Programación Orientada a Objetos. Curso 2024-25
PRÁCTICA 2. C++: std::vector, plantillas, iteradores, auto, range-for y excepciones. Organizando el proyecto con CMake.

Copia el contenido del directorio poo/p1 de la práctica 1 en un nuevo directorio poo/p2 en el que trabajaremos en esta práctica.

1.- Organizando el proyecto en carpetas y CMake

El código fuente de un proyecto software no suele ir todo en un único directorio. Debemos tener el proyecto bien organizado utilizando una buena estructura de directorios.
Para el código fuente se utiliza normalmente el directorio ‘src’, y ya hemos visto en la práctica 1 que el resultado de la compilación debe ir en el directorio ‘build’. Por tanto, crea el directorio poo/p2/src y poo/p2/build.
En el directorio 'src' iremos incorporando los distintos módulos de nuestro proyecto, cada módulo en un directorio diferente. Vamos a considerar la clase Person nuestro primer módulo, y vamos a crear para este módulo el directorio poo/p2/src/person. Mueve dentro de este directorio los ficheros person.h y person-main.cc que tienes de la práctica 1.
Cuando tenemos varios directorio debemos informar de ellos a CMake, puesto que cada directorio de nuestro proyecto debe tener un CMakeLists,txt en el que declaremos lo que queremos que haga CMake con dicho directorio.
Lo primero es crear el top-level CMakeLists.txt, es decir el que pondremos en el directorio raíz o top-level del proyecto, en nuestro caso el directorio poo/p2 será ahora el *top-level directory* de nuestro proyecto de esta práctica. Para ello vamos a copiar en este directorio el fichero CMakeLists.txt de la práctica anterior y le vamos a quitar la línea:

```
add_executable(person-main person-main.cc)
```

ya que en este directorio ya no se encuentra el archivo person-main.cc, lo hemos movido a el directorio p2/src/person y será allí donde lo declaremos.

Una vez borrada la línea mencionada, añadimos la siguiente línea:

```
add_subdirectory(src)
```

ya que en este directorio (poo/p2) no hay que compilar nada simplemente le indicamos a CMake que nuestro proyecto consta de un subdirectorio, en este caso el directorio 'src',

Ahora tenemos que crear un CMakeLists.txt dentro del directorio poo/p2/src que como tampoco tiene ningún fichero que compilar simplemente tendrá la línea:

```
add_subdirectory(person)
```

Ahora sí, en el directorio poo/p2/src/person tendremos que poner un fichero CMakeLists.txt que incluya las instrucciones para compilar los ficheros `person.h,` el fichero `person-main,cc` y el fichero `person.cc` que añadiremos en el siguiente ejercicio.
Pero antes vamos a añadir algunas funciones a la clase Person.

2.- STL std::vector

C++ dispone de una extensa y prestigiosa librería estándar considerada una obra maestra de la ingeniería informática con infinidad de clases y algoritmos. Dicha librería se llama *Standard*

Template Library (STL). En este ejercicio vamos a usar los nuevos vectores que incorpora C++ dentro de la STL.
Un vector en cualquier lenguaje de programación se utiliza para almacenar series de elementos del mismo tipo de forma contigua en memoria. El hecho de que estén almacenados en la memoria de forma contigua hace que sea muy eficiente y muy rápido en los accesos secuenciales y aleatorios.
En C++ podemos seguir utilizando los vectores de siempre del lenguaje C:

```
int v[10]; // Esto es un vector de 10 enteros en C
```

Son simples y muy eficientes. Pero C++ dispone de una **plantilla** (ahora veremos lo que es esto) denominada `std::vector` que permite declarar y manipular vectores de una forma mucho más sencilla, con muchas funciones adicionales que facilitan su manipulación y también siendo muy eficientes.

Para declarar un vector en C++ podemos hacer:

```
std::vector<int> v;
```

'v' es un vector de enteros y es opcional declarar su tamaño ya que C++ se encargará de ir expandiendo o liberando espacio del vector automáticamente según lo vayamos usando. Para utilizar `std::vector` debemos incluir:

```
#include <vector>
```

Si te fijas el tipo base del vector es 'int', y se indica entre los símbolos '<' y '>'. Cuando en una clase, en este caso la clase vector, se puede indicar el tipo base de esta forma, en vez de clase se usa la denominación **template o plantilla**. Por tanto `std::vector` es una plantilla. De ahí que la STL se llama en realidad biblioteca estándar de plantillas (Standard Template Library). Durante el curso estudiaremos en más profundidad las plantillas.

En vez de 'int' podríamos haber usado cualquier tipo base o clase que quisiéramos para declarar un vector, tanto los tipos propios de C++ como nuestros propios tipos y clases. Podríamos haber declarado distintos vectores así:

```
std::vector<float> v; // vector de floats
std::vector<double> v; // vector de doubles
std::vector<Person> v; // vector de personas
```

Vamos añadir a la clase Person un vector en el que vamos a guardar las preferencias o intereses de cada persona como: deportes favoritos, pasatiempos, géneros musicales, comida, etc., para personalizar recomendaciones. Cada preferencia serán de tipo std::string.
Añade un vector a la parte privada de clase Person que se llame preferences_ y las siguientes funciones:

- GetPreferences() retorna el vector preferences_.
- AddPreference() que recibe como parámetro una cadena (std::string) y la añade al final del vector. Para ello añadir al final del vector usa la función:

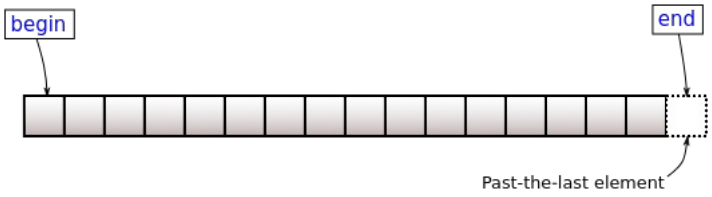
  ```
  preferences_.push_back(s1); // siendo 's1' un string
  ```

- AddPreferences() que recibe 3 parámetros de tipo std::string. El primero se añadirá al principio del vector, el segundo en medio y el tercero al final. Cuando se inserta en medio,

se obtiene la posición de inserción redondeando a la baja la expresión size/2.
Estas inserciones las haremos con llamadas a la función insert() de std::vector que recibe como primer parámetro la posición del elemento a insertar y como segundo parámetro el elemento a insertar. Por ejemplo para insertar al principio, en medio y al final del vector la cadena 's':

```
preferences_.insert(preferences_.begin(), s1);
preferences_.insert(preferences_.end(), s3);
preferences_.insert(preferences_.begin()+(preferences_.size()/2), s2);
```

NOTA IMPORTANTE:
En C++ para indicar una posición dentro de una estructura de datos se usa el concepto de **iterador** (iterator) que es una generalización del concepto de puntero pero de forma que pueda usarse para recorrer cualquier estructura de datos de la STL.
Podemos usar iteradores con nuestro vector preferences_, de forma que preferences_.begin() apunta al primer elemento del vector y preferences_.end() apunta al final del vector como en la siguiente figura:



Fíjate que la función preferences_.size() devuelve el tamaño del vector.

Mira la documentación de std::vector en la URL:
https://en.cppreference.com/w/cpp/container/vector

• ShowPreferences() que muestra en pantalla todas las preferencias de una persona cada una en una línea diferente.
Los iteradores nos facilitan mucho recorrer una estructura de datos de la STL. Por ejemplo, en nuestro caso podríamos usar lo que se denomina **range-for** en C++. Y la instrucción **auto** que declara el tipo de la variable al más adecuado en cada momento, en este caso a un iterador para el vector:

```
void ShowPreferences(){
   std::cout << "\nLas preferencias son: " << std::endl;
   for (auto s: preferences_){
        std::cout << s << std::endl;
   }
}
```

Las funciones que son muy breves (1-2 líneas máximo) se pueden poner directamente en el fichero person.h (son **funciones inline**). Pero la función ShowPreferences() anterior y otras que vamos a añadir a partir de ahora son más complejas y su cuerpo deben ir en el fichero `person.cc` de modo que para las funciones más largas pondremos su prototipo en el fichero `person.h` y su cuerpo en el fichero `person.cc`.

En el fichero .h únicamente estarán la declaración de la clase con sus datos, las funciones inline y los prototipos de las funciones largas.
Para escribir el código de una función miembro de una clase en el fichero `person.cc` hay que poner delante de su nombre el nombre de la clase a la que pertenece. Por ejemplo, la función AddPreferences() se declara en el fichero `person.cc` de la siguiente manera:

```
void Person::AddPreferences(std::string s1, std::string s2, std::string s3){

. . . aquí irá su código . . .

}
```

En el fichero `person.h` únicamente añadiremos a la parte pública de la clase Person:

```
      void AddPreferences(std::string s1, std::string s3, std::string s3);
```

A partir de ahora, haz siempre lo mismo con todas las funciones largas de todas las clases que hagamos.

Añade al final del programa principal en person-main.cc una llamada a AddPreferences() pasandole 3 preferencias + una llamada a ShowPrefeferences() + otra llamada a AddPreferences() con otras 3 preferencias diferentes + una última llamada a ShowPreferences(). Añade 2-3 preferencias más al final con AddPreference() v vuelve a mostrar todas.

Para compilar ahora nuestro módulo 'person' tenemos que crear un fichero CMakeLists.txt en el directorio poo/p2/src/person con las siguientes instrucciones:

```
1     add_library(person person.cc person.h)
2     target_include_directories(person PUBLIC ${CMAKE_CURRENT_LIST_DIR})
3
4     add_executable(person-main person-main.cc)
5     target_link_libraries(person-main PUBLIC person)
```

- El primer parámetro de una instrucción CMake suele ser lo que se denomina **'target'**. En estas instrucciones son targets person y person-main.
- La línea 1 es necesaria para informar a CMake de que en este directorio hay un módulo o librería que se va a llamar 'person' con los ficheros que lo componen.
- La línea 2 es para indicar que aquellos programas que vayan a usar la librería 'person' tendrán que buscar sus includes en este directorio (la variable `${CMAKE_CURRENT_LIST_DIR}` indica el directorio actual).
- Las líneas 4 y 5 son para generar el ejecutable person-main. Por un lado la línea 4 indica a CMake que tenemos un ejecutable y la línea 5 indica a CMake que dicho ejecutable necesita la librería 'person'.
- Las palabras PUBLIC indican que dichas declaraciones son para ese 'target' y para cualquiera que a su vez use dicho target.

Una vez hecho esto, desde el directorio build ejecutar cmake .. y compilar mediante el makefile generado para posteriormente ejecutar y probar el programa person-main que ahora se encontrará en el directorio p2/build/src/person.

3.- Más funcionalidad para la clase Person. Manejo de excepciones

Las excepciones se producen durante la ejecución de un programa consecuencia de una situación

problemática ‘excepcional’ o de error que puede provocar un comportamiento inesperado del mismo, algo totalmente indeseado en programación. Cuando ocurre una de estas excepciones debemos poner en nuestro código un **bloque try-catch** para su gestión.

Un ejemplo de excepción en un programa es cuando se da una división por cero. Otro caso es acceder a un elemento de un vector en una posición mayor que el tamaño del vector. En este caso accedería a una posición de memoria errónea con resultado indeterminado que puede causar muchos problemas.
Los lenguajes de programación modernos traen **MANEJO DE EXCEPCIONES** y debemos familiarizarnos con él ya que es fundamental para una programación segura y de calidad.
En nuestro caso, vamos a añadir una función que cambie el valor del vector de preferencias. Esto lo haremos con la función ‘at’ de std::vector. Dicha función lanza una excepción cuando se intenta acceder al vector fuera de rango (out_of_range).

Entonces la incorporamos en un bloque try-catch de la siguiente manera:

```
try
{
    preferences_.at(555) = “surf”;
}
catch (std::out_of_range const& exc)
{
    std::cout << exc.what() << '\n';
}
```

Bloques try-catch:

- El bloque try{} contiene el **código a monitorizar**, es decir, el código que puede contener el error y, por tanto, puede lanzar una excepción
- El bloque catch(){} captura la ejecución del programa si se produce la excepción del tipo indicado entre paréntesis. Los datos de la excepción irán en este caso en la variable ‘exc’ y accedemos a dichos datos con la función what(). A la variable ‘exc’ podríamos darle cualquier nombre válido.

Añade este código (modificando lo necesario) a una función de la clase Person que se llame ChangePreference() que recibe como primer parámetro un entero con la posición del elemento a cambiar en el vector y como segundo parámetro la cadena con el nuevo valor de esa preferencia.

Añade a tu programa person-main.cc lo siguiente:

1. Nos pide por teclado el entero con la posición a cambiar del vector.
2. Nos pide por teclado la cadena con el nuevo valor de la preferencia.
3. Una llamada a la función ChangePreference() con dicha una posición.
4. Una línea final de la función main() que saque en pantalla la cadena “--END--.”
5. Ejecuta el programa con distintos valores para la posición (menores y mayores que el tamaño del vector). Analiza el comportamiento del programa en cada caso.
6. Quita ahora el bloque try-catch y haz las mismas pruebas. Comprueba el comportamiento del programa con y sin bloque try-catch comprobando si se ejecuta la línea final del programa que saca la cadena “END.”. Analiza los resultados y asegúrate de entender cada situación y el papel que juega el control de excepciones.