Universidad de Jaén
Departamento de Informática

Programación Orientada a Objetos

http://tiny.cc/poouja

Curso 2022/2023

José Ramón Balsas Almagro
Ángel Luis García Fernández
Víctor Manuel Rivas Santos
Juan Ruiz de Miras

Práctica 1. Organización de código fuente

Objetivos

- Conocer los diferentes mecanismos de gestión de código aportados por los lenguajes y entornos de programación.
- Saber organizar el código de una aplicación C++ físicamente mediante módulos de código y lógicamente mediante espacios de nombres.
- Comprender y saber gestionar errores mediante excepciones.
- Continuar practicando el paso de parámetros a funciones por valor y por referencia.

Contenidos

Ejercicios

1.- (Para preparar en casa antes de la sesión) Repasa de la asignatura *Fundamentos de Programación* el concepto de vector de elementos, su relación con los punteros y cómo se pasan como parámetros a funciones. Consulta los apuntes de *Programación en C++* de la asignatura de *Fundamentos de Programación* (tema 7, punteros). Puedes revisar también el material disponible sobre manejo de punteros y memoria dinámica en la plataforma de docencia virtual.

2.- Crea un **módulo denominado *vehiculo*** (con sus correspondientes ficheros *vehiculo.h* y *vehiculo.cpp*) y mueve a él tanto la estructura Vehiculo como las funciones de apoyo creadas en la práctica 0. Por supuesto, debes especificar en formato Doxygen cada uno de los ficheros que componen el módulo.

3.- Modifica el **módulo *vehiculo*** anteriormente creado para definir un espacio de nombres llamado ***vehiculos***, en el que se incluyan todas las definiciones tanto de tipos de datos, como de constantes y funciones.

4.- Implementa y especifica en formato Doxygen las siguientes funciones de apoyo adicionales (dentro del módulo anterior y en el **espacio de nombres *vehiculos***). Decide en cada caso si los parámetros deben pasarse por valor o por referencia y si deben establecerse como *const* o no.

- *int rellenarVector(Vehiculo v[], int tamv)*. Rellena un vector pidiendo al usuario que introduzca por teclado los datos de cada uno de los vehículos que lo forman. Nota: utiliza en el código de esta función la función para leer un vehículo por teclado que implementaste en la práctica anterior.
- *void mostrarEnPantalla(Vehiculo v[], int tamv)*. Muestra por pantalla el contenido del vector de vehículos (una línea por cada vehículo). Los vehículos se mostrarán de 5 en 5, haciendo una pausa y esperando a que el usuario pulse una tecla para continuar.
- *int maxPrecio(Vehiculo v[], int tamv)*. Busca y devuelve la posición en el vector del vehículo con precio máximo.
- *int buscarPorMatricula(string matricula, Vehiculo v[], int tamv)*. Busca y devuelve la posición en el vector de un vehículo a partir de su matrícula.

5.- Modifica las siguientes funciones del módulo *vehiculo* para que lancen **excepciones** cuando no puedan realizar la operación solicitada. Siempre que se llame a estas funciones desde el programa principal, deberás capturar dichas excepciones de forma adecuada, informando por pantalla del motivo por el que han ocurrido:

- *leePorTeclado* lanzará excepciones cuando el usuario no introduzca correctamente alguno de los datos del vehículo (recuerda las restricciones de formato que indicamos en la práctica 0). La excepción llevará asociada una descripción del motivo por el que se ha lanzado.
- *rellenarVector*, lanzará una excepción si el valor que se le pasa como tamaño del vector es negativo.
- *buscarPorMatricula*, lanzará una excepción si el vehículo buscado no está en el vector.

6.- Utilizando el ***módulo vehiculo*** anteriormente creado, modifica el programa principal de la siguiente forma:

- Solicita al usuario el tamaño del vector donde se colocarán los datos de los vehículos y créalo en *memoria dinámica*. No olvides liberar los recursos utilizados al final del programa.

- Utiliza las funciones anteriores para rellenar el vector, mostrarlo por la pantalla, mostrar el vehículo con mayor precio y modificar los datos de alguno de los vehículos (tendrás que pedir al usuario la matrícula del vehículo que quiere modificar).

Explicación de los conceptos

Organización de código fuente en entornos de desarrollo

La descomposición modular de un problema se puede realizar a distintos niveles dependiendo de la visión que se tenga del sistema a desarrollar. Podemos definir los siguientes **niveles** de menor a mayor grado de abstracción:

Procedimientos y Funciones
: Nivel básico de la descomposición modular de un problema. El problema se divide en tareas, y se escribe una función o un procedimiento para resolver cada una de ellas.

Módulos o Unidades *(el nombre puede variar dependiendo del lenguaje de programación)*
: Engloban a un conjunto de constantes, definiciones de tipos, variables, funciones y procedimientos relacionados con un aspecto concreto del sistema.

Bibliotecas (*Libraries*)
: Agrupación de módulos de propósito general o específico.

Proyectos
: Agrupación de módulos y bibliotecas que se compilan y enlazan para formar un programa.

Agrupación de Proyectos
: Agrupación de diferentes proyectos que componen un sistema y que comparten elementos en común.

La utilización de cualquiera de estos niveles vendrá determinada por las características del sistema a desarrollar. Cualquier entorno de programación suele disponer de utilidades para la gestión de cada uno de estos niveles.

Módulos de código en C/C++

> **¡¡¡ATENCIÓN!!!**
> Los conceptos de esta sección se ponen en práctica en este video tutorial (utiliza tu cuenta @red.ujaen.es para acceder al mismo). El código fuente está en este enlace.

Un módulo en C/C++ se construye a partir de un fichero que contiene declaraciones de constantes, definiciones de tipos, variables locales al módulo, la implementación de funciones y procedimientos, datos (imágenes, iconos, texto, sonidos, etc.).

Un módulo importante dentro de cualquier programa en C/C++ es el que contiene la función **main**. Esta función será el punto de entrada al programa y será llamada cuando el

programa comience su ejecución. *Sólo puede haber una función **main** por programa.*

Por cada módulo se deberá definir un **archivo de cabecera** para que otros módulos puedan acceder a sus elementos.

Estructura de un módulo

En general, podemos definir la siguiente estructura para un módulo en C/C++. Habitualmente, se guardará en un archivo con extensión .c o .cpp:

```
/*Archivos de cabecera de otros módulos utilizados por el actual*/
#include <...>
...
/*Declaración de constantes y variables locales al módulo*/
const float IVA=21;
const float COMISION=3;
float TAE=4.5;
...
/*Implementación de funciones y procedimientos (incluida la función main si
éste fuese el módulo principal) */
float calculoInteres (float cantidad, int dias) {
      ...
}
...
int main (int argc, char *argv[]) {  //sólo si este es el módulo principal
      ...
}
```

Archivos de cabecera

Para poder acceder a los elementos existentes en un módulo desde el código de otro, necesitamos disponer en el segundo de referencias específicas a los elementos que se utilizarán, cuáles son y cómo se definen. Esta información es necesaria para que el enlazador sea capaz de resolver estas referencias al generar el ejecutable definitivo.

Se puede acceder tanto a variables locales como a funciones de otro módulo, aunque lo normal, por cuestiones de ocultamiento de información, es acceder solo a ciertas funciones que deseen usarse en otros módulos

Para utilizar las funciones y procedimientos desde un módulo se debe conocer en el mismo al menos la sintaxis de su cabecera. Esta aportará información importante al compilador (nombre de la función, parámetros y valor devuelto) de manera que este pueda detectar si se está llamando a una función adecuada y si la llamada está bien construida. Ejemplo de cabecera:

```
float calculoInteres (float cantidad, int dias);
```

Puesto que para cada función de un módulo externo que vaya a utilizarse en otro módulo

habría que disponer de su correspondiente cabecera, es más cómodo colocarlas todas ellas en un fichero que se copiará en aquellos módulos que deseen hacer uso de esas funciones. Por este motivo, a este fichero se le denomina fichero de cabecera (header) que será insertado por el preprocesador en los módulos que lo soliciten. Para ello se utiliza la sentencia del preprocesador ***#include***.

Los archivos de cabecera tienen habitualmente la extensión **.h** (o .hpp) y el mismo nombre del módulo que contiene la implementación de las funciones a las que hace referencia. Por ejemplo, el módulo *ordenacion* está formado por los archivos *ordenacion.cpp* y *ordenacion.h*.

Adicionalmente, los archivos de cabecera suelen contener definiciones de nuevos tipos de datos, macros, constantes, etc. necesarias para utilizar los elementos del módulo. Por ejemplo:

```
#define PI 3.14159
struct Usuario {
    string mail;
    string password;
};
```

Inclusión condicional de ficheros de cabecera

En ocasiones, cuando los ficheros de cabecera se van insertando unos en otros según nuestro proyecto va creciendo, podría darse el caso de que en un mismo módulo se copiara dos o más veces el contenido de un mismo fichero de cabecera. Esto ocasionaría que aparecieran identificadores duplicados que serían detectados como un error por el compilador, generando un error difícil de detectar.

Para evitar la duplicación de inclusiones de un mismo fichero de cabecera, se recurre a utilizar la inclusión condicional de los contenidos de cada fichero de cabecera. Esta consiste en el uso de las directivas de preprocesador: ***#ifndef***, ***#define*** y ***#endif*** para englobar a los contenidos de un fichero de cabecera. Si el citado contenido no ha sido incluido previamente en el módulo, se incluirá por primera vez y se definirá una constante simbólica específica que permitirá saber al preprocesador que esa cabecera ya se ha insertado. Si en el mismo módulo se tratara de volver a insertar el mismo fichero de cabecera, al estar definida dicha constante simbólica, se detectaría esta situación y ya no se repetirían más veces en ese módulo los contenidos de ese fichero de cabecera.

Ejemplo de inclusión condicional en fichero de cabecera modulo.h:

```
/** @file modulo.h*/

#ifndef __MODULO_H
#define __MODULO_H
```

```
     //CONTENIDOS DEL FICHERO CABECERA

#endif
```

A la constante simbólica utilizada para cada módulo se le suele dar un nombre relacionado con el nombre del fichero donde se utiliza, siguiendo un criterio similar al empleado en el ejemplo.

Ejemplo de módulo

Fichero de implementación del módulo (util.cpp):

```
/**@file util.cpp*/
#include <iostream>
#include <fstream>
#include "util.h"

/** @post Escribe en la salida de error estándar (stderr) la cadena
que se pasa como parámetro.*/
void mostrarError(const std::string &mensaje) {
   std::cerr << "ERROR: " << mensaje << endl;
}
/** @post Visualiza un mensaje informativo...*/
void mostrarInfo(const std::string &mensaje) {
   std::cout << "INFO: " << mensaje << endl;
}
/** @post Devuelve true si el fichero existe. false en caso
contrario */
bool existeFichero(const std::string &nombre) {
   bool existe=false;
   std::ifstream ifs(nombre.c_str(),ifstream::in);
   if (ifs) {
      existe = true;
      ifs.close();
   }
   return existe;
}
```

Fichero de cabecera (util.h):

```
/**@file util.h*/
#ifndef __UTIL_H
#define __UTIL_H

   #include <string>
```

```
   void mostrarError(const string &mensaje);
   void mostrarInfo (const string &mensaje);
   int existeFichero(const string &nombre);

#endif
```

Programa principal (main.cpp):

```
/** @file main.cpp*/
#include "util.h"

int main() {
   string nomfich("datos.txt");
   if (existeFichero(nomfich))
      mostrarInfo("El fichero " + nomfich + " existe");
   else
      mostrarError("El fichero " + nomfich + " no existe");
   return 0;
}
```

Espacios de nombres (namespaces)

¡¡¡ATENCIÓN!!! SEGUNDO VIDEO TUTORIAL

Los conceptos de esta sección se ponen en práctica en este video tutorial (utiliza tu cuenta @red.ujaen.es para acceder al mismo). El código fuente está en este enlace.

Hemos visto que los módulos de código son un mecanismo para organizar físicamente el código además de proporcionarnos un primer nivel de ocultamiento de información. Esto es debido a que solo se pueden utilizar aquellos elementos del módulo que expresamente estén declarados en su fichero de cabecera.

Sin embargo, en programas de cierto tamaño, podría ocurrir que varios módulos diferentes utilizaran identificadores similares para referirse a elementos distintos: nombre de funciones, constantes, etc. Si estos módulos se utilizaran en sitios diferentes no habría problema; pero si un módulo intentase incluir los ficheros de cabecera de otros dos donde existe por ejemplo una constante MAXTAM, el compilador generaría un mensaje de error debido a la existencia de un identificador duplicado.

Para que las declaraciones de los módulos permanezcan lógicamente agrupadas y el usuario de los mismos sepa que está refiriéndose a ellas de forma inequívoca, C++ incorpora el concepto de **espacio de nombres** (*namespace*).

Un espacio de nombres define una agrupación lógica de diferentes tipos de elementos: declaración de tipos, constantes, funciones, variables locales, etc. que está referenciada por un identificador común: el nombre del espacio. De esta forma, si un usuario quiere acceder

a algún elemento de un espacio de nombres, debe referirse a él indicando de forma expresa que ese elemento estará declarado dentro de un espacio de nombres en concreto y no en otro. Así evitamos acceder a un elemento que pudiera denominarse de igual forma, pero que estuviera localizado en otro módulo.

Normalmente, los espacios de nombres se declaran en los ficheros de cabecera de los módulos donde se implementan.

Los espacios de nombres se definen como en el siguiente ejemplo:

```
/**@file banca.h*/
...
namespace BancoOnline {
   //DECLARACIONES y/o IMPLEMENTACIONES
   struct Cliente { ... };
   const float comision=1.5;
   enum TDiaSemana { lunes, martes, miercoles, jueves, viernes,
                     sabado, domingo };
   void solicitaPrestamo(float cantidad);
}
```

```
/**@file banca.cpp*/
...
void BancoOnline::solicitaPrestamo(float cantidad) {
  //Aquí estaría la implementación de la función solicitaPrestamo
}
```

Para utilizar un elemento declarado dentro de un espacio de nombres existen varias formas de más a menos restrictivas:

- Referenciando directamente el nombre del espacio de nombres junto al identificador accedido. Por ejemplo:

```
   std::string nombre;
```

- Seleccionando elementos concretos del espacio de nombres que pueden utilizarse sin indicar expresamente el espacio de nombres. Por ejemplo:

```
   using std::string, std::vector
```

- Seleccionando el espacio de nombres para que los identificadores que se utilicen se busquen dentro del mismo. Por ejemplo:

```
   using namespace std;
```

No obstante, si se abusa de la tercera forma "abriendo" todos los posibles espacios de nombres de un módulo, se corre el riesgo de volver a tener identificadores duplicados. Por lo tanto, se recomienda usar esta tercera forma cuando se haga uso únicamente de un solo espacio de nombres y el resto se identificadores de otros espacios se refieran de forma

específica usando los dos métodos más estrictos.

A modo de resumen:

- Abrir espacios de nombres solo en archivos .cpp cuando no haya conflictos con los identificadores.
- NUNCA abrir espacios de nombres en un archivo de cabecera (si se hiciera, todos los módulos que lo incluyeran tendrían abierto ese espacio de nombres, quieran o no).

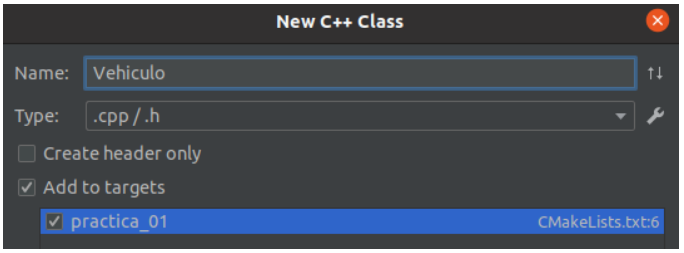
Por último, hay que tener en cuenta que una clase define un espacio de nombres que tiene como identificador el mismo nombre de la clase. El espacio de nombres de una clase se "abre" automáticamente cuando estamos dentro de la implementación de sus métodos y, por lo tanto, no es necesario indicar el nombre de la clase cuando accedemos a un atributo o intentamos llamar a otro método de la propia clase. Sin embargo, si implementamos un método de una clase fuera de su declaración, sí es necesario especificar su espacio de nombres en su cabecera para saber que dicho método pertenece a la clase.

Proyectos

Una aplicación C/C++ puede estar formada por la unión de varios módulos. Para generar el ejecutable habrá que enlazar el resultado de la compilación de todos los módulos. Un **proyecto** es una agrupación de ficheros junto con un documento que relaciona todos los elementos que se deben integrar para generar un ejecutable, biblioteca, etc. A su vez, también proporciona instrucciones a las herramientas que intervienen en el proceso de generación del programa: compilador, ensamblador, enlazador, etc.

El documento que define la forma de procesar el proyecto suele ser un fichero cuyo formato puede depender de la herramienta de desarrollo o ser un fichero estándar en texto plano (*Makefile*). Sin embargo, los ficheros *Makefile* deben adaptarse dependiendo de las características de la plataforma para la que se deseen compilar las aplicaciones, por lo que surgieron utilidades como CMake que se encarga precisamente de automatizar este proceso. CMake utiliza básicamente un fichero de texto más simple, CMakeLists.txt, donde a grandes rasgos se indican los ficheros necesarios para construir la aplicación. El fichero *CMakeLists.txt* se puede editar con un editor de texto simple, aunque normalmente es el propio entorno de desarrollo el que se encarga de adaptar de forma transparente su contenido según los cambios que hagamos en la interfaz gráfica del propio IDE: añadir un nuevo fichero .cpp o .h, renombrar un fichero, borrar un fichero, cambiar el conjunto de herramientas de compilación, cambiar la ubicación de los ficheros, etc.

Por este motivo, cada vez que añadimos un nuevo fichero al proyecto, e.g. desde la opción File→ New, es importante que en el asistente de CLion marquemos la casilla "*Add to targets*" para que el nuevo fichero sea procesado por el entorno:



Cuando queremos compilar un proyecto, el IDE, por tanto, utiliza la herramienta CMake para que se encargue de hacer los ajustes necesarios y, a su vez, delegar en la herramienta *Make*, que irá realizando todos los pasos necesarios y llamando a las diferentes herramientas necesarias.

Puesto que el procedimiento de compilación completo para aplicaciones de cierto tamaño suele ser lento, la herramienta *Make* solo procesa aquellos ficheros que no se hayan modificado desde la última vez que se lanzó.

Los IDEs actuales también permiten que existan proyectos agrupados de forma que compartan ciertos módulos de código y que al compilarse puedan procesarse cada uno de ellos de forma independiente, generándose a la vez múltiples ejecutables y/o bibliotecas

Gestión de errores mediante excepciones

Uno de los principales problemas al desarrollar software es la gestión de las condiciones de error. Esto implica:

a. Detectar que el error ocurre.
b. Identificar dónde y por qué ocurre.
c. Corregirlo.

En general, cuando ocurre un error, el flujo de ejecución normal de la aplicación no puede continuar y el programa debe tratarlo antes de poder continuar o, si no es posible, finalizar el programa de forma controlada. Una primera aproximación para abordar estas situaciones consiste en el uso de los denominados ***códigos de error*,** que permiten, por ejemplo, que una función pueda informar si ha podido realizar un determinado cálculo o, por el contrario, si no ha sido posible, informando así al código que la llamó para que tenga en cuenta que los resultados que espera no están disponibles. Sin embargo, la utilización de estos códigos presenta algunos problemas:

1. **Complican la interfaz de la función**, añadiendo parámetros extra para devolver los códigos o utilizando el resultado de la propia función para devolver este código, lo que, en ocasiones hace poco intuitivo su uso.
2. Obligan al código que llama a la función a **comprobar el código de error devuelto** antes de continuar su funcionamiento, lo que puede complicar su implementación.
3. Si el código que llama a la función que devuelve un error no debe o puede tratar el error, está forzado a su vez a **propagar este código de error** a quien lo llamó, complicando a su vez a funciones de nivel superior.

4. **La interpretación de estos códigos de error puede ser complicada** o variar con el tiempo, lo que afectará a quien tenga que tratarlos, e.g. función que devuelve -1 si el cálculo es incorrecto, -2 si no hay memoria, -3 si los datos de entrada no son válidos, etc.

Las **excepciones** son un mecanismo aportado por los lenguajes de programación para solventar en cierta medida este tipo de problemas. Sus principales objetivos son:

1. **Separar el manejo de errores** del código "normal".
2. **Facilitar la propagación de errores** de la función que los detecta a la función que puede o debe tratarlos.
3. **Diferenciar y clasificar los tipos de errores** para facilitar a quien corresponda cómo identificarlos y, por consiguiente, cómo tratarlos.

Se denomina **excepciones** a aquellas condiciones de errores imprevistos: agotamiento de memoria, ficheros que no existen, errores en rangos de intervalos, etc. Algunas excepciones son generadas por el sistema, pero nuestros programas también pueden generarlas ante un error ocurrido en tiempo de ejecución.

Cuando una función detecta que hay un error que evita que se pueda seguir realizando un determinado procesamiento, se **lanza una excepción**, con lo que termina la ejecución del código de la función y se pasa a un modo de procesamiento de la excepción en el que pueden ocurrir dos situaciones:

1. Nadie ***captura la excepción***, por lo que el programa termina abruptamente (aborta).
2. Alguien ***captura y la maneja adecuadamente***, intentando reconducir la situación sin hacer que el programa termine o al menos informando de qué ha ocurrido y dando los detalles que se tengan.

En C++ se utilizan los siguientes elementos sintácticos para la gestión de excepciones:

1. Se intenta (***try***) ejecutar un bloque de código y se decide qué hacer si se produce una circunstancia excepcional durante su ejecución.
2. Si no se produce la circunstancia, el programa sigue su curso normal; si se produce, se lanza (***throw***) una excepción y se pasa a la etapa siguiente.
3. Si se ha lanzado la excepción, la ejecución del programa es desviada a un bloque de código específico (un manejador o *handler*) donde la excepción es capturada (***catch***) y se decide qué hacer al respecto.

Ejemplo de manejo de excepciones:

```
/** ...
  * @throw string Si hay algún error con el dato introducido por el usuario
  */
void leePosicionVector(int vector[], int tamVector) {
   int pos, valor;
```

```
   cout << "Indique una posición del vector: ";
   cin >> pos;
   //Si detectamos una situación anómala lanzamos la excepción
   //En este caso se informa de quién lanza y el motivo
   if ( pos<0 ){
      throw string("[leePosicionVector]: El valor está por debajo del"
                   " rango");
   }
   if ( pos >=tamVector ){
      throw string("[leePosicionVector]:El valor está por encima del"
                   " rango");
   }
   //Lo siguiente no se ejecuta si se lanza una excepción
   cout << "Indique un valor a asignar: ";
   cin >> valor;
   vector[pos]=valor;
}
```

```
int main(int argc, char** argv) {
   int v[] = { 0, 0 };
   int pos, valor, MAX_TAM=2;
   try {
      leePosicionVector(v,MAX_TAM);
      //Lo siguiente no se ejecuta si se lanza una excepción
      cout << "El valor se ha almacenado correctamente";
  } catch(const string &e) {
      //Capturamos la excepción e informamos
      cout << "El dato no se ha leído: " << e <<endl;
  }
  // Continúa el programa
  return 0;
}
```

Las excepciones que se lanzan pueden ser de cualquier tipo, ya sean tipos simples como *int* o incluso tipos compuestos como *struct* o clases. El tipo de una excepción se utiliza, por un lado para *identificarla* (en la construcción *catch* correspondiente) y para *almacenar la información* asociada a dicha excepción, p.e. un mensaje descriptivo del motivo por el que se ha lanzado, un código de error y/o información adicional para saber más del propio error, como datos concretos que lo produjeron. En definitiva, información que ayude al programador a identificar y corregir el problema que lo ocasionó.

Por ahora utilizaremos excepciones de tipo *int* o *string* para aquellas funciones que deban informar de un error en su procesamiento para ir familiarizándonos con esta nueva forma de gestionar los errores. Más adelante veremos cómo utilizar las características que aporta la programación orientada a objetos para mejorar las ventajas que nos aporta el uso de excepciones.

Bibliografía

- AGUILAR, Luis Joyanes; MARTÍNEZ, Ignacio Zahonero. “Excepciones” En Programación en C, C++, Java y UML. Ed. McGraw-Hill, 2014. Cap. 20.