Nombre: Adrián Fernández Ortega
DNI: 02566277Y

PNL IFCT0609: Gestión de duplicados

1. Análisis

1.1. Enunciado

Realizar un programa en cualquier lenguaje que lea, del directorio desde el que se le invoque, los archivos que tiene y que detecte, con el empleo de una función hash, aquellos que sean copias, ofreciendo opciones para la resolución de la circunstancia.

1.2. Elección del lenguaje

Para esto, he decidido utilizar el lenguaje C++, ya que, como descendiente directo de C, es un lenguaje que tiene más acceso a las funciones del sistema y es un lenguaje compilado, lo que le hace más rápido.

He elegido C++ en lugar de C por la orientación a objetos, que en este caso resulta de utilidad. Además, con la orientación a objetos se pretende que se pueda modificar el programa para utilizarlo modularmente con otros programas que puedan requerir esta utilidad.

Como IDE para controlar los errores y compilar he elegido Code::Blocks, porque es sencillo y rápido. Para edición de texto, he preferido Visual Studio Code que, aunque es algo lento y no compila muy bien, tiene buenas funciones de predicción de texto y temas que facilitan ver qué tramos abarca cada bloque de código.

1.3. Distribución del tiempo

Para que este programa funcione, lo más importante es que lea un directorio, recoja los archivos que tiene y sea capaz de comparar esos archivos con más o menos detalle hasta determinar si son duplicados.

Esta práctica tiene una duración de 80 horas, por lo que, después de utilizar los primeros días para la planificación y la documentación, me centré en que el programa tuviera estas funcionalidades y no tuviera problemas graves.

Para ayudar con la distribución del tiempo y la concreción de las tareas, he utilizado el programa Notion de gestión de tareas y notas (sugerido por mi compañero Jonatan).

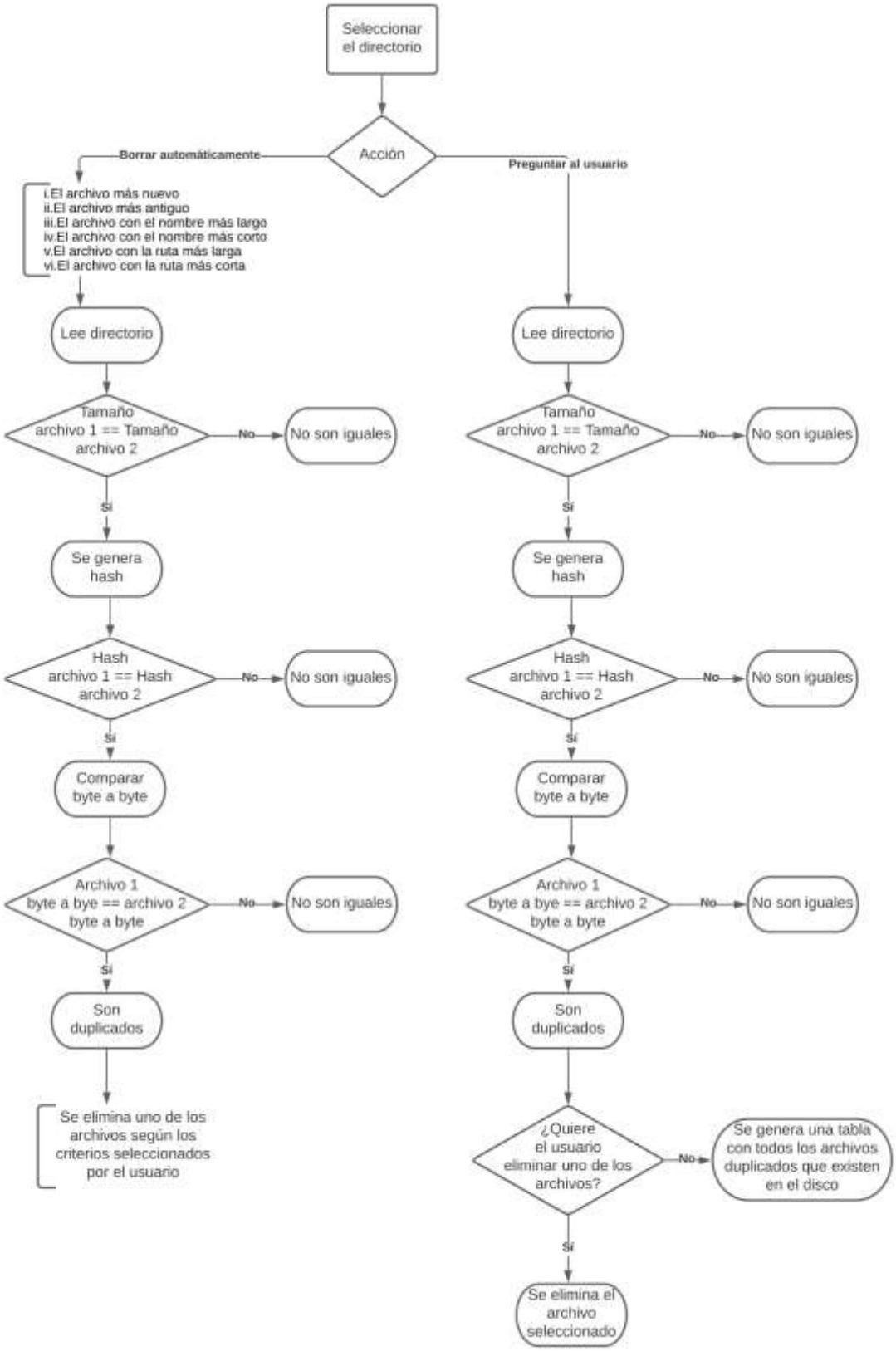
Hay algunas funcionalidades extra que me habría gustado poder añadir, como leer los archivos que hay en subdirectorios dentro del directorio, poder leer directorios cuya ruta incluya caracteres especiales, o comparar los archivos de dos carpetas diferentes, pero, como al hacer un cuenco, lo más importante es que no tenga fugas y, una vez conseguido eso, se puede pasar a los detalles. Al solo tener 80 horas, la funcionalidad básica es lo único que he podido hacer totalmente.

2. Diseño

2.1. Diagrama de flujo

Para crear el diagrama de flujo inicial en el que me he basado para la práctica, me fijé en el funcionamiento de un programa llamado CloneSpy. A partir de éste y aplicándolo al tipo de programa que quería hacer, hice este diagrama de flujo:

Nombre: Adrián Fernández Ortega
DNI: 02566277Y



3. Documentación del proceso

3.1. Creación del proyecto

Para la creación de un proyecto de C++, simplemente hay que crear un nuevo documento desde el IDE elegido y guardarlo con la extensión .cpp. C++ no requiere más archivos que el propio .cpp y, al compilar, genera un archivo .o con el código compilado, y un ejecutable .exe (en el caso de Windows, la extensión del ejecutable dependerá del sistema operativo).

3.2. Ruta

El primer paso del programa es pedir al usuario que seleccione una ruta donde hacer el control de duplicados y, a partir de la ruta que dé el usuario, leer todos los archivos que haya en ese directorio.

En C++ tenemos las funciones del archivo de cabecera dirent.h. Con estas funciones podemos abrir el directorio (opendir), leerlo (readdir) y, una vez hayamos terminado, cerrarlo (closedir).

Para este tipo de funciones es importante tener un control, en mi caso mediante un bucle if, de si la función se ejecuta correctamente. Esta función no admite directorios con caracteres especiales como tildes o eñes, así que en el else de este if se le indica al usuario.

La función readdir devuelve un puntero a un objeto de tipo struct dirent. Este objeto tiene un atributo llamado d_name, que almacena el nombre del archivo que se está leyendo. Este atributo es el que nos interesa.

3.3. Archivos

Para poder gestionar los archivos y, llegado el caso, borrarlos, es necesario poder referenciarlos.

Si los únicos datos que tuviéramos de los archivos fueran el nombre y el tamaño, sería factible introducirlos en un array bidimensional y referenciar ese array cuando queramos referirnos a cada archivo.

Sin embargo, al seguir la creación del programa, se hace evidente que queremos y necesitamos más datos de cada archivo. Por lo tanto, la forma más lógica de organizarlo será crear una clase con todos los atributos que queremos del archivo e instanciarla con cada archivo.

Estos objetos se podrían crear con nombres únicos individualmente, pero, ya que no sabemos cuántos objetos serán necesarios (si 3 ó 300), lo más lógico es crear estos objetos con un bucle. Pero, para guardarlos, ya que todos los objetos se crean con el mismo nombre de referencia, necesitamos un array que guarde punteros a cada objeto. En este caso, creamos un vector, que tiene una funcionalidad parecida al array, pero el array no permite guardar objetos.

Los atributos que se obtienen de cada objeto son: nombre (el nombre del archivo tal y como aparece en el sistema), tam (tamaño en bytes), nombreRuta (que junta el nombre de la ruta con el del archivo para crear una ruta absoluta), tipo (para diferenciar los elementos que son carpetas de los que son archivos), hash (que almacena el hash que se calcula cuando dos archivos coinciden), duplicado (un valor booleano que será true si el archivo está duplicado) y modificado (que guarda la última fecha de modificación, útil si el usuario decide usar la fecha de modificación como criterio de eliminación).

3.4. Tamaño y fecha de modificación

La función readdir proporciona el nombre del archivo, pero hacen falta otros datos que no proporciona. La primera comparación que se quiere hacer es el tamaño de los archivos. Si dos archivos tienen el mismo tamaño, puede que sean duplicados, pero si tienen distinto tamaño es imposible que sean duplicados.

La función stat da el tamaño del archivo en bytes dándole el nombre del archivo con la ruta absoluta. Además, stat nos da la fecha de modificación y el tipo del archivo.

Asignar el atributo de tipo es útil para que las carpetas no se incluyan en las comparaciones.

Una vez tenemos el tamaño, se comparan los archivos. Si dos archivos tienen el mismo tamaño, se consideran sospechosos de ser duplicados y se pasa a generar su hash.

3.5. Hash

Las funciones hash son algoritmos que, dado un archivo o una variable de cualquier tamaño, genera un identificador.

En este caso, he elegido el hash MD5, que genera un identificador de 32 símbolos hexadecimales.

Una función hash puede coincidir en dos archivos que sean diferentes (aunque las probabilidades son muy bajas), pero dos archivos que sean iguales no pueden tener un hash diferente.

Pude encontrar una implementación del hash MD5 en internet que, creando un archivo de cabecera .h e incluyéndolo en el proyecto con «`#include “md5.h”`», genera los hashes MD5 fácilmente.

Por lo tanto, tras obtener el hash de los archivos sospechosos, se comparan y, en caso de que coincidan, siguen siendo sospechosos y se pasa a compararlos byte a byte.

3.6. Comparación de archivos

Si dos archivos tienen el mismo tamaño y el mismo hash la posibilidad de que sean distintos es muy pequeña, pero existe. Por lo tanto, antes de pasar a borrar uno de los dos archivos, hay que asegurarse de que los archivos son exactamente iguales, byte a byte.

El motivo por el que se pasa por estos tres niveles de comparación es el tiempo. Comparar el tamaño de dos archivos es una tarea muy sencilla y rápida, aunque los archivos sean de varios GB. Generar el hash puede llevar varios segundos, más cuando más largo sea el archivo. Pero lo que más tiempo llevará será ir leyendo el archivo línea a línea y comparándolo con el otro archivo. Comparar todos los archivos byte a byte sería un gasto innecesario de recursos y de tiempo.

Para esta comparación he utilizado la función fgets. Esta función tiene un parámetro que son los bytes que quieres recoger. Recoge ese número de byes o hasta el final de la línea, lo que sea antes. Estas líneas las guardo en unas variables contenido1 (para el primer archivo) y contenido2 (para el segundo archivo). De esta forma, el programa va comparando línea a línea y asignando cada línea a esas variables. De otra forma, se generaría una copia del archivo dentro del programa para poder compararlo, lo que supondría un gasto enorme de memoria.

En esta comparación, si una línea es distinta en los archivos, significa que no son duplicados, por lo que se puede descartar. Si línea a línea todas son iguales, entonces se puede confirmar que los archivos son duplicados.

3.7. Gestión de los duplicados

En mi caso he decidido preguntar cómo se van a gestionar los archivos al principio del programa. Se le dan dos opciones al usuario: borrar uno de los archivos duplicados automáticamente siguiendo un criterio o poder elegir una vez se encuentren los duplicados.

En caso de que el usuario seleccione el borrado automático, puede elegir el criterio con el que se borrarán (el más reciente o el más antiguo; el que tenga el nombre más largo o más corto; el que tenga la ruta más larga o más corta). Una vez elegido, ya no se preguntará nada al usuario hasta llegar al final del programa. Las opciones de ruta más larga o más corta son virtualmente iguales que las de nombre más largo o más corto en este caso, porque no se ha llegado a añadir la funcionalidad de leer subdirectorios, por lo que todos los archivos que lea el programa tendrán la misma ruta. He decidido conservar estos criterios para facilitar la posibilidad de añadir esa funcionalidad en un futuro.

En caso de que seleccione poder elegir, una vez que se encuentre un duplicado se le mostrarán los nombres de los archivos y podrá elegir borrar uno de los dos o no borrar ninguno.

Borrar un archivo del sistema es bastante fácil en C++, con la función remove() de C. La única dificultad es convertir el nombre de la ruta del archivo a borrar a cadena de C. Si esta función se ejecuta correctamente, devuelve un 0, lo que es útil para utilizar un if else como comprobación de que la eliminación ha funcionado.

3.8. Salir del programa

Una vez no se encuentran más duplicados, el programa termina. En algunos casos, los programas de consola se cierran una vez terminados, por lo que he añadido un cin al final para que el usuario tenga que introducir algún carácter y evitar que el programa se cierre sin que el usuario pueda leer los resultados.

3.9. Encapsulación

Para seguir el paradigma de la orientación a objetos haciendo que los atributos del objeto sean privados, se generan métodos para obtener y para cambiar cada atributo desde fuera del objeto (conocidos como getter y setter). De esta forma, en lugar de acceder al nombre de un archivo con `vectorArchivos[i].nombre`, es necesario acceder con `vectorArchivos[i].getNombre()`, lo que supone menos posibilidades de cambiar los atributos por error.

El siguiente paso en la orientación a objetos sería dividir el funcionamiento del programa en funciones externas al método main, que puedan ser reutilizadas en otros programas (modularidad). Por falta de tiempo, no se ha hecho esta parte.

3.10. Testeo

Antes de dar el programa por terminado, he realizado varias pruebas buscando los fallos en funcionalidad y posibles errores que el usuario pueda cometer. En primer lugar, he intentado que cualquier bloque de código que deba ser ejecutado, pero que pueda fallar, tenga un else que imprima en pantalla dónde ha fallado. Con este else, pude ver que cuando introduces una ruta que contenga caracteres especiales como tildes, la función de readdir no se ejecuta.

También he probado el programa en otros sistemas operativos. El ordenador que he utilizado para programar el programa tiene Windows 8.1, así que he probado la aplicación en un ordenador con Windows 10 (que es más común) y también lo he probado en Linux (aunque para esto he tenido que compilarlo dentro de Linux, ya que el ejecutable de Windows no funcionará en Linux). Para Linux la única modificación que he tenido que hacer es que después

de la ruta se añada una barra inclinada (/) en lugar de la barra invertida (\) que se añade en Windows.

Otras pruebas que he hecho han sido usar diferentes rutas: rutas muy largas, nombres de archivos muy largos, o introducir la ruta con barras inclinadas (/) en lugar de barras invertidas (\).

4. Problemas y margen de mejoras

Este programa genera algunos problemas.

En primer lugar, la función readdir no admite caracteres especiales, pero Windows sí. Por lo que introducir cualquier ruta que tenga tildes o eñes dará un error. Para que se puedan admitir sería necesario crear una función con la funcionalidad de readdir desde cero, lo que sería inviable en el margen de tiempo del que disponemos.

Otro problema que tiene es que, al comparar los archivos con un bucle, si tenemos el archivo archivo1 y el archivo archivo2 que son duplicados, al hacer los bucles muestra al usuario que ha encontrado que “archivo1 y archivo2 son duplicados” y a continuación muestra que ha encontrado que “archivo2 y archivo1 son duplicados”. Esto puede hacer que el usuario elimine uno de los archivos y, seguidamente, pensando que son archivos diferentes, borre el otro (en caso de que se intente borrar el mismo archivo ya borrado, dará error).

Una funcionalidad que no he conseguido terminar pero que he intentado desarrollar es la lectura de subcarpetas. Al leer un directorio y encontrar un archivo que sea de tipo directorio, se abriría y se leerían los archivos que contiene (y se haría recursivamente si esa carpeta contiene a su vez subcarpetas).

Otra funcionalidad sería comparar los archivos de una carpeta con los archivos de otra carpeta distinta (en lugar de compararlos con los archivos de su misma carpeta). Esto serviría para gestionar copias de seguridad (si ya hay un archivo idéntico en la copia de seguridad, no hace falta copiarlo).

Además, este programa funciona en la consola. Es posible crear programas de C++ con interfaz gráfica y esto sería deseable para que la presentación del programa sea más accesible a los usuarios.

Por último, este programa se ha creado únicamente con un objeto (con atributos y métodos getters y setters) y una función main. Para cumplir con los objetivos principales de la programación orientada a objetos y tener un programa reutilizable sería necesario reorganizar el programa de forma que cada acción tenga su función y de forma que se pueda reutilizar para futuros proyectos.

La encapsulación de la programación orientada a objetos pide que los objetos sean lo más seguros posible, por lo que lo ideal sería que existan métodos que devuelvan cada atributo que tiene el objeto y métodos que modifiquen dichos atributos, de forma que el programa que no tenga acceso directo a los atributos.

5. Entrega

Para la entrega, tengo los siguientes archivos:

5.1. PNL IFCT0609 Documentación.pdf

Este archivo de documentación del proceso de programación

5.2. PNL – diagrama.pdf

Incluyo el diagrama de flujo de este documento en un PDF a parte para que no se pierda calidad al ampliarlo.

5.3. main.1.4.cpp

Código fuente del programa, versión 1.4 del mismo.

5.4. md5.h

Archivo de cabecera que contiene el código para generar códigos hash. Fuente: https://bobobobo.wordpress.com/2010/10/17/md5-c-implementation/

5.5. main.1.4.o

Código compilado del programa, generado por el compilador.

5.6. main.1.4.8.exe

Ejecutable del programa, versión para Windows, generado en un equipo con Windows 8.1.

5.7. main.1.4.10.exe

Ejecutable del programa, versión para Windows, generado en un equipo con Windows 10. Esta versión funciona en Windows 8.1, pero la versión de Windows 8.1 no funciona en Windows 10.

5.8. main

Ejecutable del programa, versión para Linux. Generado en una máquina virtual con Xubuntu 20.04.2.0.

5.9. main.1.5.cpp

Código fuente del programa, versión 1.5 con una posible mejora que habría que terminar de implementar: este código lee los archivos que hay en un directorio y, si alguno es una carpeta, la abre y lee los archivos que ésta contiene. Si la carpeta que se lee tiene más de una subcarpeta, el programa deja de responder. Si la capeta tiene solo una subcarpeta, el programa se ejecuta normalmente.

5.10. main.1.5.exe

Ejecutable de la versión 1.5 del programa, con los problemas mencionados anteriormente.

6. Conclusiones

A pesar de que el programa tiene ciertos fallos y tiene algunas funcionalidades que se podrían añadir en un futuro, lo importante es que realiza la función pedida: es capaz de detectar si dos archivos son iguales y borrar uno de ellos.

Resulta también interesante pensar que este certificado de profesionalidad empezó hace 8 meses, cuando no sabía programar en ningún lenguaje, y ahora he podido generar un archivo ejecutable que tiene una funcionalidad que es tan interesante que existen programas comerciales que la realizan.

Además, existe un gran margen de mejoras para este proyecto, que podré seguir desarrollando una vez acabado el módulo.

Este programa me ha hecho aplicar funciones que ya conocía (como las funciones para abrir y manipular un archivo), aprender algunos otros (como las funciones de stat y readdir), aplicar conocimientos que no había puesto en práctica (como las funciones getter y setter) y pensar en lo que quiero seguir aprendiendo y aplicando para poder mejorarlo aún más.