Universidad de Jaén
Departamento de Informática

v 1.0

José Ramón Balsas Almagro
Ángel Luis García Fernández
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++ mediante módulos de código y mediante espacios de nombres.
- Comprender y saber gestionar errores utilizando excepciones.
- Continuar practicando el paso de parámetros a funciones por valor y por referencia.

Contenidos

Punto de partida

El punto de partida de esta práctica es el código fuente que escribiste para la práctica anterior.

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 *Fundamentos básicos 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 de esta asignatura.

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 anterior. Evidentemente, debes incluir la documentación correspondiente en cada uno de los ficheros que componen el módulo.

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

4.- Implementa y escribe la especificación de 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.

- *void rellenarVector(Vehiculo v[], int tamv)*. Rellenará un vector leyendo desde teclado cada uno de los vehículos que lo forman. Nota: esta función utilizará la función para leer un vehículo por teclado que implementaste en la práctica anterior.
- *void mostrarEnPantalla(Vehiculo v[], int tamv)*. Mostrará por pantalla la lista de vehículos (una línea por cada vehículo). Mostrará los vehículos de 5 en 5, haciendo una pausa y esperando a que se pulse una tecla para continuar.
- *int maxPrecio(Vehiculo v[], int tamv)*. Calculará y devolverá la posición del vehículo de mayor precio.
- *int buscarPorMatricula(string matricula, Vehiculo v[], int tamv)*. Buscará y devolverá la posición que ocupa un vehículo en el vector a partir de su matrícula.

5.- Modifica las siguientes funciones del módulo *vehiculo* para que lancen **excepciones** cuando no puedan realizar la operación solicitada. Captura dichas excepciones de forma adecuada en el programa principal, 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. La excepción llevará asociada una descripción del motivo por el que se ha lanzado (es decir, cuál es el dato mal introducido).
- *rellenarVector* lanzará una excepción si el tamaño del vector que se le pasa como parámetro es negativo.
- *buscarPorMatricula* lanzará una excepción si el vehículo que se busca 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*. ¡Ojo! No olvides liberar los recursos utilizados al final del programa.

- Utiliza las funciones implementadas en ejercicios anteriores para rellenar el vector, mostrarlo por la pantalla, mostrar el vehículo más caro y modificar los datos de alguno de los vehículos (habrá que preguntar al usuario la matrícula del coche que quiere modificar y buscarlo en el vector).

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 cada una de ellas.

Módulos o Unidades
: 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 compilarán y enlazarán 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++

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, desde el código de un módulo, poder acceder a los elementos existentes en otro módulo, necesitamos disponer de referencias específicas a los elementos de ese segundo módulo 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.

Desde el código de un módulo se puede acceder tanto a variables como a funciones de otro módulo. Sin embargo, lo normal, por cuestiones de ocultamiento de información, es que un módulo solo declare como accesibles ciertas funciones (la **parte pública** del módulo) para que puedan usarse en otros módulos.

Para utilizar las funciones y procedimientos de un módulo, el compilador necesita conocer al menos la sintaxis de sus **cabeceras**. Estas aportan información importante: el nombre de la función, el tipo de dato del valor que devuelve y sus parámetros. Sabiendo esto, el compilador puede detectar a qué función se está llamando, y si la llamada es correcta.

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

Puesto que para cada función pública de un módulo que vaya a utilizarse en otro módulo hay que disponer de su correspondiente cabecera, es más cómodo colocarlas todas 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 file), y tenemos que indicarlo en los módulos que necesitemos. Con la directiva del preprocesador ***#include*** hacemos esta indicación. Cuando iniciamos la compilación del programa, una de las primeras cosas que hace el preprocesador es insertar de manera transparente para nosotros el contenido de esos archivos de cabecera en el lugar de las instrucciones #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.

Ejemplo: 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.

Ejemplo:

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

Inclusión condicional de ficheros de cabecera

Cuando un proyecto tiene varios módulos con sus correspondientes ficheros de cabecera, podría darse el caso de que el mismo fichero de cabecera se incluya desde diferentes ubicaciones. Esto ocasionaría que aparecieran **identificadores duplicados** que serían detectados como un error por el compilador, generando un error difícil de encontrar.

Para evitar la duplicación de inclusiones de un mismo 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. Así, si el citado contenido no ha sido incluido anteriormente en el módulo, se incluirá por primera vez y se definirá una constante simbólica específica que permitirá al preprocesador saber 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 repetiría más veces en ese módulo los contenidos de ese fichero de cabecera.

Ejemplo de fichero de cabecera que aplica esta técnica:

```
/** @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"   // La cabecera del módulo se incluye siempre

/** @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 del módulo (util.h):

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

```
   #include <string>

   void mostrarError(const std::string &mensaje);
   void mostrarInfo (const std::string &mensaje);
   bool existeFichero(const std::string &nombre);

#endif
```

Programa principal que utiliza el módulo (main.cpp):

```
/** @file main.cpp*/
#include "util.h"   // Incluye la cabecera del módulo que utiliza

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)

Hemos visto que los módulos son un mecanismo para organizar el código además de proporcionarnos un primer nivel de **ocultamiento de información**. Esto es debido a que desde fuera del módulo solo se pueden utilizar aquellos elementos que expresamente estén declarados en su fichero de cabecera (por ejemplo: si en el archivo de cabecera no está la declaración de una función que sí está implementada en el módulo, esta función permanece oculta para el exterior).

Sin embargo, en programas de cierto tamaño, podría ocurrir que varios módulos diferentes utilizaran identificadores similares para referirse a elementos públicos distintos: nombres de funciones, constantes, etc. Si un módulo intentase incluir los ficheros de cabecera de otros dos en los que existieran, por ejemplo, constantes con el nombre 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

el nombre del espacio en el que se encuentra, además de su identificador. 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*/   // Archivo de cabecera
...
namespace BancoOnline {   // Definición del espacio de nombres
   //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*/   // Implementación del módulo
#include "banca.h"   // Siempre se incluye la cabecera del módulo
...
/* Como la implementación de la función está fuera de la
definición del espacio de nombres, tenemos que añadir el nombre
del espacio al nombre de la función */
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, son estas:

- Referenciando directamente el nombre del espacio de nombres junto al identificador accedido (se utiliza la notación "::" para separar el nombre del espacio del identificador):

```
std::string nombre = "hola";   // Tipo string del espacio std
std::string otroTexto = "adiós";
```

- Seleccionando elementos concretos del espacio de nombres (normalmente al inicio del fichero). Desde que se hace esta selección, se pueden utilizar sus identificadores sin indicar expresamente el espacio de nombres:

```
using std::string, std::vector;   // Solo se escribe una vez en el
                                  // fichero
//...
string nombre = "hola";   // Tipo string definido en el espacio std
string otroTexto = "adiós";
```

- Seleccionando el espacio de nombres completo (normalmente al inicio del fichero), para que los identificadores que se utilicen se busquen dentro del mismo:

```
using namespace std;   // Solo se escribe una vez en el fichero
//...
string nombre = "hola";   // Tipo string definido en el espacio std
string otroTexto = "adiós";
```

No obstante, si se abusa de la tercera forma, "abriendo" todos los posibles espacios de nombres necesarios en un módulo, se corre el riesgo de volver a tener identificadores duplicados. Por lo tanto, se recomienda usar esta tercera forma con un solo espacio de nombres (el más utilizado), y el resto de identificadores de otros espacios se utilicen 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 para saber que dicho método pertenece a la clase, tal y como se muestra en el ejemplo de código anterior.

Proyectos

Una aplicación C/C++ puede estar formada por la unión de varios módulos, y para generar el ejecutable habrá que combinar todos esos módulos. Un **proyecto** es una agrupación de ficheros junto con información de configuración 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 (*Makefile*). Estos ficheros estándar 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.

Cuando queremos compilar un proyecto, el IDE puede procesar directamente la configuración para ir realizando todos los pasos necesarios e ir llamando a las diferentes herramientas auxiliares necesarias, o bien llamar a una utilidad que se encarga de hacerlo. Esta utilidad está disponible en la mayoría de compiladores y tiene el nombre genérico de *make*. Incluso podría ejecutarse esta utilidad desde la línea de comandos para compilar un proyecto completo sin necesidad de abrir el IDE.

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, el flujo de ejecución normal de la aplicación no puede continuar cuando ocurre un error, 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, si no ha sido posible, notificar 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 no 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, p.e. una 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, **lanza una excepción**, con lo que **se detiene** la ejecución del código de la función y se pasa al 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 la excepción 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. Si no hay problemas, el programa sigue su curso normal; si hay un error, se lanza (***throw***) una excepción.
2. Por otra parte, los bloques de código que pueden producir excepciones se encierran en una estructura ***try***. Esto es: se **intenta** ejecutar un bloque de código que puede producir excepciones.
3. Si en algún punto del bloque de código dentro del *try* se ha lanzado una excepción, **la ejecución de ese código se interrumpe**, y el flujo del programa se desvía 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:

```
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 de las anteriores
   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 de dato, 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 por otro 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 los datos concretos que lo produjeron. En definitiva, información que ayude al programador a identificar y corregir el problema que ocasionó la excepción.

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.