

2. Selección de los estándares a seguir

A continuación se presentan, a modo de lista no numerada, todos los estándares que hemos decidido aplicar a nuestro proyecto:

- **DCL51-CPP**. No declare ni defina un identificador reservado.
- **DCL52-CPP**. Nunca califique un tipo de referencia con constante o volátil.
- **DCL59-CPP**. No defina un espacio de nombres sin nombre de encabezado.
- **MEM51-CPP**. Desasignar correctamente la memoria asignada a los objetos dinámicamente.
- **MEM52-CPP**. Detectar errores de asignación de memoria.
- **FIO50-CPP**. No ingrese y elabore alternativamente desde un flujo sin una llamada de posicionamiento interviniente.
- **FIO51-CPP**. Cerrar los archivos cuando ya no sean necesarios.
- **ERR51-CPP**. Manejar todas las excepciones.
- **OOP53-CPP**. Escribir los inicializadores de miembros de los constructores en el orden canónico.
- **MSC52-CPP**. Las funciones que devuelven un valor deben devolver un valor desde todas las rutas de salida.

3. Aplicación de estándares sobre el proyecto

3.1. DCL51-CPP. No declare ni defina un identificador reservado

3.1.1. Enlace al sitio web del estándar seleccionado

<https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL51-CPP.+Do+not+declare+or+define+a+reserved+identifier>

3.1.2. Explicación sobre su utilidad

El estándar de C++ (ISO/IEC 14882-2014) que hace referencia a los nombres reservados especifica las siguientes reglas:

- *” Una unidad de traducción que incluye un encabezado de biblioteca estándar no debe `#define` o `#undef` nombres declarados en ningún encabezado de biblioteca estándar.*

- *Una unidad de traducción no debe `#define` o `#undef` nombres léxicamente idénticos a palabras clave, a los identificadores enumerados en la Tabla 3, o a los atributos-tokens descritos en 7.6.*
- *Cada nombre que contenga un guión bajo doble `_ _` o comience con un guión bajo seguido de una letra mayúscula está reservado a la implementación para cualquier uso.*
- *Cada nombre que comienza con un guión bajo se reserva para la implementación para su uso como nombre en el espacio de nombres global.*
- *Cada nombre declarado como un objeto con enlace externo en un encabezado está reservado a la implementación para designar ese objeto de biblioteca con enlace externo, tanto en el espacio de nombres estándar como en el espacio de nombres global.*
- *Cada firma de función global declarada con enlace externo en un encabezado está reservada a la implementación para designar esa firma de función con enlace externo.*
- *Cada nombre de la biblioteca C estándar declarado con enlace externo está reservado a la implementación para su uso como un nombre con enlace C externo, tanto en el espacio de nombres estándar como en el espacio de nombres global.*
- *Cada firma de función de la biblioteca C estándar declarada con enlace externo está reservada a la implementación para su uso como firma de función con enlace externo C y externo C++, o como nombre del ámbito del espacio de nombres en el espacio de nombres global.*
- *Para cada tipo T de la biblioteca C estándar, los tipos `::T` y `std::T` están reservados para la implementación y, cuando se defina, `::T` será idéntico a `std::T`.*
- *Los identificadores de sufijos literales que no comienzan con un guión bajo están reservados para una futura estandarización.”*

Los identificadores y nombres de atributos a los que se hace referencia en el extracto anterior son `override`, `final`, `alignas`, `carry_dependency`, `deprecated` y `noreturn`. Ningunos otros identificadores están reservados.

Declarar o definir identificadores en un contexto en el que estén reservados derivará en un comportamiento indefinido o impredecible. Para evitar esto, siempre hay que evitar reservar o definir identificadores que estén reservados.

3.1.3. Aplicación del estándar al proyecto

Nuestro proyecto cumple perfectamente con el estándar, ya que no hace uso de ningún identificador reservado. Uno de los ejemplos de las reglas definidas en el apartado anterior es la nomenclatura de las cabeceras de los archivos, a continuación una captura de pantalla de una de ellas:

```
1  /**
2   * @file PedidoBiblioteca.h
3   * @brief Archivo cabecera donde se almacena toda la información relacionada con la clase PedidoBiblioteca.
4   */
5
6  #ifndef PEDIDOBIBLIOTECA_H
7  #define PEDIDOBIBLIOTECA_H
```

Figura 1: Captura de pantalla del cumplimiento del estándar DCL51-CPP.

Otro ejemplo de cumplimiento del estándar es el uso del identificador T en la implementación de las plantillas del programa. A continuación, una captura de pantalla de dicho uso:

```
10 #include <iostream>
11 using namespace std;
12
13 template<class T>
14
15 /**
16 * @brief Plantilla genérica para un nodo de las estructuras de datos que soportara esta aplicación.
17 */
18 struct nodo {
19     T data; ///< Dato parametrizado. Puede ser un usuario, un libro, una fecha, un ISBN...etc.
20     nodo<> * sig; ///< Nodo siguiente en la estructura parametrizado a cualquier tipo de los anteriores.
21 };
22
```

Figura 2: Captura de pantalla del cumplimiento del estándar DCL51-CPP.

Estos son los dos ejemplos más claros de cumplimiento del estándar. No podemos mostrar más capturas de pantalla ya que, al no hacer uso de identificadores reservados, no podemos mostrar las correcciones de los incumplimientos.

3.2. DCL52-CPP. Nunca califique un tipo de referencia con constante o volátil

3.2.1. Enlace al sitio web del estándar seleccionado

<https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL52-CPP.+Never+qualify+a+reference+type+with+const+or+volatile>

3.2.2. Explicación sobre su utilidad

Como sabemos de cursos anteriores, C++ no permite modificar el valor de una variable que haya sido calificada con constante (const).

En el caso de objetos que sea referenciados, es decir, utilizando el caracter reservado '&' se puede cometer el error de escribir la expresión de la siguiente forma: "*char const& p*", C++ ignora o prohíbe la asignación de referencia a la palabra reservada *const*". Este hecho puede desencadenar en escrituras accidentales en la variable constante, probocando resultados no esperados en la ejecución del programa.

Para que la expresión fuera correcta, debería escribirse de una de las siguientes formas: "*const char &p*" ó "*char const &p*"

3.2.3. Aplicación del estándar al proyecto

En nuestro proyecto hacemos uso repetidas veces de expresiones con referencias de este tipo, y en todas ellas respetamos la sintaxis que se describe en el presente estándar.

A continuación, presentamos capturas de pantalla de alguna de las expresiones que se encuentran en el código fuente a modo de ejemplo:

```
68 /**
69  * @brief Comparar fechas.
70  * @param [in] f Fecha(dir, const).
71  * @return bool. True en el caso de que la fecha actual sea menor que la fecha parámetro, false en cualquier otro caso.
72  */
73 bool Fecha::operator<(const Fecha &f) {
74     if (anio < f.anio)
75         return true;
76     else if (anio > f.anio)
77         return false;
78
79     if (mes < f.mes)
80         return true;
81     else if (mes > f.mes)
82         return false;
83
84     if (dia < f.dia)
85         return true;
86     else if (dia > f.dia)
87         return false;
88
89     if (hora < f.hora)
90         return true;
91     else if (hora > f.hora)
92         return false;
93
94     if (min < f.min)
95         return true;
96     else
97         return false;
98 }
```

Figura 3: Captura de pantalla del cumplimiento del estándar DCL52-CPP.

```
221 /**
222  * @brief Función auxiliar de conversión desde estructura de tiempo tm de time.h.
223  * @param [out] t tm (const, dir).
224  */
225 void Fecha::LeerTiempo(const tm &t) {
226     dia = t.tm_mday;
227     mes = t.tm_mon + 1;
228     anio = t.tm_year + 1900;
229     hora = t.tm_hour;
230     min = t.tm_min;
231 }
232
```

Figura 4: Captura de pantalla del cumplimiento del estándar DCL52-CPP.

3.3. DCL59-CPP. No defina un espacio de nombres sin nombre en un archivo de encabezado.

3.3.1. Enlace al sitio web del estándar seleccionado

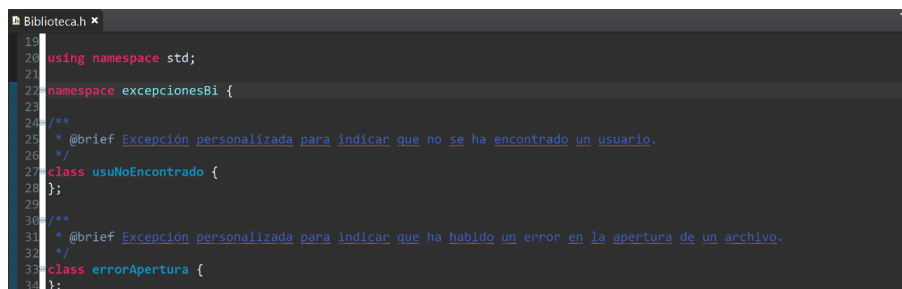
<https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL59-CPP.+Do+not+define+an+unnamed+namespace+in+a+header+file>

3.3.2. Explicación sobre su utilidad

El código C++ de calidad de producción utiliza con frecuencia archivos de encabezado como medio para compartir código entre unidades de traducción. Un archivo de encabezado es cualquier archivo que se inserta en una unidad de traducción a través de una directiva. No defina un espacio de nombres sin nombre en un archivo de encabezado. Cuando se define un espacio de nombres sin nombre en un archivo de encabezado, puede dar lugar a resultados sorprendentes. Debido al vínculo interno predeterminado, cada unidad de traducción definirá su propia instancia única de los miembros del espacio de nombres sin nombre que se usan en ODR dentro de esa unidad de traducción. Esto puede causar resultados inesperados, hinchar el ejecutable resultante o desencadenar inadvertidamente el comportamiento indefinido debido a infracciones de una regla de una definición.

3.3.3. Aplicación del estándar al proyecto

Los espacios de nombres sin nombre se usan para definir un espacio de nombres que es único para la unidad de traducción, donde los nombres contenidos tienen vinculación interna de forma predeterminada.



```

18
19
20 using namespace std;
21
22 namespace excepcionesBi {
23
24 /**
25  * @brief Excepción personalizada para indicar que no se ha encontrado un usuario.
26  */
27 class usuNoEncontrado {
28 };
29
30 /**
31  * @brief Excepción personalizada para indicar que ha habido un error en la apertura de un archivo.
32  */
33 class errorApertura {
34 };

```

Figura 5: Captura de pantalla del cumplimiento del estándar DCL59-CPP.

Aquí observamos como en un archivo cabecera al incluir un nuevo espacio de nombres, se le ha tenido que poner un nombre distinto para no confundir a la unidad de traducción. En este caso el nombre ofrecido ha sido el de “excepcionesBi”.



Figura 6: Captura de pantalla del cumplimiento del estándar DCL59-CPP.

Así puede ser perfectamente referenciado desde el .cpp de dicho archivo Biblioteca como podemos ver en la imagen:

3.4. MEM51-CPP. Desasignar correctamente la memoria asignada a los objetos dinámicamente.

3.4.1. Enlace al sitio web del estándar seleccionado

<https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM51-CPP.+Properly+deallocate+dynamically+allocated+resources>

3.4.2. Explicación sobre su utilidad

Desasignar un puntero que no ha sido asignado con `new ()` es un comportamiento indefinido porque el valor del puntero no se obtuvo mediante una función de asignación.

Al mismo tiempo, desasignar un puntero que ya se ha pasado a una función de desasignación es un comportamiento indefinido porque el valor del puntero ya no apunta a la memoria que se ha asignado dinámicamente. Puede estar apuntando a cualquier parte.

Cuando `new ()` es invocado, el resultado es una llamada a un operador sobrecargable con el mismo nombre. Esta función se puede llamar directamente, pero tiene las mismas restricciones que su contraria dijéramos.

Es decir, `delete ()` y pasarle un parámetro de puntero tiene las mismas restricciones que llamar al operador `delete ()` en ese puntero. Además, las sobrecargas están sujetas a la resolución del alcance, por lo que es posible (pero no permitido) llamar a un operador específico de clase para asignar un objeto, pero a un operador global para desasignar el objeto. En resumen, y como siempre se nos ha enseñado, tras un `new ()` debe de haber un `delete ()`, con lo cual, aparte de evitar comportamientos inesperados, haremos una buena gestión de la memoria, recurso finito que hay que tratar con eficiencia.

3.4.3. Aplicación del estándar al proyecto

Aquí vemos como se llama al operador `new ()`:

```
/**
 * @brief Constructor por defecto.
 */
Aplication::Aplication() {
    bi = Biblioteca();
    usu = new Usuario();
    lusu = new lista_sin<Usuario>();
    li = Libro();
    pedbi = new PedidoBiblioteca();
    pedbi = new lista_sin<PedidoBiblioteca *>();
    pedusu = new lista_sin<PedidoUsuario *>();
    libro = new lista_sin<Libro *>();
    pedbipunt = new PedidoBiblioteca();
}
```

Figura 7: Captura de pantalla del cumplimiento del estándar MEM51-CPP.

Para desasignar correctamente la memoria asignada dinámicamente a estos objetos observamos como en el destructor se hacen las llamadas correspondientes al operador delete () que es la contraparte de new () y además a la función de limpieza, que internamente hace también una llamada al operador delete ().

```
/**
 * @brief Destructor de la clase Aplication.
 */
virtual ~Aplication() {
    delete usu;
    pedbi->limpia();
    pedusu->limpia();
}
};
```

Figura 8: Captura de pantalla del cumplimiento del estándar DCL59-CPP.

3.5. MEM52-CPP. Detectar errores de asignación de memoria

3.5.1. Enlace al sitio web del estándar seleccionado

<https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM52-CPP.+Detect+and+handle+memory+allocation+errors>

3.5.2. Explicación sobre su utilidad

Como sabemos de cursos anteriores, el operador de asignación de memoria dinámica new (), produce una excepción si se produce un error en la asignación. Muchas veces nos hemos preguntado si el resultado de la llamada era nullptr o puntero nulo haciendo comprobaciones.

Pues bien, no es necesario hacer dichas comprobaciones debido a la excepción que se produce.

Además, también nos evita que el valor devuelto no lo sea antes de tener acceso al puntero o de tener a este apuntando a lo que conocemos como “basura”.

Esta excepción es la (std::bad_alloc), que detecta si no se puede asignar suficiente memoria.

3.5.3. Aplicación del estándar al proyecto

Por ejemplo, si no se ha podido asignar memoria para `usu` con el operador `new` (`()`), aparte de las excepciones personalizadas que nos mostrarían un mensaje, es correcto que en el primer bloque `catch` se atrapara también esta excepción si acaso fallase la asignación.

```
57 try {
58     usu = bi.buscaUsuario(alogin, aclave);
59     pedusu = bi.buscaPedidosUsuarioPendientes(usu);
60     while (i < pedusu->tamano()) {
61         cout << *(pedusu->lee(i)) << endl;
62         i++;
63     }
64 } catch (const excepcionesBi::usuNoEncontrado& e) {
65     cerr << e.what();
66 }
```

Figura 9: Captura de pantalla del incumplimiento del estándar MEM52-CPP.

Una vez arreglado, se muestra ya en los `catch` la “`bad_alloc`” si falla la asignación de memoria:

```
57 try {
58     usu = bi.buscaUsuario(alogin, aclave);
59     pedusu = bi.buscaPedidosUsuarioPendientes(usu);
60     while (i < pedusu->tamano()) {
61         cout << *(pedusu->lee(i)) << endl;
62         i++;
63     }
64 } catch (bad_alloc&) {
65 } catch (const excepcionesBi::usuNoEncontrado& e) {
66     cerr << e.what();
67 } catch (excepcionesBi::pedidoUsuarioNoencontrado& e) {
68     cerr << e.what();
69 }
70 }
```

Figura 10: Captura de pantalla del cumplimiento del estándar MEM52-CPP.

3.6. FIO50-CPP. No ingrese y elabore alternativamente desde un flujo sin una llamada de posicionamiento.

3.6.1. Enlace al sitio web del estándar seleccionado

<https://wiki.sei.cmu.edu/confluence/display/cplusplus/FIO50-CPP>.
+Do+not+alternately+input+and+output+from+a+file+stream+without+an+intervening+positioning+call

3.6.2. Explicación sobre su utilidad

Cuando se abre un archivo, entre la secuencia de entrada al mismo y la de cierre, ha de situarse una función o llamada para un posicionamiento correcto dentro del mismo.

Si no nos situamos correctamente dentro de el podemos dar lugar a algunos comportamientos no deseados como:

- El puntero de lectura/escritura puede escribir datos al final de un archivo y a continuación leer del mismo. Si no hay una llamada para un posicionamiento intermedio, el comportamiento es indefinido.

- El puntero de lectura/escritura puede comenzar leer datos al comienzo de un archivo y a continuación escribir en el mismo.

De ahí que sea necesario una función intermedia, que posicione correctamente el puntero de lectura/escritura, situada entre la entrada al fichero y el cierre del mismo para evitar comportamientos indeseados. En este caso ha sido la función `std::basic_istream<T>::seekg()`.

3.6.3. Aplicación del estándar al proyecto

```

59  /**
60   * @brief Devuelve una lista con los libros que contengan el título que se le pasa como parametro.
61   * @param [in] fichero string. Fichero donde se encuentra almacenada la información de los libros.
62   */
63  void Biblioteca::cargalibros(string fichero) {
64      ifstream entrada;
65      entrada.open(fichero.c_str(), ios::in);
66      string aTitulo;
67      string aAutores;
68      string aEditorial;
69      string aISBN;
70      string aAnio;
71      string aPrecioActual;
72      string espacio;
73      if (entrada) {
74          entrada.seekg(0, ios::beg);
75          while (!entrada.eof()) {
76              getline(entrada, aTitulo);
77              getline(entrada, aAutores);
78              getline(entrada, aAnio); // Convierte a entero un string.
79              getline(entrada, aEditorial);
80              getline(entrada, aISBN);
81              getline(entrada, aPrecioActual); // Convierte a entero un string.
82              getline(entrada, espacio);
83              Libro *lib = new Libro(aTitulo, aAutores, aEditorial, aISBN,
84                                   atoi(aAnio.c_str()), (float) atof(aPrecioActual.c_str()));
85              libro.aumenta(lib);
86          }
87      } else {
88          throw excepcionesBi::errorApertura();
89      }
90      entrada.close();
91  }

```

Figura 11: Captura de pantalla del cumplimiento del estándar FIO50-CPP.

En esta solución compatible, la `std::basic_istream<T>::seekg()` función se llama en la línea 73, entre la salida y la entrada, posicionando el puntero al comienzo de la lectura, eliminando el comportamiento indefinido.

3.7. FIO51-CPP. Cerrar los archivos cuando ya no sean necesarios.

3.7.1. Enlace al sitio web del estándar seleccionado

<https://wiki.sei.cmu.edu/confluence/display/cplusplus/FIO51-CPP.+Close+files+when+they+are+no+longer+needed>

3.7.2. Explicación sobre su utilidad

Una llamada a la función `std::basic_filebuf<T>::open()` siempre debe ir acompañada de otra llamada a la función `std::basic_filebuf<T>::close()`

antes de la finalización del ciclo de vida del último puntero que almacenase el valor devuelto por la llamada de la primera función ó antes de la finalización del programa, lo que ocurriese antes.

La mala praxis de este estándar puede provocar la utilización innecesaria de memoria estática durante toda la ejecución del programa. En el peor de los casos si se abrieran muchos archivos y no se cerrara ninguno durante la ejecución del programa, podría llegar a provocar un desbordamiento de la memoria estática. Aún utilizando memoria dinámica, si no cerramos el archivo cuando ya no sea necesario, seguiríamos desperdiciando memoria igualmente.

3.7.3. Aplicación del estándar al proyecto

En el proyecto elegido para la realización de las prácticas solo se hace lectura de un fichero en una única función en todo el programa, esta función es "voidBiblioteca :: cargaLibros(string fichero)".

```

58 /**
59  * @brief Devuelve una lista con los libros que contengan el titulo que se le pasa como parametro.
60  * @param [in] fichero string. Fichero donde se encuentra almacenada la información de los libros.
61  */
62 void Biblioteca::cargaLibros(string fichero) {
63     ifstream entrada;
64     entrada.open(fichero.c_str(), ios::in);
65     string aTitulo;
66     string aAutores;
67     string aEditorial;
68     string aISBN;
69     string aAnio;
70     string aPrecioActual;
71     string espacio;
72     if (entrada) {
73         while (!entrada.eof()) {
74             getline(entrada, aTitulo);
75             getline(entrada, aAutores);
76             getline(entrada, aAnio); /// Convierte a entero un string.
77             getline(entrada, aEditorial);
78             getline(entrada, aISBN);
79             getline(entrada, aPrecioActual); /// Convierte a entero un string.
80             getline(entrada, espacio);
81             Libro *lib = new Libro(aTitulo, aAutores, aEditorial, aISBN,
82                                   atoi(aAnio.c_str()), (float) atof(aPrecioActual.c_str()));
83             libro.aumenta(lib);
84         }
85     } else {
86         throw excepcionesBI::errorApertura();
87     }
88     entrada.close();
89 }

```

Figura 12: Captura de pantalla del cumplimiento del estándar FIO51-CPP.

En la captura de pantalla anterior observamos que en la línea número 64 se abre el archivo "ficheroz", cuando ya se han realizado todas las operaciones de lectura del contenido del mismo, se procede a su cierre en la línea 88, quedando verificado el cumplimiento del estándar.

3.8. ERR51-CPP. Manejar todas las excepciones.

3.8.1. Enlace al sitio web del estándar seleccionado

<https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR51-CPP.+Handle+all+exceptions>

3.8.2. Explicación sobre su utilidad

Cuando se produce una excepción, el control se transfiere al controlador más cercano con un tipo que coincide con el tipo de la excepción producida. Si no se encuentra ningún controlador coincidente directamente dentro de los controladores para un bloque try en el que se produce la excepción, la búsqueda de un controlador coincidente continúa buscando dinámicamente controladores en los bloques de prueba circundantes del mismo subproceso.

Todas las excepciones producidas por una aplicación deben ser detectadas por un controlador de excepciones coincidente. Incluso si la excepción no se puede recuperar correctamente, el uso del controlador de excepciones coincidente garantiza que la pila se desenrollará correctamente y proporciona la oportunidad de administrar correctamente los recursos externos antes de finalizar el proceso.

3.8.3. Aplicación del estándar al proyecto

Aquí podemos ver como se lanzan excepciones, pero no son tratadas correctamente:



```

52     return usur.lee(1);
53 }
54 }
55 throw excepcionesBi::usuNoEncontrado();
56 }

```

Figura 13: Captura de pantalla del incumplimiento del estándar ERR51-CPP.



```

81     getline(entrada, espacio);
82     Libro *lib = new Libro(aTitulo, aAutores, aEditorial, aISBN,
83         atoi(aAnio.c_str()), (float) atof(aPrecioActual.c_str()));
84     lib.aumenta(lib);
85 }
86 } else {
87     throw excepcionesBi::errorApertura();
88 }

```

Figura 14: Captura de pantalla del incumplimiento del estándar ERR51-CPP.

Ya que las clases no derivan de la clase exception ni tienen nada definido para ellas:

```
*Biblioteca.h*
8 #include "Libro.h"
9 #include "PedidoBiblioteca.h"
10 #include "PedidoUsuario.h"
11 #include "Usuario.h"
12 #include "lista_sin.h"
13 #include "Fecha.h"
14 #include <fstream>
15 #include <cstdio>
16 #include <cstdlib>
17 #include <string>
18 #include <sstream>
19
20 using namespace std;
21
22 namespace excepcionesBi {
23
24 /**
25  * @brief Excepción personalizada para indicar que no se ha encontrado un usuario.
26  */
27 class usuNoEncontrado {
28 };
29
30 /**
31  * @brief Excepción personalizada para indicar que ha habido un error en la apertura de un archivo.
32  */
33 class errorApertura {
34 };
35 }
```

Figura 15: Captura de pantalla del incumplimiento del estándar ERR51-CPP.

Una vez le hemos dado definición a dichas excepciones y con la inclusión de la directiva `#include <exception>` en el fichero de cabecera, ya podemos controlar las excepciones, lo que garantiza que la pila se desenrolla hasta la función y permite una gestión elegante de los recursos externos.

```
#include <exception>
using namespace std;
namespace excepcionesBi {
/**
 * @brief Excepción personalizada para indicar que no se ha encontrado un usuario.
 */
class usuNoEncontrado : public exception {
public:
    const char* what() const throw () {
        return "\nError: El usuario no se pudo encontrar.";
    }
};
/**
 * @brief Excepción personalizada para indicar que ha habido un error en la apertura de un archivo.
 */
class errorApertura : public exception {
public:
    const char* what() const throw () {
        return "\nError: El Fichero no se pudo abrir.";
    }
};
/**
 * @brief Excepción personalizada para indicar que no se ha encontrado un libro.
 */
class libroNoencontrado : public exception {
public:
    const char* what() const throw () {
        return "\nError: El libro no se pudo encontrar.";
    }
};
}
```

Figura 16: Captura de pantalla del cumplimiento del estándar ERR51-CPP.

Se atrapan las excepciones lanzadas por los `throw` en los bloques `catch` pero no se tratan adecuadamente.


```

57     try {
58         usu = bi.buscaUsuario(alogin, aclave);
59         pedusu = bi.buscaPedidosUsuarioPendientes(usu);
60         while (i < pedusu->tamano()) {
61             cout << *(pedusu->lee(i)) << endl;
62             i++;
63         }
64     } catch (excepcionesBi::usuNoEncontrado&) {
65         cout << " Usuario no encontrado. " << endl;
66     } catch (excepcionesBi::pedidoUsuarioNoencontrado&) {
67         cout << " El usuario no tiene pedidos pendientes. " << endl;
68     }
69 }

```

Figura 17: Captura de pantalla del incumplimiento del estándar ERR51-CPP.

Y ya aquí se atrapan en el catch correspondiente y se tratan adecuadamente.

```

60     try {
61         usu = bi.buscaUsuario(alogin, aclave);
62         pedusu = bi.buscaPedidosUsuarioPendientes(usu);
63         while (i < pedusu->tamano()) {
64             cout << *(pedusu->lee(i)) << endl;
65             i++;
66         }
67     } catch (const excepcionesBi::usuNoEncontrado& e) {
68         cerr << e.what();
69     } catch (excepcionesBi::pedidoUsuarioNoencontrado& e) {
70         cerr << e.what();
71     }

```

Figura 18: Captura de pantalla del cumplimiento del estándar ERR51-CPP.

3.9. OOP53-CPP. Escribir los inicializadores de miembros de los constructores en el orden canónico.

3.9.1. Enlace al sitio web del estándar seleccionado

<https://wiki.sei.cmu.edu/confluence/display/cplusplus/OOP53-CPP.+Write+constructor+member+initializers+in+the+canonical+order>

3.9.2. Explicación sobre su utilidad

Este estándar nos dice que la lista de inicializadores de miembros para un constructor permite que los miembros se inicialicen a valores especificados y que los constructores de clases base sean llamados con argumentos específicos.

Es decir, en el caso de que un atributo miembro de la clase necesite el valor de otro atributo miembro, este debe de estar inicializado previamente. El atributo miembro dependiente irá después del miembro del que depende en la lista de atributos y en el constructor.

En el caso de que no se cumpliera este estándar y en el constructor el miembro dependiente se intentara inicializar antes de que se haya inicializado el miembro del que depende, daría lugar a un comportamiento indefinido, como por ejemplo leer memoria no inicializada (datos basura).

En definitiva, se deben de escribir siempre los inicializadores de miembros en un constructor en el orden canónico: primero las clases base directas en el orden en que aparecen en la lista de especificador-base para la clase, luego los miembros de datos no estáticos en el orden en que se declaran en la definición de la clase.

3.9.3. Aplicación del estándar al proyecto

En nuestro proyecto se encuentran las siguientes clases:

- Application
- Biblioteca
- Fecha
- Libro
- lista_sin
- PedidoBiblioteca
- PedidoUsuario
- Usuario

Para la clase Application, tenemos la siguiente lista de miembros:

```
13: /**
14:  * @brief Clase principal la cual derivará en sus variantes de admin y de usuario.
15:  */
16: class Application {
17:     Biblioteca bi;           ///< Objeto de la clase Biblioteca.
18:     Usuario *usu;           ///< Referencia a un usuario en concreto.
19:     lista_sin<Usuario> lusu;  ///< Lista de los usuarios registrados en la biblioteca.
20:     Libro li;               ///< Objeto de la clase libro.
21:     PedidoBiblioteca *pedbi; ///< Referencia a un pedido específico hecho por la biblioteca
22:     lista_sin<PedidoBiblioteca> *pedbi; ///< Lista de todos los libros pedidos en cada solicitud.
23:     lista_sin<PedidoUsuario> *pedusu;  ///< Lista de los pedidos hechos por los usuarios.
24:     lista_sin<Libro> *li;             ///< Lista de todos los libros pedidos por todos los usuarios.
25:     PedidoBiblioteca *pedbiunt;      ///< Lista de los pedidos de la biblioteca no tramitados.
26: }
```

Figura 19: Captura de pantalla de los atributos miembros de la clase Application.

En la siguiente captura de pantalla se puede apreciar cómo no se cumple claramente con el estándar, ya que la inicialización de los atributos miembros de la clase no siguen el orden establecido en la declaración de la clase y hay atributos que no se inicializan.

```

9  /**
10 * @brief Constructor por defecto.
11 */
12 Application::Application() {
13     pedusu = new lista_sin<PedidoUsuario *>;
14     pedbi = new lista_sin<PedidoBiblioteca *>;
15     libro = new lista_sin<Libro *>;
16     usu = new Usuario;
17     pedbipunt = new PedidoBiblioteca;
18     pedbipunt = NULL;
19     pedBi = new PedidoBiblioteca;
20 }
21
22

```

Figura 20: Captura de pantalla del incumplimiento del estándar OOP53-CPP.

A continuación, se ha incluido la inicialización de los atributos que no aparecían y se han inicializado todos los atributos en el orden con el que se declaran en la clase:

```

8
9  /**
10 * @brief Constructor por defecto.
11 */
12 Application::Application() {
13     bi = Biblioteca();
14     usu = new Usuario;
15     lusu = new lista_sin<Usuario*>();
16     li = Libro();
17     pedBi = new PedidoBiblioteca;
18     pedbi = new lista_sin<PedidoBiblioteca *>;
19     pedusu = new lista_sin<PedidoUsuario *>;
20     libro = new lista_sin<Libro *>;
21     pedbipunt = new PedidoBiblioteca;
22 }
23

```

Figura 21: Captura de pantalla del cumplimiento del estándar OOP53-CPP.

Para la clase Biblioteca, tenemos la siguiente lista de miembros:

```

55 /**
56 * @brief Clase que representa la información y el funcionamiento de una biblioteca.
57 */
58 class Biblioteca {
59
60     lista_sin<Usuario *> usu; ///< Lista donde se almacenan todos los usuarios.
61     lista_sin<PedidoUsuario *> pedusu; ///< Lista donde se almacenan todos los pedidos de los usuarios.
62     lista_sin<PedidoBiblioteca *> pedbi; ///< Lista donde se almacenan todos los pedidos hechos por la biblioteca.
63     lista_sin<Libro *> libro; ///< Lista donde se almacenan todos los libros que posee la biblioteca.
64     Usuario *usu; ///< Puntero al último usuario introducido en la biblioteca.
65

```

Figura 22: Captura de pantalla de los atributos miembros de la clase Biblioteca.

En la siguiente captura de pantalla se puede apreciar cómo sí se cumple claramente con el estándar, ya que la inicialización de los atributos miembros de la clase siguen el orden establecido en la declaración de la clase y se inicializan todos los atributos.

```

67
68 /**
69  * @brief Constructor por defecto de la clase Biblioteca.
70  */
71 Biblioteca() :
72     usur(), pedido_usu(), pedidoBl(), libro() {
73     usu = new Usuario;
74 }
75

```

Figura 23: Captura de pantalla del cumplimiento del estándar OOP53-CPP.

Para la clase Fecha, tenemos la siguiente lista de miembros:

```

11
12 /**
13  * @brief Clase sencilla para representar fechas y horas.
14  */
15 class Fecha {
16     unsigned dia; ///< Información de día.
17     unsigned mes; ///< Información de mes.
18     unsigned año; ///< Información de año.
19     unsigned hora; ///< Información de hora.
20     unsigned min; ///< Información de minutos.
21     static const unsigned diasMes[12]; ///< Almacena los días por mes.
22

```

Figura 24: Captura de pantalla de los atributos miembros de la clase Fecha.

En la siguiente captura de pantalla se muestra el código fuente del constructor por defecto de la clase. Se puede observar que se delega la inicialización de las variables en una función externa, por lo que daremos por no cumplido el estándar.

```

13 /**
14  * @brief Constructor por defecto de la clase Fecha.
15  * Crea una fecha con la hora actual.
16  */
17 Fecha::Fecha() {
18     time_t tiempoActual;
19     struct tm *fechaActual;
20     time(&tiempoActual); ///< Obtiene la hora actual del sistema.
21     fechaActual = localtime(&tiempoActual); ///< Decodifica la hora en campos separados.
22     leerTiempo(*fechaActual);
23 }
24

```

Figura 25: Captura de pantalla en la que se muestra el incumplimiento del estándar OOP53-CPP.

Para corregirlo, añadimos la inicialización de los atributos miembros de la clase al final, en el orden de declaración de los mismos:

```

13
14 /**
15  * @brief Constructor por defecto de la clase Fecha.
16  * Crea una fecha con la hora actual.
17  */
18 Fecha::Fecha() {
19     time_t tiempoActual;
20     struct tm *fechaActual;
21     time(&tiempoActual); /// Obtiene la hora actual del sistema.
22     fechaActual = localtime(&tiempoActual); /// Decodifica la hora en campos separados.
23     dia = fechaActual->tm_mday;
24     mes = fechaActual->tm_mon + 1;
25     anio = fechaActual->tm_year + 1900;
26     hora = fechaActual->tm_hour;
27     min = fechaActual->tm_min;
28 }
29

```

Figura 26: Captura de pantalla en la que se muestra el cumplimiento del estándar OOP53-CPP.

Tanto para el constructor parametrizado como para el constructor por copia, se puede observar en las siguientes capturas de pantalla que se cumple con el estándar:

```

29
30 /**
31  * @brief Constructor parametrizado de la clase Fecha.
32  * Crea una fecha concreta. Devuelve una excepción ErrorFechaIncorrecta si la fecha introducida no es correcta.
33  */
34 Fecha::Fecha(unsigned aDia, unsigned aMes, unsigned aAnio, unsigned aHora,
35              unsigned aMin) {
36     comprobarFecha(aDia, aMes, aAnio, aHora, aMin); /// Filtra las fechas incorrectas.
37     dia = aDia;
38     mes = aMes;
39     anio = aAnio;
40     hora = aHora;
41     min = aMin;
42 }
43

```

Figura 27: Captura de pantalla en la que se muestra el cumplimiento del estándar OOP53-CPP.

```

29
30 /**
31  * @brief Constructor por copia de la clase Fecha.
32  */
33 Fecha(const Fecha &f) :
34     dia(f.dia), mes(f.mes), anio(f.anio), hora(f.hora), min(f.min) {
35 }
36

```

Figura 28: Captura de pantalla en la que se muestra el cumplimiento del estándar OOP53-CPP.

Para la clase Libro, tenemos la siguiente lista de miembros:

```

13 /**
14  * @brief Clase que representa toda la información necesaria de un libro.
15  */
16 class Libro {
17     string titulo;      ///< Título del libro.
18     string autores;     ///< Autor/es del libro.
19     string editorial;   ///< Editorial que publica el libro.
20     string ISBN;        ///< Código ISBN identificativo del libro (International Standard Book Number).
21     int año;            ///< Año en el que se publica el libro.
22     float precioActual; ///< Precio actual del libro.
23 }

```

Figura 29: Captura de pantalla de los atributos miembros de la clase Libro.

En la siguiente captura de pantalla se puede comprobar que en el constructor por defecto de la clase no se cumple el estándar, ya que no respeta el orden de los atributos miembros de la clase:

```

9 /**
10  * @brief Constructor por defecto de la clase Libro.
11  * @pre Al estar inicializando un objeto de la clase, todos los atributos aparecen vacíos o inicializados a cero.
12  */
13 Libro::Libro() {
14     titulo = "";
15     autores = "";
16     editorial = "";
17     ISBN = "";
18     precioActual = 0;
19     año = 0;
20 }

```

Figura 30: Captura de pantalla en la que se muestra el incumplimiento del estándar OOP53-CPP.

Esto se soluciona reordenando la inicialización en el orden correcto:

```

8
9 /**
10  * @brief Constructor por defecto de la clase Libro.
11  * @pre Al estar inicializando un objeto de la clase, todos los atributos aparecen vacíos o inicializados a cero.
12  */
13 Libro::Libro() {
14     titulo = "";
15     autores = "";
16     editorial = "";
17     ISBN = "";
18     año = 0;
19     precioActual = 0;
20 }

```

Figura 31: Captura de pantalla en la que se muestra el cumplimiento del estándar OOP53-CPP.

Para el constructor parametrizado y el constructor por copia observamos que sí se cumple el estándar:

```

22 /**
23  * @brief Constructor parametrizado de la clase Libro.
24  * @param [in] aTitulo string. Título del libro.
25  * @param [in] aAutores string. Autor/es del libro.
26  * @param [in] aEditorial string. Editorial del libro.
27  * @param [in] aISBN string. ISBN del libro.
28  * @param [in] aAño int. Año de publicación del libro.
29  * @param [in] aPrecioActual float. Precio del libro.
30  */
31 Libro::Libro(string aTitulo, string aAutores, string aEditorial, string aISBN, int aAño, float aPrecioActual) {
32     titulo = aTitulo;
33     autores = aAutores;
34     editorial = aEditorial;
35     isbn = aISBN;
36     año = aAño;
37     precioActual = aPrecioActual;
38 }
39

```

Figura 32: Captura de pantalla en la que se muestra el cumplimiento del estándar OOP53-CPP.

```

29
30 /**
31  * @brief Constructor por copia de la clase Libro.
32  * @param [in] lib Libro(dir). Instancia de la clase Libro de la cual se va realizar una copia.
33  */
34 Libro(const Libro &lib) {
35     this->titulo = lib.titulo;
36     this->autores = lib.autores;
37     this->editorial = lib.editorial;
38     this->isbn = lib.isbn;
39     this->año = lib.año;
40     this->precioActual = lib.precioActual;
41 }
42

```

Figura 33: Captura de pantalla en la que se muestra el cumplimiento del estándar OOP53-CPP.

Para la clase plantilla lista_sin, tenemos la siguiente lista de miembros:

```

22
23 template<class T>
24 /**
25  * @brief Plantilla genérica de estructura de datos lista enlazada.
26  */
27 class lista_sin {
28     nodo<T> *nuevo; ///< Nuevo nodo en la lista enlazada.
29     nodo<T> *primero; ///< Primer nodo en la lista enlazada.
30     nodo<T> *ultimo; ///< Último nodo en la lista enlazada.
31     unsigned numNodos; ///< Número de nodos de la lista enlazada.
32

```

Figura 34: Captura de pantalla de los atributos miembros de la clase lista_sin.

Podemos observar en las siguientes capturas de pantalla que tanto el constructor por defecto como el constructor por copia cumplen con el estándar:

```
109
110 template<class T>
111 lista_sin<T>::lista_sin() {
112     nuevo = NULL;
113     primero = NULL;
114     ultimo = NULL;
115     numElem = 0;
116 }
117
```

Figura 35: Captura de pantalla en la que se muestra el cumplimiento del estándar OOP53-CPP.

```
117
118 template<class T>
119 lista_sin<T>::lista_sin(lista_sin &list) {
120     nuevo = NULL;
121     primero = NULL;
122     ultimo = NULL;
123     numElem = 0;
124     for (unsigned i = 0; i < list.tamano(); i++) {
125         this->aumenta(list.lee(i));
126     }
127 }
128
```

Figura 36: Captura de pantalla en la que se muestra el cumplimiento del estándar OOP53-CPP.

Para la clase clase PedidoBiblioteca, tenemos la siguiente lista de miembros:

```
12
13 /**
14  * @brief Clase que representa de manera genérica un pedido hecho por la biblioteca.
15  */
16 class PedidoBiblioteca {
17     fecha fecha;        ///< Queda registrada la fecha del pedido actualizada con la fecha y hora del sistema.
18     float importe;      ///< Importe total de todos los usuarios.
19     bool tramitado;     ///< Booleano a true si el pedido esta tramitado, false en otro caso.
20     unsigned num;       ///< Número de pedido de biblioteca.
21     lista_sin<PedidoUsuario> pedido_usu; ///< Registro en la estructura de datos del pedido de un usuario.
22
```

Figura 37: Captura de pantalla de los atributos miembros de la clase PedidoBiblioteca.

El constructor por defecto, parametrizado y por copia de la clase no respetan el orden establecido de los atributos miembros, no cumpliendo de este modo con el estándar:

```
8 /**
9  * @brief Constructor por defecto de la clase PedidoBiblioteca.
10 */
11 PedidoBiblioteca::PedidoBiblioteca() :
12     fecha() {
13         importe = 0;
14         tramitado = false;
15         this->pedido_usu = pedido_usu;
16         num = 0;
17     }
18
```

Figura 38: Captura de pantalla en la que se muestra el incumplimiento del estándar OOP53-CPP.


```

26
27
28  /**
29   * @brief Constructor parametrizado de la clase PedidoBiblioteca.
30   * @param [in] anum unsigned.
31   */
32  PedidoBiblioteca(unsigned anum) :
33      fecha() {
34      importe = 0;
35      tramitado = false;
36      this->pedido_usu = pedido_usu;
37      this->num = anum;
38  }

```

Figura 39: Captura de pantalla en la que se muestra el incumplimiento del estándar OOP53-CPP.

```

38
39
40  /**
41   * @brief Constructor por copia de la clase PedidoBiblioteca.
42   * @param [in] pedbi PedidoBiblioteca (dir). Instancia de PedidoBiblioteca que se quiere copiar.
43   */
44  PedidoBiblioteca(PedidoBiblioteca &pedbi) {
45      this->fecha = pedbi.fecha;
46      this->importe = pedbi.importe;
47      this->tramitado = pedbi.tramitado;
48      this->pedido_usu = pedbi.pedido_usu;
49      this->num = pedbi.num;
50  }

```

Figura 40: Captura de pantalla en la que se muestra el incumplimiento del estándar OOP53-CPP.

Para poder cumplir con el estándar lo único que tenemos que hacer es ordenar la inicialización de los atributos del siguiente modo:

```

7
8  /**
9   * @brief Constructor por defecto de la clase PedidoBiblioteca.
10  */
11  PedidoBiblioteca::PedidoBiblioteca() :
12      fecha() {
13      importe = 0;
14      tramitado = false;
15      num = 0;
16      this->pedido_usu = pedido_usu;
17  }
18

```

Figura 41: Captura de pantalla en la que se muestra el cumplimiento del estándar OOP53-CPP.

```

26
27
28  /**
29   * @brief Constructor parametrizado de la clase PedidoBiblioteca.
30   * @param [in] anum unsigned.
31   */
32  PedidoBiblioteca(unsigned anum) :
33      fecha() {
34          importe = 0;
35          tramitado = false;
36          this->num = anum;
37          this->pedido_usu = pedido_usu;
38      }

```

Figura 42: Captura de pantalla en la que se muestra el cumplimiento del estándar OOP53-CPP.

```

38
39
40  /**
41   * @brief Constructor por copia de la clase PedidoBiblioteca.
42   * @param [in] pedbi PedidoBiblioteca (dir). Instancia de PedidoBiblioteca que se quiere copiar.
43   */
44  PedidoBiblioteca(PedidoBiblioteca &pedbi) {
45      this->fecha = pedbi.fecha;
46      this->importe = pedbi.importe;
47      this->tramitado = pedbi.tramitado;
48      this->num = pedbi.num;
49      this->pedido_usu = pedbi.pedido_usu;
50  }

```

Figura 43: Captura de pantalla en la que se muestra el cumplimiento del estándar OOP53-CPP.

Para la clase clase PedidoUsuario, tenemos la siguiente lista de miembros:

```

13
14
15  /**
16   * @brief Clase que representa el pedido que un usuario hace a la biblioteca.
17   */
18  class PedidoUsuario {
19      fecha fecha;      ///< Fecha que queda registrada al hacer un pedido.
20      int prioridad;    ///< Prioridad concedida al pedido del usuario.
21      float precio;     ///< Precio del pedido que ha hecho el usuario.
22      bool tramitado;   ///< Booleano que nos va a indicar si el pedido ha sido tramitado o no.
23      usuario *usuario; ///< Puntero que referencia a un usuario en concreto.
24      libro *libro;     ///< Puntero que referencia a un libro en concreto.

```

Figura 44: Captura de pantalla de los atributos miembros de la clase PedidoUsuario.

Para los constructores por defecto y parametrizado podemos observar que no se cumple el estándar, como se puede observar en las siguientes capturas de pantalla:

```

8  /**
9  * @brief Constructor por defecto del pedido de un usuario en concreto.
10 * @return La inicialización de un pedido por parte del usuario con su fecha, precio, etc.
11 */
12 PedidoUsuario::PedidoUsuario() :
13     fecha() {          ///< Fecha que queda registrada al hacer un pedido.
14     prioridad = 0;      ///< Prioridad concedida al pedido del usuario.
15     precio = 0;         ///< Precio del pedido inicializado a cero.
16     tramitado = false;  ///< De entrada el pedido aun no ha sido tramitado.
17     libro = NULL;       ///< De entrada no se esta apuntando a ningun libro en concreto.
18     usuario = NULL;     ///< De entrada no se esta apuntando a ningun usuario en concreto.
19 }
20

```

Figura 45: Captura de pantalla en la que se muestra el incumplimiento del estándar OOP53-CPP.

```

21 /**
22 * @brief Constructor parametrizado de la clase PedidoUsuario.
23 * @param [in] libro Libro (ref).
24 * @param [in] usuario Usuario (ref).
25 * @param [in] aFecha Fecha.
26 * @param [in] aPrioridad int.
27 * @param [in] aPrecio float.
28 * @param [in] aTramitado bool.
29 */
30 PedidoUsuario::PedidoUsuario(Libro *libro, Usuario *usuario, Fecha aFecha, int aPrioridad, float aPrecio, bool aTramitado) {
31     fecha = aFecha;      ///< Copia de la fecha que queda registrada al hacer un pedido.
32     prioridad = aPrioridad; ///< Copia de la prioridad que queda registrada al hacer un pedido.
33     precio = aPrecio;    ///< Copia del precio de un pedido.
34     tramitado = aTramitado; ///< Copia de la tramitación de un pedido.
35     this->usuario = usuario; ///< Referencia al usuario mediante el objeto this.
36     this->libro = libro;    ///< Referencia al libro mediante el objeto this.
37 }
38

```

Figura 46: Captura de pantalla en la que se muestra el incumplimiento del estándar OOP53-CPP.

Para hacer que se cumpla con el estándar, debemos inicializar los atributos miembros según el orden establecido en la declaración de la clase, como se muestra en las siguientes capturas de pantalla:

```

7  /**
8  * @brief Constructor por defecto del pedido de un usuario en concreto.
9  * @return La inicialización de un pedido por parte del usuario con su fecha, precio, etc.
10 */
11
12 PedidoUsuario::PedidoUsuario() :
13     fecha() {          ///< Fecha que queda registrada al hacer un pedido.
14     prioridad = 0;      ///< Prioridad concedida al pedido del usuario.
15     precio = 0;         ///< Precio del pedido inicializado a cero.
16     tramitado = false;  ///< De entrada el pedido aun no ha sido tramitado.
17     usuario = NULL;     ///< De entrada no se esta apuntando a ningun usuario en concreto.
18     libro = NULL;       ///< De entrada no se esta apuntando a ningun libro en concreto.
19 }
20

```

Figura 47: Captura de pantalla en la que se muestra el cumplimiento del estándar OOP53-CPP.

```

20
21 /**
22  * @brief Constructor parametrizado de la clase PedidoUsuario.
23  * @param [in] libro Libro (ref).
24  * @param [in] usuario Usuario (ref).
25  * @param [in] aFecha Fecha.
26  * @param [in] aPrioridad int.
27  * @param [in] aPrecio float.
28  * @param [in] aTramitado bool.
29  */
30 PedidoUsuario::PedidoUsuario(Fecha aFecha, int aPrioridad, float aPrecio, bool aTramitado, Usuario *usuario, Libro *libro) {
31     fecha = aFecha;      ///< Copia de la fecha que queda registrada al hacer un pedido.
32     prioridad = aPrioridad; ///< Copia de la prioridad que queda registrada al hacer un pedido.
33     precio = aPrecio;     ///< Copia del precio de un pedido.
34     tramitado = aTramitado; ///< Copia de la tramitación de un pedido.
35     this->usuario = usuario; ///< Referencia al usuario mediante el objeto this.
36     this->libro = libro;    ///< Referencia al libro mediante el objeto this.
37 }
38

```

Figura 48: Captura de pantalla en la que se muestra el cumplimiento del estándar OOP53-CPP.

Para la clase Usuario, tenemos la siguiente lista de miembros:

```

12
13 /**
14  * @brief Clase que representa a un usuario de la biblioteca.
15  */
16 class Usuario {
17     string nombre;    ///< Nombre del usuario.
18     string clave;     ///< Clave que lo autenticará ante el sistema.
19     string login;     ///< Login del usuario.

```

Figura 49: Captura de pantalla de los atributos miembros de la clase Usuario.

En el caso de esta clase, solo se dispone de un constructor por defecto. En este, no se respeta el estándar ya que no se inicializan los atributos miembro en el orden canónico, como se puede observar en la siguiente captura de pantalla:

```

8
9 /**
10  * @brief Constructor por defecto de la clase Usuario.
11  */
12 Usuario::Usuario() {
13     nombre = "";
14     clave = "";
15     login = "";
16 }

```

Figura 50: Captura de pantalla en la que se muestra el incumplimiento del estándar OOP53-CPP.

3.10. MSC52-CPP. Las funciones que devuelven un valor deben devolver un valor desde todas las rutas de salida.

3.10.1. Enlace al sitio web del estándar seleccionado

<https://wiki.sei.cmu.edu/confluence/display/cplusplus/MS52-CPP.+Value-returning+functions+must+return+a+value+from+all+exit+paths>

3.10.2. Explicación sobre su utilidad

La utilidad de la aplicación de este estándar es bastante conocida. Este estándar nos dice que, siempre que una función devuelve un valor, cada ruta de ejecución de la misma debe de devolver siempre un valor. En el caso de que en algún caso no se devolviera nada, esto podría desembocar en un comportamiento indefinido de la aplicación.

3.10.3. Aplicación del estándar al proyecto

En nuestro proyecto, todas las funciones que devuelven un valor cumplen con este estándar. Dentro de las funciones que devuelven un valor, en nuestro proyecto hay tres tipos diferenciados: las funciones que solo tienen una ruta de ejecución, las funciones que tienen más de una ruta de ejecución y devuelven un valor en cada ruta, y las funciones que tienen más de una ruta de ejecución pero no devuelven un valor en todas.

Las funciones que solo tienen una ruta de ejecución las obviaremos en este informe, ya que el cumplimiento de este estándar resulta algo trivial de comprobar.

Para el segundo tipo de funciones pondremos un par de ejemplos a continuación en los que se puede comprobar claramente que no existe ninguna ruta de ejecución en la que no se devuelva ningún valor.

```

14  /**
15   * @brief Introduce un nuevo Usuario en la biblioteca.
16   * @param [in] login string. Login del usuario. Login del nuevo Usuario.
17   * @param [in] nombre string. Nombre del Usuario. Nombre del nuevo Usuario.
18   * @param [in] clave string. Clave del Usuario. Clave del nuevo Usuario.
19   * @return bool. True si no se puede introducir el Usuario, false en cualquier otro caso.
20   */
21  bool Biblioteca::nuevoUsuario(string login, string nombre, string clave) {
22      usu->rellena(login, nombre, clave);
23      for(int i;
24          if (usu.tamano() == 0) { /// devuelve true si está el Usuario y ya no se puede introducir Usuario por motivos obvios.
25              usu.aumenta(usu);
26              return true;
27          } else {
28              for (i = 0; i < usu.tamano(); i++) {
29                  if (usu->daClave() == usu.lee(i)->daClave()) {
30                      return false;
31                  } else {
32                      usu.aumenta(usu);
33                      return true;
34                  }
35              }
36          }
37      }
38      return false;
39  }
40  }
41  }

```

Figura 51: Captura de pantalla del cumplimiento del estándar MSC52-CPP.

```

67  /**
68   * @brief Comparar fechas.
69   * @param [in] f Fecha(dir, const).
70   * @return bool. True en el caso de que la fecha actual sea menor que la fecha parámetro, false en cualquier otro caso.
71   */
72  bool Fecha::operator<(const Fecha &f) {
73      if (anio < f.anio)
74          return true;
75      else if (anio > f.anio)
76          return false;
77      if (mes < f.mes)
78          return true;
79      else if (mes > f.mes)
80          return false;
81      if (dia < f.dia)
82          return true;
83      else if (dia > f.dia)
84          return false;
85      if (hora < f.hora)
86          return true;
87      else if (hora > f.hora)
88          return false;
89      if (min < f.min)
90          return true;
91      else if (min > f.min)
92          return false;
93      return false;
94  }
95  }
96  }
97  }
98  }
99  }

```

Figura 52: Captura de pantalla del cumplimiento del estándar MSC52-CPP.

Para el tercer tipo de funciones, si bien no se devuelve un valor en alguna de las rutas de ejecución, esto es debido a que la devolución de un valor no válido produciría un error de ejecución. Por este motivo, no se devuelve un valor sino que se ejecuta una excepción que detiene la ejecución para evitar comportamientos impredecibles.

Teniendo en cuenta lo anterior, en el resto de rutas de ejecución que no causan excepciones sí se devuelve siempre un valor. Por este motivo, se considera cumplido el estándar. A continuación, dejamos un par de capturas de funciones de este tipo:

```

230
231 /**
232  * @brief Devuelve una lista de las referencias de los pedidos de biblioteca tramitados.
233  * @return Lista de referencias a los pedidos de biblioteca tramitados.
234  */
235 lista_sit<PedidoBiblioteca *> * Biblioteca::buscaPedidosBibliotecaTramitados() {
236     unsigned i = 0;
237     lista_sit<PedidoBiblioteca *> * biTramitados = new lista_sit<PedidoBiblioteca *>;
238
239     while (i < pedidoBli.tamano()) { /// Busca pedidos tramitados de biblioteca y los devuelve en una lista.
240         if (pedidoBli.lee(i)->daTrami() == true) {
241             biTramitados->aumenta(pedidoBli.lee(i));
242         }
243         i++;
244     }
245     if (biTramitados->tamano() != 0)
246         return biTramitados;
247     else
248         throw excepcionesBi::pedidoBibliotecaNoencontrado();
249 }
250

```

Figura 53: Captura de pantalla del cumplimiento del estándar MSC52-CPP.

```

207
208 /**
209  * @brief Devuelve una lista con las referencias a los pedidos de biblioteca pendientes.
210  * @return Lista de referencias a los pedidos de biblioteca pendientes.
211  */
212 lista_sit<PedidoBiblioteca *> * Biblioteca::buscaPedidosBibliotecaPendientes() {
213     unsigned i = 0;
214     lista_sit<PedidoBiblioteca *> * biPendientes = new lista_sit<
215         PedidoBiblioteca *>;
216     if (biPendientes->tamano() == 0)
217         throw excepcionesBi::pedidoBibliotecaNoencontrado();
218
219     while (i < pedidoBli.tamano()) { /// Se buscan los pedidos de biblioteca pendientes y se devuelven en una lista.
220         if (pedidoBli.lee(i)->daTrami() == false && pedidoBli.lee(i)->daImporte() > 1) {
221             biPendientes->aumenta(pedidoBli.lee(i));
222         }
223         i++;
224     }
225     if (biPendientes->tamano() != 0)
226         return biPendientes;
227     else
228         throw excepcionesBi::pedidoBibliotecaNoencontrado();
229 }
230

```

Figura 54: Captura de pantalla del cumplimiento del estándar MSC52-CPP.

4. Análisis estático automatizado de código

4.1. Error nº1: Memory leak

4.1.1. Origen/explicación del error detectado

En la siguiente captura de pantalla se muestra el código fuente de la función *modifica* de la clase plantilla *lista_sin* de nuestra aplicación.

```

174 template<class T>
175 void lista_sin<T>::modifica(T elem, unsigned pos) {
176     unsigned i = 0;
177     nodo<T> *iter;
178     iter = new struct nodo<T>;
179     iter = primero;
180     if (iter == NULL) {
181         throw ErrorElemento();
182     } else {
183         while (iter && i < pos) {
184             i++;
185             iter = iter->sige;
186         }
187         iter->date = elem;
188     }
189 }
190

```

Figura 55: Captura de pantalla de la función *modifica*

En la línea 179 cppchek nos indica que ha detectado un error de código, en concreto, se trata de un error de pérdida de memoria para la variable *iter*. Esto es debido a que en la línea 177 se declara la variable (la cual es un puntero a una estructura llamada *nodo*, que hemos definido nosotros anteriormente en el mismo archivo), y en la línea 178 se inicializa con la palabra reservada *new*. Al realizar esto anterior, hemos reservado memoria dinámica para la estructura y la hemos inicializado.

(cppcheck error) Memory leak: iter lista_sin.h /prac3G5 line 179 cppcheck Problem

Figura 56: Detalle del mensaje de error de cppcheck

El error se origina cuando, en la línea 179, hacemos que la variable *iter* (que es un puntero) apunte al atributo de la clase *primero* sin liberar la memoria de la estructura con la que inicializamos anteriormente. Esto hace que la memoria que ocupa dicha estructura no se libere y que, por lo tanto, se produzca una pérdida de memoria.

4.1.2. Corrección realizada para solventar el error

La manera de resolver esta incidencia es de lo más trivial. Tan solo deberemos realizar la declaración de la variable y, acto seguido, asignarle la dirección de

memoria del atributo de la clase *primero*. A continuación, dejamos una captura de pantalla del estado de la función tras la resolución del error:

```

173
174 template<class T>
175 void lista_sin<T>::modifica(T elem, unsigned pos) {
176     unsigned i = 0;
177     nodo<T> *iter;
178     iter = primero;
179     if (iter == NULL) {
180         throw ErrorElemento();
181     } else {
182         while (iter && i < pos) {
183             i++;
184             iter = iter->sige;
185         }
186         iter->date = elem;
187     }
188 }
189

```

Figura 57: Captura de pantalla de la función modifica tras la resolución del error

nota: Cabe destacar que también se podría haber resuelto este error realizando un "delete iter" antes de asignarle la dirección de memoria del atributo *primero* pero en este hemos descartado esta opción ya que la estructura con la que se inicializa no se utiliza para nada y supondría la realización de operaciones innecesarias.

4.2. Error nº2: Memory leak

4.2.1. Origen/explicación del error detectado

En la siguiente captura de pantalla se muestra el código fuente de la función *elimina_dato* de la clase plantilla *lista_sin* de nuestra aplicación.

```

213
214 template<class T>
215 T lista_sin<T>::elimina_dato(unsigned pos) {
216     unsigned i = 1;
217     T var;
218     nodo<T> *viejo;
219     viejo = new struct nodo<T>;
220     viejo = primero;
221     nuevo = primero;
222     if (numElem == 1) {
223         viejo = primero;
224         var = viejo->date;
225         delete viejo;
226         numElem--;
227         primero = NULL;
228         ultimo = NULL;
229         return var;
230     } else {
231         while (nuevo && i < pos) {
232             i++;
233             nuevo = nuevo->sige;
234         }
235         viejo = nuevo->sige;
236         nuevo->sige = viejo->sige;
237         var = viejo->date;
238         delete viejo;
239         numElem--;
240         return var;
241     }
242 }
243

```

Figura 58: Captura de pantalla de la función elimina_dato

Este error se materializa de manera idéntica que al anterior error: Primero, la variable *viejo* (puntero) se declara en la línea 218. A continuación, en la línea 219 se reserva memoria para una nueva instancia de la estructura *nodo*. En la línea 220 se le asigna a la variable *viejo* la dirección de memoria del atributo de clase *primero*, generándose en este momento la pérdida de memoria al no haber liberado la memoria a la que se apuntaba anteriormente.

(cppcheck error) Memory leak: viejo lista_sin.h /prac3G5 line 220 cppcheck Problem

Figura 59: Detalle del mensaje de error de cppcheck

4.2.2. Corrección realizada para solventar el error

Para solucionar este error simplemente deberemos eliminar la línea 219 de la función. De esta manera, declaramos la variable *viejo* y justo a continuación le asignamos la dirección de memoria del atributo de clase *primero*. De esta manera el mensaje de pérdida de memoria habrá desaparecido:

```

213
214 template<class T>
215 T lista_sin<T>::elimina_dato(unsigned pos) {
216     unsigned i = 1;
217     T var;
218     nodo<T> *viejo;
219     viejo = primero;
220     nuevo = primero;
221     if (numElem == 1) {
222         viejo = primero;
223         var = viejo->date;
224         delete viejo;
225         numElem--;
226         primero = NULL;
227         ultimo = NULL;
228         return var;
229     } else {
230         while (nuevo && i < pos) {
231             i++;
232             nuevo = nuevo->sige;
233         }
234         viejo = nuevo->sige;
235         nuevo->sige = viejo->sige;
236         var = viejo->date;
237         delete viejo;
238         numElem--;
239         return var;
240     }
241 }
242

```

Figura 60: Captura de pantalla de la función elimina_dato

4.3. Error nº3: Mismatching allocation and deallocation

4.3.1. Origen/explicación del error detectado

A continuación presentamos una captura de pantalla de la función *limpia* de la clase plantilla *lista_sin*.

```

247
248 template<class T>
249 void lista_sin<T>::limpia() {
250     while (primero) {
251         nuevo = primero;
252         primero = primero->sige;
253         delete[] nuevo;
254     }
255 }
256

```

Figura 61: Captura de pantalla de la función limpia

En esta función, en la línea 253, cppcheck nos indica que hay un error. Dicho error es originado dado que hacemos un "delete[]", es decir, una liberación de memoria para un vector/array de punteros. En el caso de la variable *nuevo*, podemos comprobar que se trata de un atributo de la clase *lista_sin* y que se

trata de un puntero a una estructura llamada *nodo*. Justo esto anterior es lo que genera el error, es decir, estamos intentando liberar la memoria asignada de un vector/array cuando la variable **no es ninguno de estos**.

Errors (6 items)				
(cppcheck error) Mismatching allocation and deallocation: lista_sin < Libro * >::nuevo	lista_sin.h	/prac3G5	line 253	cppcheck Problem
(cppcheck error) Mismatching allocation and deallocation: lista_sin < PedidoBiblioteca * >::nuevo	lista_sin.h	/prac3G5	line 253	cppcheck Problem
(cppcheck error) Mismatching allocation and deallocation: lista_sin < PedidoUsuario * >::nuevo	lista_sin.h	/prac3G5	line 253	cppcheck Problem
(cppcheck error) Mismatching allocation and deallocation: lista_sin < Usuario * >::nuevo	lista_sin.h	/prac3G5	line 253	cppcheck Problem
(cppcheck error) Mismatching allocation and deallocation: lista_sin < Usuario >::nuevo	lista_sin.h	/prac3G5	line 253	cppcheck Problem
(cppcheck error) Mismatching allocation and deallocation: lista_sin::nuevo	lista_sin.h	/prac3G5	line 253	cppcheck Problem

Figura 62: Detalle del mensaje de error de cppcheck

4.3.2. Corrección realizada para solventar el error

Para corregir este error lo único que deberemos realizar es sustituir la sentencia `delete[] nuevo` por la sentencia `delete nuevo`. Una vez realizado esto el error habrá desaparecido, como se puede comprobar en la siguiente captura de pantalla:

```

247
248 template<class T>
249 void lista_sin<T>::limpia() {
250     while (primero) {
251         nuevo = primero;
252         primero = primero->sige;
253         delete nuevo;
254     }
255 }
256

```

Figura 63: Captura de pantalla de la función limpia tras la resolución del error

4.4. Error nº4: Class has a constructor with 1 argument that is not explicit

4.4.1. Origen/explicación del error detectado

En la siguiente captura de pantalla se observa el constructor parametrizado de la clase *PedidoBiblioteca*.

```

26
27 /**
28  * @brief Constructor parametrizado de la clase PedidoBiblioteca.
29  * @param [in] anum unsigned.
30  */
31 PedidoBiblioteca(unsigned anum) :
32     fecha() {
33     importe = 0;
34     tramitado = false;
35     this->num = anum;
36     this->pedido_usu = pedido_usu;
37 }
38

```

Figura 64: Captura de pantalla del constructor parametrizado de la clase PedidoBiblioteca

Como se puede observar en la captura de pantalla, en este constructor se hace uso de una lista de inicializadores de miembro, la cual inicializa el valor de los atributos miembro antes de la ejecución del cuerpo del constructor.

En la lista observamos que se encuentra el atributo miembro *fecha* inicializado con su constructor por defecto. Esto es debido a que entre paréntesis no hemos escrito ningún nombre de variable, ya que no se le pasa ningún parámetro del tipo fecha al constructor.

No obstante, el parámetro *anum* no se usa en la lista, sino que se usa posteriormente en el cuerpo del constructor. Aquí es donde se haya el origen del error.

Como bien hemos estudiado en los primeros cursos de la carrera, cuando usamos la lista de inicializadores de miembro, es común asignar un valor por defecto a todos los parámetros del constructor. De este modo, si no se le pasara algún parámetro a este constructor, inicializaría el atributo miembro con el valor por defecto que tenga declarado el parámetro en cuestión.

El parámetro *anum* no tiene ningún valor por defecto y es por este motivo que la herramienta cppcheck nos genera el error.



Figura 65: Detalle del mensaje de error de cppcheck

4.4.2. Corrección realizada para solventar el error

La forma de corregir el error generado es muy sencilla. Nos limitaremos a asignarle un valor por defecto al parámetro *anum*, en este caso el mismo valor que se le asigna al atributo miembro *num* en el constructor por defecto. De este modo, podremos comprobar que el error habrá desaparecido.

```

26
27  /**
28   * @brief Constructor parametrizado de la clase PedidoBiblioteca.
29   * @param [in] anum unsigned.
30   */
31  PedidoBiblioteca(unsigned anum = 0) :
32      fecha(){
33          importe = 0;
34          tramitado = false;
35          this->num = anum;
36          this->pedido_usu = pedido_usu;
37      }
38

```

Figura 66: Captura de pantalla del constructor parametrizado de la clase PedidoBiblioteca tras la corrección del error

Nota: Una segunda forma de corregir el error habría sido declarar el constructor como explícito, es decir, añadir *explicit* a la declaración del constructor. Hemos descartado esta opción ya que en algunos casos esto podría ocasionar un comportamiento indeseado en la ejecución del programa en ciertas situaciones.

4.5. Error nº5: Parameter can be declared with const

4.5.1. Origen/explicación del error detectado

Este error tiene su origen en el constructor por copia de la clase *PedidoBiblioteca*. A continuación, se presenta una captura de pantalla de dicho constructor.

```

38
39  /**
40   * @brief Constructor por copia de la clase PedidoBiblioteca.
41   * @param [in] pedbi PedidoBiblioteca (dir). Instancia de PedidoBiblioteca que se quiere copiar.
42   */
43  PedidoBiblioteca(PedidoBiblioteca &pedbi) {
44      this->fecha = pedbi.fecha;
45      this->importe = pedbi.importe;
46      this->tramitado = pedbi.tramitado;
47      this->num = pedbi.num;
48      this->pedido_usu = pedbi.pedido_usu;
49  }
50

```

Figura 67: Captura de pantalla del constructor por copia de la clase *PedidoBiblioteca*

Como se puede observar en la captura, cppcheck nos indica que el error se encuentra en la línea 43 del archivo. Nos recomienda que el parámetro *pedbi* puede ser declarado como constante. Este parámetro es el la instancia del objeto del cuál se hace una copia

Es bastante común (y recomendable) que en los constructores por copia y en los operadores de asignación se pasen como constante los objetos de los cuales queremos copiar el valor de sus atributos. Esto se realiza para evitar que, bien por descuido o bien por error de programación, se modifiquen los valores de los atributos miembros de estos parámetros.

Es por este motivo que cppcheck nos avisa de que podemos hacer este parámetro constante.

(cppcheck style) Parameter 'pedbi' can be declared with const PedidoBiblioteca.h /prac3G5 line 43 cppcheck Problem

Figura 68: Detalle del mensaje de error de cppcheck

4.5.2. Corrección realizada para solventar el error

Para corregir este error tan solo deberemos hacer constante el parámetro *pedbi*. Una vez hecho esto habrá desaparecido el aviso de cppcheck.

```

38
39  /**
40   * @brief Constructor por copia de la clase PedidoBiblioteca.
41   * @param [in] pedbi PedidoBiblioteca (dir). Instancia de PedidoBiblioteca que se quiere copiar.
42   */
43  PedidoBiblioteca(const PedidoBiblioteca &pedbi) {
44      this->fecha = pedbi.fecha;
45      this->importe = pedbi.importe;
46      this->tramitado = pedbi.tramitado;
47      this->num = pedbi.num;
48      this->pedido_usu = pedbi.pedido_usu;
49  }
50

```

Figura 69: Captura de pantalla del constructor por copia de la clase PedidoBiblioteca tras la corrección del error

Nota: En el caso de que en nuestro constructor necesitáramos por algún motivo modificar el valor de alguno de los atributos miembro del parámetro y, aún así, nos saltara el aviso de cppcheck deberíamos hacer caso omiso ya que de este modo nos daría un error al compilar el código.

Referencias

- [1] <https://www.sei.cmu.edu/about/divisions/cert/index.cfm>
- [2] <https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards>
- [3] <https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard>
- [4] <https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682>
- [5] <https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL51-CPP.+Do+not+declare+or+define+a+reserved+identifier>
- [6] <https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL52-CPP.+Never+qualify+a+reference+type+with+const+or+volatile>
- [7] <https://wiki.sei.cmu.edu/confluence/display/cplusplus/OOP53-CPP.+Write+constructor+member+initializers+in+the+canonical+order>
- [8] <https://wiki.sei.cmu.edu/confluence/display/cplusplus/MS52-CPP.+Value-returning+functions+must+return+a+value+from+all+exit+paths>
- [9] <https://wiki.sei.cmu.edu/confluence/display/cplusplus/FIO51-CPP.+Close+files+when+they+are+no+longer+needed>
- [10] <https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL59-CPP.+Do+not+define+an+unnamed+namespace+in+a+header+file>
- [11] <https://wiki.sei.cmu.edu/confluence/display/cplusplus/FIO50-CPP.+Do+not+alternately+input+and+output+from+a+file+stream+without+an+intervening+positioning+call>
- [12] <https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR51-CPP.+Handle+all+exceptions>
- [13] <https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM51-CPP.+Properly+deallocate+dynamically+allocated+resources>
- [14] <https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM52-CPP.+Detect+and+handle+memory+allocation+errors>
- [15] <http://cppcheck.sourceforge.net/>