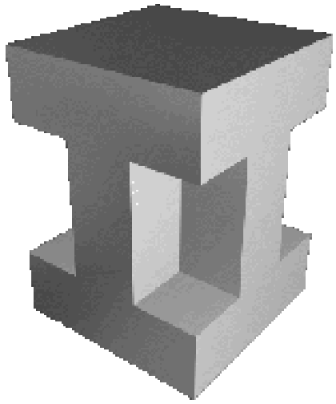
GUÍA DE ESTILO DE C++ (REDUCIDA)



Ingeniería Informática

Seminario de Programación
Departamento de Informática

V.2.0 (11/07/2014)

GUÍA DE ESTILO DE C++

1. Introducción 2
2. Lectura y mantenimiento del código 2
2.1. Encapsulación y ocultación de información 3
2.2. Uso de comentarios 3
2.3. Uso de espacios y líneas en blanco 5
2.4. Elección adecuada de identificadores 6
3. Organización interna 7
3.1. Declaraciones: 7
3.2. Pares de llaves: { } 7
4. Portabilidad y eficiencia 9
4.1. Reglas para mejorar la portabilidad: 9
4.2. Reglas para mejorar la eficiencia: 10
ANEXO A. Documentación de código mediante Doxygen 11
Introducción 11
Doxygen. Documentación básica 11
Configuración de Doxygen 13
5. Programa ejemplo 16

1. Introducción

Este documento tiene como finalidad proporcionar un conjunto de reglas que nos ayuden a escribir programas en C++ con un "buen estilo". Un código escrito con buen estilo es aquel que tiene las siguientes propiedades:

- Está organizado.
- Es fácil de leer.
- Es fácil de mantener.
- Es fácil detectar errores en él.
- Es eficiente.

Hay muchos estilos que cumplen estas características. En este documento simplemente vamos a dar uno de ellos. Este documento es una versión resumida y adaptada de la guía de estilo para C de Juan José Moreno Moll.

2. Lectura y mantenimiento del código

Los principios de esta sección sirven para aumentar la legibilidad del código y para facilitar su mantenimiento. Éstos son:

- Organizar programas utilizando técnicas para encapsular y ocultar información.

- Aumentar la legibilidad usando los espacios y líneas en blanco.
- Añadir comentarios para ayudar a otras personas a entender el programa.
- Utilizar identificadores que ayuden a entender el programa.

2.1. Encapsulación y ocultación de información

La encapsulación y ocultación de información ayudan a organizar mejor el código y evitan el acoplamiento entre funciones del código.

La **encapsulación** permite agrupar elementos afines del programa. Los subprogramas afines se agrupan en ficheros (unidades), y los datos en grupos lógicos (estructuras de datos).

Ocultación de información: Un subprograma *no necesita* saber lo siguiente:

- La fuente de los parámetros que se le pasan como entrada.
- Para que servirán sus salidas.
- Qué subprogramas se activaron antes que él.
- Qué subprogramas se activarán después que él.
- Cómo están implementados internamente otros subprogramas.

Para conseguir esto se deben seguir las siguientes reglas:

- No hacer referencia o modificar variables globales (evitar efectos laterales).
- Declarar las variables y tipos como locales a los subprogramas que los utilizan.
- Si queremos evitar cambios indeseados en parámetros, pasarlos por valor.
- Un procedimiento sólo debe modificar los parámetros pasados en su llamada.

2.2. Uso de comentarios

Los comentarios dan información sobre lo que hace el código en el caso que no sea fácil comprenderlo con una lectura rápida. Se usan para **añadir información** o **para aclarar secciones de código**. No se usan para describir el programa. Por lo tanto no se deben poner comentarios a una sola instrucción. Los comentarios se añaden en los niveles siguientes:

- **Comentario a ficheros:** Al comienzo de cada fichero se añade un prólogo del fichero que explica el propósito del fichero y da otras informaciones.
- **Comentarios a subprogramas**: Al comienzo de cada subprograma se añade un *prólogo del subprograma* que explica el propósito del subprograma.
- **Comentarios dentro del código**: Estos comentarios se añaden junto a la definición de algunas variables (las más importantes), para explicar su propósito, y al comienzo de algunas secciones de código, especialmente complicadas, para explicar que hacen.

Los comentarios se pueden escribir en diferentes estilos dependiendo de su longitud y su propósito. En cualquier caso seguiremos las siguientes **reglas generales**:

- Los **comentarios** en general se escriben **en líneas que no contienen código** y antes del código que queremos clarificar. Esta regla se aplica siempre si el comentario tiene más de una línea.
- **Sólo** en dos casos se permite **poner en la misma línea** un comentario y una instrucción: **comentarios a una definición de variable**, que explica la finalidad de esta variable. Y un **comentario** para indicar **final de una estructura del lenguaje**.

Para **Comentario a ficheros** y **Comentarios a subprogramas** seguiremos las pautas indicadas en el ANEXO A (Documentación en código mediante Doxygen).

Aquí vamos a describir como hacer **comentarios dentro del código**. Dentro de este tipo de comentarios se pueden distinguir:

- Comentarios en cajas: Usados para prólogos o para separar secciones.
- Separador de secciones: Son líneas que sirven para separar secciones, funciones, etc.
- Comentarios para bloques grandes: Se usan al comienzo de bloques de código grandes para describir esa porción de código.
- Comentarios cortos: Se usan para describir datos y casi siempre se escriben en la misma línea donde se define el dato. También se usan para indicar el final de una estructura.
- Comentarios para bloques pequeños: Se escriben para comentar bloques de instrucciones pequeños. Se colocan antes del bloque que comentan y a la misma altura de sangrado.

Comentarios en cajas:

Ejemplo: comentario tipo prólogo en una caja:

```
/************************************************************/
/*      AUTOR: Nombre                                       */
/*                                                          */
/*      PROPÓSITO:                                          */
/*                                                          */
/************************************************************/
```

Separador de secciones:

Ejemplo: Separador de secciones.

```
/************************************************************/
```

Comentarios para bloques grandes:

Ejemplo: comentarios para bloques grandes de código.

```
/*
 * Usar para comentarios a más de un bloque de
 * sentencias.
 */
```

Comentarios cortos:

En caso de utilizar este tipo de comentario, seguir las siguientes reglas:

- Utilizar uno o más tabuladores para separar la instrucción y el comentario.
- Si aparece más de un comentario en un bloque de código o bloque de datos, todos comienzan y terminan a la misma altura de tabulación.

Ejemplo: Comentarios en un bloque de datos.

```
int alu_teoria;       // Numero de alumnos por grupo teoria
int alu_practicas;    // Numero de alumnos por grupo practicas
```

Ejemplo: Comentarios en un bloque de definición de constntes.

```
const float DOL_A_EUR = 0.736903;   // Cambio de dolares a euros 07/OCT/2013
const float SOL_A_EUR = 0.26573129; // Cambio de n.soles a euros 07/OCT/2013
```

Ejemplo: Comentarios al final de una estructura.

```
for (i = 1; i <= FINAL; i++)
{
    while (!correcto)
    {
        correcto = false;
        cout << "\n Introduce dato:";
        cin >> dato;
        if (dato < 100)
            correcto = true;
    } // WHILE
    cout << "\n"
} // FOR
```

Comentarios para bloques pequeños:

Ejemplo:

```
// Hallamos la nota media de todos los examenes
for (i = 0; i < NUM_ALUMNOS; i++)
    suma = suma + nota[i];
media = suma / NUM_ALUMNOS;
```

2.3. Uso de espacios y líneas en blanco

Los espacios en blanco facilitan la lectura y el mantenimiento de los programas. Los espacios en blanco que podemos utilizar son: líneas en blanco, carácter espacio, sangrado.

Línea en blanco:

Se utiliza para separar "párrafos" o secciones del código. Cuando leemos un programa entendemos que un fragmento de código entre dos líneas en blanco forma un conjunto con una cierta relación lógica.

Veamos como separar secciones o párrafos en el programa:

- Las secciones que forman un programa se separan con al menos una línea en blanco (declaración de constantes, declaración de variables, programa principal, …).
- Dentro de un subprograma se separan con una línea en blanco las secciones de declaraciones y el código del subprograma.
- Dentro de un subprograma se separan con una línea en blanco los fragmentos de instrucciones muy relacionadas entre sí (por ejemplo, conjunto de instrucciones que realizan una operación).

Espacio en blanco:

Los espacios en blanco sirven para facilitar la lectura de los elementos que forman una expresión. Los espacios en blanco se utilizan en los casos siguientes:

- Las variables y los operadores de una expresión deben estar separados por un elemento en blanco.
- Las lista de definición de variables, y las listas de parámetros de una función se deben separar por un espacio en blanco.

Ejemplos:

Espaciado correcto: `media = suma / cuenta;`

Espaciado incorrecto: `media=suma/cuenta;`

Espaciado dentro de una lista de parámetros:

```
ConCat(s1, s2, s3);
string ConCat (string s1, string s2, string s3)
```

Sangrado:

El sangrado se utiliza para mostrar la estructura lógica del código. El sangrado óptimo es el formado por **cuatro espacios**. Es un compromiso entre una estructuración legible y la posibilidad de que alguna línea (con varios sangrados) del código supere el ancho de una línea de una hoja de papel o del monitor.

2.4. Elección adecuada de identificadores

Los identificadores que dan nombre a subprogramas, constantes, tipos o variables han de colaborar a la autodocumentación del programa, aportando información sobre el cometido que llevan a cabo. Para elegir nombre se deben seguir las siguientes recomendaciones generales:

- Elegir nombres comprensibles y en relación con la tarea que corresponda al objeto nombrado.
- Seguir un criterio uniforme con las abreviaturas de nombres. Elegir abreviaturas que sugieran el nombre completo.
- Utilizar prefijos y sufijos cuando sea necesario.
- Uso del carácter '_' o de una letra mayúscula para distinguir las palabras que forman un identificador.

Nombres habituales:

Hay algunos nombres cortos que se usan muy habitualmente. El uso de estos nombres (y sólo de estos) es aceptable.

Ejemplo: nombres cortos aceptables.

```
c, ch        caracteres
i, j, k      índices
n            contadores
p, q         punteros
s, cad       cadenas
```

Uso de mayúsculas y minúsculas:

Para reconocer fácilmente la clase de un identificador se utilizan las siguientes normas de utilización de mayúsculas y minúsculas cuando se construyen nombres:

- **Variables**: los nombres de las variables se construyen con palabras (o abreviaturas) minúsculas separadas por el carácter '_'.

  Ejemplo:

  ```
  int num_alumnos;
  ```

- **Nombres de Funciones y Tipos**: los nombres de las funciones y los tipos se escribirán con una o más palabras en minúsculas excepto la primera letra, que es mayúscula (las demás no). No se usa el carácter '_'. La primera letra mayúscula de cada palabra indica la separación entre palabras.

Ejemplo:

```
InsertarAlumno (...)
```

- **Constantes e identificadores del preprocesador**: usan nombres construidos con palabras (o abreviaturas) en mayúsculas separadas por el carácter '_'.

Ejemplo:

```
const int MAX_ALUMNOS = 50;
```

3. Organización interna

En este apartado vamos a describir la forma correcta de escribir ciertas estructuras del lenguaje.

3.1. Declaraciones:

Las diferentes secciones de declaraciones deberán estar claramente diferenciadas y seguir el siguiente orden:

- Directivas **`#include`**
- Declaración de constantes (**`#define`** o **`const`**).
- Declaración de tipos (**`typedef`**).
- Declaración de variables (variables globales).
- Prototipos de funciones

Después de los prototipos irá el código del programa principal (**`main()`**) y a continuación el código del resto de las funciones.

Las declaraciones deben seguir las siguientes reglas:

- Alinear los nombres de las variables de forma que la primera letra del nombre de cada variable esté en la misma columna.
- Definir una variable por línea junto con su comentario (si éste es necesario).

  Como excepción a esta regla podemos agrupar en la misma línea variables auxiliares, contadores, etc.
- Si un grupo de subprogramas utiliza un parámetro o una variable interna para una labor semejante **llamar con el mismo nombre** a esas variables en todas las funciones.
- No utilizar nombres para variables internas que sean iguales a nombres de variables globales.

3.2. Pares de llaves: `{ }`

La forma que tiene C++ de agrupar instrucciones en bloques es utilizar las llaves **`{ }`**. Su colocación se debe hacer en **líneas reservadas para cada una de ellas**, sin ninguna otra instrucción en esa línea. Ambas deben ir en la misma columna que la instrucción de la que dependen.

Ejemplo:

```
for (i = 0; i < N; i++)
{
    vect[i] = vect2[i];
    suma = suma + vect[i];
}
```

Sentencia if:

La sentencia o sentencias que corresponden a la sección *if* o *else* van siempre en una nueva línea sangrada:

```
if (expresión)
    sentencia;
else
    otra_sentencia;
```

```
if (expresión)
{
    sentencia1;
    sentencia2;
}
else
{
    otra_sentencia1;
    otra_sentencia2;
}
```

Sentencia if...else if:

Pueden enlazarse varias sentencias *if...else*. Este enlazado se puede realizar de diferentes maneras:

Siguiendo la idea de diferentes estructuras *if...else*, simplemente puesta una a continuación de la otra:

```
if (expresión)
    sentencia;
else
    if (expresión)
        otra_sentencia;
    else
        otra_sentencia2;
```

```
if (expresión)
{
    sentencia1;
    sentencia2;
}
else
{
    if (expresión)
    {
        sentencia3;
        sentencia4;
    }
    else
    {
        sentencia5;
        sentencia6;
    }
}
```

Utilizando la idea de una sentencia '*else if*' como una sola sentencia:

```
if (expresión)
    sentencia;
else if (expresión)
    otra_sentencia;
else
    otra_sentencia2;
```

```
if (expresión)
{
    sentencia1;
    sentencia2;
}
else if (expresión)
{
        sentencia3;
        sentencia4;
}
else
{
    sentencia5;
    sentencia6;
}
```

Si vamos a enlazar muchas sentencias *if...else*, es recomendable la escritura en la segunda forma.

Sentencia switch:

Los valores asociados a la sentencia *switch* irán en línea aparte sangrada. El bloque de sentencias asociado comenzará en otra línea aparte y también sangrada. Todos los bloques de sentencias pertenecientes al CASE comenzarán en la misma columna.

```
switch (expresión)
{
    case 1:
        sentencia1;
        sentencia2;
        break;
    case 2:
        otra_sentencia;
        break;
    default:
        otra_mas;
}
```

Sentencias de repetición:

La sentencia o sentencias pertenecientes a la estructura de repetición (***for***, ***while*** o ***do ... while***) van siempre en una nueva línea sangrada:

```
while (expresión)
    sentencia;
```

```
while (expresión)
{
    sentencia1;
    sentencia2;
}
```

```
do
    sentencia;
while (expresión);
```

```
do
{
    sentencia1;
    sentencia2;
}
while (expresión);
```

```
for (iniciacion; expresion; incremento)
    sentencia;

for (iniciacion; expresion; incremento)
{
    sentencia1;
    sentencia2;
}
```

4. Portabilidad y eficiencia

Un código bien escrito debe poder ejecutarse en otras máquinas o plataformas haciendo un mínimo de cambios. Nuestro código debe ser lo más portable posible.

4.1. Reglas para mejorar la portabilidad:

- Usar ANSI C++ en la medida de lo posible.
- Escribir código portable en principio. Considerar detalles de optimización sólo cuando sea necesario. Muchas veces la optimización es diferente según la plataforma o la máquina utilizada. En todo caso:

a) documentar estos detalles para poder modificarlos si necesitamos cambiar el código de plataforma.

b) aislar la optimación de otras partes del código.

- Algunos subprogramas son no portables de forma inherente (por ejemplo '*drivers*' de dispositivos suelen ser distintos en sistemas operativos distintos). Aislar este código del resto.
- Organizar el código en ficheros de forma que el código portable esté en ficheros distintos del código no portable.
- Computadoras distintas pueden tener un tamaño de tipos diferente. Utilizar el operador *sizeof()* para asegurar la portabilidad en este caso.
- Evitar las multiplicaciones y divisiones hechas con desplazamientos de bits.
- No reemplazar los subprogramas estándar con subprogramas realizados por nosotros. Si no nos gusta una implementación de un subprograma estándar realizar un subprograma semejante pero con otro nombre.
- Utilizar las directivas de compilación condicional para mejorar la portabilidad del código.

4.2. Reglas para mejorar la eficiencia:

- Recordar que el código debe ser mantenido.
- Si la eficiencia del programa no es crítica sustituir instrucciones rápidas por instrucciones comprensibles.
- Si la eficiencia es importante (sistemas en tiempo real) y es necesario utilizar expresiones no comprensibles y muy complicadas pero rápidas, añadir comentarios que ayuden a comprender el código.
- Reducir al mínimo las operaciones de entrada/salida.
- Usar los operadores: **`++`**, **`--`**, **`+=`**, **`-=`**, etc...
- Liberar memoria tan pronto como sea posible.
- Cuando se pasan estructuras grandes a un subprograma, hacerlo por referencia. Esto evita manipular datos sobre la pila y hace la ejecución más rápida.

ANEXO A. Documentación de código mediante Doxygen

Introducción

El objetivo de este anexo es presentar una de las alternativas existentes para la documentación de código C++, y posterior generación de su API en diferentes formatos (PDF, HTML, RTF, ...).

Concretamente nos centraremos en introducir los aspectos más básicos de Doxygen.

Doxygen es un sistema de documentación para C++, C, Java, Objective-C, IDL (Corba and Microsoft) y extensiones de PHP, C# y D.

Doxygen permite:

- Generar documentación on-line en HTML accesible por navegador y/o manuales off-line de referencia (en Latex), desde un conjunto de ficheros fuente. Además tiene soporte para generar salida en RTF, PostScript, PDF, HTML comprimido y páginas man de UNIX. La documentación es extraida directamente desde el código fuente, lo que permite una consistencia entre ambos.
- Se puede configurar Doxygen para que extraiga la estructura de código no documentado, permitiendo conocer las relaciones entre las diferentes partes del código, y los métodos invocados desde ciertos ficheros.
- Se puede llevar el uso de Doxygen al extremo, y usarlo para generar la documentación completa de ciertos proyectos.

Doxygen. Documentación básica

Existen diferentes estilos de documentación de código mediante Doxygen (javadoc style, qt style, ...) todos ellos correctos. En este anexo se van a presentar los aspectos básicos de la documentación en estilo **javadoc**, que es compatible con los estándares de documentación de código java, pudiéndose ampliar a aspectos más avanzados a partir del manual oficial de Doxygen.

Cabeceras de fichero

Toda pauta de codificación/documentación correcta debe contemplar la asociación a todo fichero de código de una cabecera (que se ubicará al principio del fichero), que describa brevemente dicho fichero, reflejando: fichero al que hace referencia (**`@file`**), descripción del contenido del mismo, fecha de creación (**`@date`**), versión (**`@version`**), autor o autores (**`@author`**). Así mismo, también puede contener otros atributos como: copyright de la empresa a la que pertenece (**`@copyright`**), tareas pendientes de realizar (**`@todo`**), etc...

El siguiente ejemplo ilustra la cabecera mínima a usar para un fichero de prácticas:

¡¡IMPORTANTE!!

```
/**
* @file nombre_fichero.cpp
*
* Prácticas de EDA
* Práctica 3.
*
* @author  Fernando Barber
* @author  Raul Pena
* @date    17/06/2014
* @version 1.0
*/
```

Otro ejemplo:

```
/*************************************************************************/ /**
 *
 * @file Ejemplo_0.cpp
 *
 * Este programa distribuye diferentes donaciones en diferentes monedas entre
 * varios apartados (comedor, centro de salud y administracion)
 *
 * @version 1.0 Version actualizada (05/09/2012)\n
 *          1.1 Actualizacion de los cambios monetarios a fecha 07/10/2013
 *
 * @author Ricardo Ferris, Fernando Barber
 * @date 07/10/2013
 *
 *************************************************************************/
```

Funciones y métodos

Todas las funciones, salvo la función principal, o métodos de clases deberán necesariamente documentarse identificando:

- La funcionalidad o servicio que realizan.
- Las precondiciones (**`@pre`**) que debe cumplir la rutina, si existen.
- Los parámetros (**`@param`**) de entrada y salida de la rutina, asociando a cada uno de ellos una breve descripción.
- Información del valor de retorno de la función (**`@return`**) o el valor o valores de retorno de la función (**`@retval`**), si los tuviera.

El siguiente ejemplo muestra como se documenta una rutina que dispone de varios parámetros de entrada y uno de salida.

¡¡IMPORTANTE!!

```
/**
 *
 * Realiza la división entera, devolviendo cociente y resto
 *
 * @pre Divisor != 0
 *
 * @param x Dividendo
 * @param y Divisor
 * @param resto Resto de la división
 * @return Cociente de la división
 */
int Div(int x, int y, int & resto)
{
    ...
}
```

Opcionalmente, se puede añadir información a los parámetros indicando si son de entrada o de salida:

¡¡IMPORTANTE!!

```
/**
 *
 * Realiza la división entera, devolviendo cociente y resto
 *
 * @pre Divisor != 0
 *
 * @param[in] x Dividendo
 * @param[in] y Divisor
 * @param[out] resto Resto de la división
 * @return Cociente de la división
 */
```

```
int div(int x, int y, int & resto)
{
    ...
}
```

Otro ejemplo de documentación de funciones podría ser el siguiente: ¡¡IMPORTANTE!!

```
/*********************************************************************/ /**
 *
 *  Funcion que pide el nombre del alumno a buscar y lo muestra en pantalla,
 *  si lo encuentra en el vector.
 *
 *  @param [in]  v_alu    Vector donde estan los alumnos
 *  @param [in]  num_alu  Numero de alumnos validos del vector
 *
 *  @retval true  Si hemos encontrado al alumno
 *  @retval false Si no hemos encontrado al alumno
 *
 */ /*********************************************************************/
bool EncontrarAlumno (V_Alumnos v_alu, int num_alu)
{
    ...
}
```

Otros elementos

Se pueden comentar, para que aparezcan en la documentación generada, otros elementos de los programas como: constantes, registros, clases, definiciones de tipo, etc.

En todos estos casos, la documentación seria similar a la de las funciones.

Ejemplo de declaración de constantes

```
/**
 * Numero maximo de alumnos con el que podremos trabajar
 */
const int MAX_ALU = 200;
```

En el caso de las definiciones de constantes y variables, es posible realizar una definición compacta, en la misma linea de la declaración de la contante mediante el calificador **///<**

Ejemplo de declaración de constantes con definición compacta

```
const double KMH2MPS = 1000.0 / 3600;  ///< Factor de conversion km/h a mts/s
```

Ejemplo de declaración de registros

```
/**
 * Informacion basica del alumno
 */
struct Alumno
{
    string nombre; ///< Nombre del alumno.
    int nota;      ///< Nota del alumno.
    int curso;     ///< Curso en el que el alumno esta matriculado.
};
```

Ejemplo de declaración de definición de tipos

```
/**
 * Definicion del vector de alumnos.
 */
typedef Alumno V_Alumnos[MAX_ALU];
```

Configuración de Doxygen

El ejecutable **doxygen** es el principal ejecutable de la herramienta, responsable de parsear los ficheros de código y generar la documentación del mismo.

El ejecutable **doxytag** únicamente es necesario si se desea generar referencias a documentación externa, que no es extraída a partir del código del proyecto.

Por último, el comando **doxywizard** ofrece un front-end gráfico para editar ficheros de configuración de doxygen.

Doxygen emplea un fichero de configuración para especificar todos sus parámetros, de manera que todo proyecto que emplee Doxygen como herramienta de documentación debe disponer de este fichero desarrollado a medida para el proyecto.

Para simplificar la generación del fichero de configuración de Doxygen se puede obtener una plantilla documentada mediante el comando doxygen, pasándole el parámetro -g:

```
doxygen -g <config-file>
```

donde **<config-file>** es el nombre del fichero de configuración a crear.

El fichero de configuración de Doxygen tiene una estructura muy similar a la de un sencillo Makefile. A continuación se presenta un ejemplo, en el cual se establecen una serie de parámetros para generar la documentación de un proyecto denominado *Práctica 5 MP: Algoritmos de Ordenación Rápida*, de entre los cuales podemos destacar:

- La documentación se generará en un directorio relativo a la ubicación del fichero de configuración: **../doc/api**.
- El lenguaje de generación será el español.
- No se utilizarán tags para referencias a documentación externa.
- No se generará documentación en latex, man o rtf.
- El código fuente se encuentra en un directorio relativo: **../src/cpp** y se asocia con los ficheros de extensión cpp o h.
- Se generará la mayor salida posible, sin refinar por tipo de métodos, atributos, ...

Ejemplo: Fichero de configuración de doxygen

```
PROJECT_NAME      = "Titulo ha rellenar"
OUTPUT_DIRECTORY  =
OUTPUT_LANGUAGE   = Spanish
GENERATE_TAGFILE  =
GENERATE_LATEX    = NO
GENERATE_MAN      = NO
GENERATE_RTF      = NO
CASE_SENSE_NAMES  = NO
INPUT             =
FILE_PATTERNS     = *.cpp *.h
EXAMPLE_PATH      =
QUIET             = YES
JAVADOC_AUTOBRIEF = YES
EXTRACT_ALL       = YES
```

```
EXTRACT_PRIVATE   = YES
EXTRACT_STATIC    = YES
```

Una vez instanciado el fichero de configuración según las necesidades del proyecto, para obtener la documentación bastará con ejecutar el siguiente comando:

```
doxygen <config-file>
```

Doxygen genera la documentación en el directorio de salida configurado, organizándola en subdirectorios para cada uno de los formatos seleccionados.

Destacar que únicamente se generará la documentación de aquellas partes del código que hayan sido especificadas de acuerdo a las pautas indicadas.

A través de la utilización de "*DoxyWizard (GUI front-end for creating configuration files)*" también podemos cargar un fichero de configuración previamente definido y generar fácilmente la documentación de nuestros programas.

Para más información podeis consultar la página:

http://www.stack.nl/~dimitri/doxygen/

5. Programa ejemplo

Ejemplo 1:

```
/********* Ejemplo 1 ************************************************/ /**
 *
 * @file Ejemplo_1.cpp
 *
 * Proposito: Este programa transforma velocidades en kilometros por hora
 *            a metros por segundo, millas por hora y nudos.
 *
 * @version 1.1
 * @author Ricardo Ferris
 * @date 07/10/2013
 *
 ************************************************************************/

#include <iostream>
using namespace std;

const double KMH2MPS = 1000.0 / 3600;       ///< Factor de conversion km/h a mts/s
const double KMH2MPH = 1 / 1.609344;        ///< Factor de conversion km/h a millas/h
const double KMH2NUDOS = 0.539956803455723; ///< Factor de conversion km/h a nudos

/*
 * Programa principal
 */
int main (void)
{
    float v_kph;
    double v_mps,
           v_mph,
           v_nudos;

    // Empezamos el programa explicando al usuario lo que hace
    cout << "Este programa transforma velocidades en kilometros por hora ";
    cout << "a metros por segundo, millas por hora y nudos.\n\n";

    // Pedimos al usuario la velocidad en kilometros por hora
    cout << "Dame velocidad: ";
    cin >> v_kph;

    // Pasamos la velocidad original a las diferentes unidades
    v_mps = v_kph * KMH2MPS;
    v_mph = v_kph * KMH2MPH;
    v_nudos = v_kph * KMH2NUDOS;

    // Mostramos los resultados obtenidos
    cout << "La velocidad en kilometros por hora es: " << v_kph << " km/h\n";
    cout << "La velocidad en metros por segundo es: " << v_mps << " m/s" << endl;
    cout << "La velocidad en millas por hora es: " << v_mph << "mph" << endl;
    cout << "La velocidad en nudos es: " << v_nudos << " nudos" << endl;
    cout << endl;

    return 0;
}
```

Ejemplo 2:

```
/********* Ejemplo 2 ************************************************/ /**
 *
 * @file Ejemplo_2.cpp
 *
 * Manejo de notas de alumnos en un array. En este programa trataremos
 * con la lista de alumnos de una clase.
 *
 * @version 0.9
 * @author Fernando Barber
 * @date 12/09/1991
 *
 ************************************************************************/

#include <iostream>
#include <string>
using namespace std;

/**
 * Numero maximo de alumnos con el que podremos trabajar en nuestra
 * aplicacion
 */
const int MAX_ALU = 200;

/**
 * Informacion basica del alumno
 */
struct Alumno
{
    string nombre; ///< Nombre del alumno.
    int nota;      ///< Nota del alumno.
    int curso;     ///< Curso en el que el alumno esta matriculado.
};

/**
 * Definicion del vector de alumnos.
 */
typedef Alumno V_Alumnos[MAX_ALU];

void IntroducirAlumno(Alumno & alu);
void EscribirAlumno(Alumno alu);
void Introducir(V_Alumnos v_alu, int & num_alu);
int PosAlu(string nom_alu, const V_Alumnos v_alu, int num_alu);
bool EncontrarAlumno(V_Alumnos v_alu, int num_alu);

/**
 * Programa Principal.
 * Mostraremos un menu a traves del cual podremos acceder a las
 * diferentes opciones que nos proporciona el programa
 */
int main (void)
{
    V_Alumnos alumnos;
    int num_alu;
    int opcion;

    num_alu = 0;
    do
    {
        cout << "Menu:\n";
        cout << " 1-Introducir alumnos\n";
        cout << " 2-Encontrar un alumno\n";
        cout << " 3-Salir\n";
```

```
        cout << "Introducir opcion:";
        cin >> opcion;
        // Borra el salto de linea del buffer
        cin.ignore();

        switch (opcion)
        {
            case 1:
                Introducir(alumnos, num_alu);
                break;
            case 2:
                EncontrarAlumno(alumnos, num_alu);
                break;
            case 3:
                break;
            default:
                cout << "Opcion introducida incorrecta.\n";
        }
    }
    while (opcion != 3);

    return 0;
}

/************************************************************************/ /**
 *
 *  Pide la informacion de un alumno por teclado.
 *
 *  @param [out]  alu  Alumno que rellenaremos
 *
 */ /************************************************************************/
void IntroducirAlumno (Alumno & alu)
{
    cout << "\n Nombre:";
    getline(cin, alu.nombre);
    if (alu.nombre != "")
    {
        cout << "\n Nota:";
        cin >> alu.nota;
        cout << "\n Curso:";
        cin >> alu.curso;
        // Borra el salto de linea del buffer
        cin.ignore();
    }

    return;
}

/************************************************************************/ /**
 *
 *  Escribe la informacion de un alumno por pantalla.
 *
 *  @param [in]  alu  Alumno que queremos mostrar por pantalla
 *
 */ /************************************************************************/
void EscribirAlumno (Alumno alu)
{
    cout << alu.nombre << endl;
    cout << alu.nota << endl;
    cout << alu.curso << endl;

    return;
}

/************************************************************************/ /**
```

```
 *
 *  Funcion para pedir una secuencia de alumnos hasta que se introduce
 *  un nombre en blanco.
 *
 *  @param [out]  v_alu    Vector de alumnos rellenado adecuadamente
 *  @param [out]  num_alu  Numero de alumnos introducido
 *
 */ /*********************************************************************/
void Introducir (V_Alumnos v_alu, int & num_alu)
{
    Alumno alu;

    cout << "Introduccion del alumno " << (num_alu + 1);
    cout << "\n Pon un nombre vacio para salir";
    IntroducirAlumno(alu);
    while (alu.nombre != "")
    {
        num_alu ++;
        v_alu[num_alu] = alu;
        cout << "Introduccion del alumno " << (num_alu + 1);
        cout << "\n Pon un nombre vacio para salir";
        IntroducirAlumno (alu);
    }

    return;
}

/*********************************************************************/ /**
 *
 *  Funcion que devuelve la posicion de un alumno del que sabemos su nombre.
 *  Devuelve -1 si el alumno no esta en el vector.
 *
 *  @param [in]  nom_alu  Nombre del alumno que buscamos
 *  @param [in]  v_alu    Vector donde estan los alumnos
 *  @param [in]  num_alu  Numero de alumnos validos del vector
 *
 *  @return      Posicion en la que se encuentra el alumno con el nombre
 *               pasado como parametro. Devuelve -1 si no se encuentra el
 *               nombre del alumno.
 *
 */ /*********************************************************************/
int PosAlu (string nom_alu, const V_Alumnos v_alu, int num_alu)
{
    int i;
    int posicion;

    i = 0;
    while ( (i < num_alu) && (v_alu[i].nombre != nom_alu) )
        i++;

    if (v_alu[i].nombre != nom_alu )
        posicion = -1;
    else
        posicion = i;

    return posicion;
}

/*********************************************************************/ /**
 *
 *  Funcion que pide el nombre del alumno a buscar y lo muestra en pantalla,
 *  si lo encuentra en el vector.
 *
 *  @param [in]  v_alu    Vector donde estan los alumnos
 *  @param [in]  num_alu  Numero de alumnos validos del vector
```

```
 *
 *  @retval true  Si hemos encontrado al alumno
 *  @retval false Si no hemos encontrado al alumno
 *
 */ /*************************************************************************/
bool EncontrarAlumno (V_Alumnos v_alu, int num_alu)
{
    int posicion;
    string nombre;

    cout << "\n Que alumno quieres encontrar? \n";
    getline (cin, nombre);
    posicion = PosAlu (nombre, v_alu, num_alu);
    if (posicion == -1)
        cout << "Alumno no encontrado \n";
    else
        EscribirAlumno (v_alu[posicion]);

    return posicion == 1;
}
```

Ejemplo 3:

```
/************************* Ejemplo_3.cpp ********************************/ /**
 *
 * @file Ejemplo_3.cpp
 *
 * @brief Prácticas de Informatica
 *
 * Práctica 1. Construir un grafo a partir de la información de los nodos y
 *             arcos contenidos en dos ficheros y a partir del grafo determinar
 *             los puntos de ruptura del grafo y mostrarlos por pantalla y
 *             guardar la informacion en un fichero binario.
 *
 * @version 0.9
 * @author Ricardo Ferris
 * @date 06-05-2005
 */ /*************************************************************************/

#include <iostream>
#include <fstream>
#include <string>
using namespace std;

#include "00Definiciones.h"
#include "00Lista.h"
#include "00Grafo.h"

void Presentacion (void);
bool LeerGrafo (Grafo &);
void DeterminarPuntosDeRuptura (Grafo, Lista &);
void MostrarLista (Lista);
bool GuardarLista (Lista);

/*************************************************************************
 ********************** Programa Principal ****************************
 *************************************************************************/
int main (void)
{
    Grafo grafo;
    Lista list;

    Presentacion ();

    /*
     * Intentamos leer el grafo desde fichero
     */
    if (!LeerGrafo (grafo) )
        /*
         * Si la lectura no es posible informamos al usuario
         */
        cout << "Se ha producido algun error en la apertura de los ficheros.";
    else
    {
        /*
         * Si hemos conseguido leer el grafo, buscamos los puntos de ruptura,
         * los mostramos por pantalla y los guardamos en un fichero
         */
        DeterminarPuntosDeRuptura (grafo, list);
        MostrarLista (list);
        GuardarLista (list);
    }

    return 0;}
```

```
/*********************************************************************/ /**
 *
 *  Lee la información de los ficheros de nodos y de arcos y la pone en
 *  el vector de nodos y la matriz de arcos.
 *  En principio los ficheros son fijos y son "Nodos.dat" y "Relaciones.dat".
 *
 *  @param [out]  grafo  Grafo que queremos rellenar
 *
 *  @return       true  -> La operacion ha podido ser realizada con exito<br>
 *                false -> La operacion no ha podido ser realizada con exito
 *                         (Ha habido problemas abriendo alguno de los ficheros)
 *
 */ /*********************************************************************/
bool LeerGrafo (Grafo & grafo)
{
    return grafo.LeerGrafo ("Nodos.dat", "Relaciones.dat");
}

/*********************************************************************/ /**
 *
 *  Determina los puntos de ruptura existentes en el grafo y los guarda en una
 *  lista.
 *
 *  @param [in]      grafo  Grafo de que queremos determinar los puntos de
 *                          ruptura
 *  @param [out]     list   Lista donde guardaremos los puntos de ruptura
 *
 */ /*********************************************************************/
void DeterminarPuntosDeRuptura (Grafo grafo, Lista & list)
{
    grafo.DeterminarPuntosDeRuptura (list);

    return;
}

/*********************************************************************/ /**
 *
 *  Muestra por pantalla la información referente a los elementos (planetas)
 *  que estan en la lista
 *
 *  @param [in]  list  Lista que queremos mostrar por pantalla
 *
 */ /*********************************************************************/
void MostrarLista (Lista list)
{
    Planeta x;

    cout << "Los planetas que son puntos de ruptura en la red de ";
    cout << "comunicaciones son los siguientes:\n";

    list.IrAInicio ();

    while (!list.FinalLista () )
    {
        list.Consultar (x);
        cout << x.codigo << " - ";
        cout << x.nombre;
        cout << " (" << x.coord[0] << "," << x.coord[1];
        cout << ","  << x.coord[2] << ")" << endl;
        list.Avanzar ();
    }
}

/*********************************************************************/ /**
 *
```

```
 *  Guarda en un fichero binario, 'ruptura.dat', la informacion de los
 *  elementos (planetas) guardados en la lista pasada como parametro
 *
 *  @param [in]  list  Lista donde estan guardados los elementos a guardar
 *
 *  @return      true  -> La operacion ha podido ser realizada con exito<br>
 *               false -> La operacion no ha podido ser realizada con exito
 *                        (Ha habido problemas abriendo el fichero)
 *
 */ /*********************************************************************/
bool GuardarLista (Lista list)
{
    ofstream f;
    Planeta x;
    bool exito;

    f.open ("ruptura.dat");

    if (!f)
        exito = false;
    else
    {
        exito = true;
        list.IrAInicio ();
        while (!list.FinalLista () )
        {
            list.Consultar (x);
            f.write ( (char *)(&x), sizeof (x) );
            list.Avanzar ();
        }
        f.close ();
    }

    return exito;
}

/*********************************************************************/ /**
 *
 *  Muestra por pantalla una pequeña explicacion de la tarea que realiza
 *  el programa
 *
 */ /*********************************************************************/
void Presentacion (void)
{
    cout << endl;
    cout << "  ***************** PRACTICA 8: GRAFOS *************************\n";
    cout << endl;
    cout << "   Este programa determina los puntos de ruptura de un grafo.\n";
    cout << endl;
    cout << "   La informacion del grafo debe estar guardada en dos ficheros\n";
    cout << "   de texto,  uno para la informacion  de  los nodos (Nodos.dat)\n";
    cout << "   y otro para los arcos (Relaciones.dat).";
    cout << endl;
    cout << "   Al finalizar el programa se muestran por pantalla y se guardan\n";
    cout << "   en un fichero binario los datos de los puntos de ruptura del\n";
    cout << "   grafo.\n";
    cout << endl;
    cout << "  ************************************************************\n";
    cout << "  *                                                          *\n";
    cout << "  *   Programa desarrollado por Ricardo Ferris.              *\n";
    cout << "  *   Version 0.9                                            *\n";
    cout << "  *                                                          *\n";
    cout << "  ************************************************************\n";
    cout << endl;
    cout << endl;
```

```
    cout << "      ";
    system ("PAUSE");

    return;
}
```

Ejemplo 4: Clases

Fichero: **CalcularE.cpp**

```
/*************************************************************************/ /**
 *
 * @file CalcularE.cpp
 *
 * Este programa calcula el numero E utilizando fracciones.
 *
 * @version 1.0
 * @author Ricardo Ferris
 * @date 07/05/2010
 *
 *************************************************************************/

#include <iostream>
using namespace std;
#include "fraccion.h"

int Factorial (int);

int main()
{
    Fraccion res, num;
    float error;
    int i;

    cout << "Dame error aceptado: ";
    cin >> error;

    res.Asignar(0, 1);

    i = 0;
    num.Asignar(1, Factorial (i) );
    while (num.Evaluar() > error)
    {
        res = res + num;
        i++;
        num.Asignar(1, Factorial (i) );
    }

    cout << "El numero e calculado es " << res.Evaluar() << endl;

    return 0;
}

/*********************************************************************/ /**
 *
 *  Calcula el factorial de un numero
 *
 *  @pre  El valor de x debe ser cero o mayor que cero
 *
 *  @param [in]  x   Numerador a asignar
 *
 *  @return  Valor del factorial de 'x'
 *
 */ /*****************************************************************/
int Factorial (int x)
{
    int res, i;

    res = 1;
```

```
    for (i = 1; i <= x; i++)
        res *= i;

    return res;
}
```

Fichero: **Fraccion.h**

```
/*************************************************************************/ /**
 *
 * @file Fraccion.h
 *
 * Fichero de cabecera de la clase Fraccion
 *
 * @version 1.0
 * @author Ricardo Ferris
 * @date 07/05/2010
 *
 *************************************************************************/

/**
 * Clase para el manejo de fracciones
 */
class Fraccion
{
    public:
        void Asignar (int num, int denom);
        void Leer (void);
        void Escribir (void);
        Fraccion operator+ (Fraccion f) const;
        Fraccion operator- (Fraccion f) const;
        Fraccion Menos (void) const;
        float Evaluar (void);
        friend bool operator== (Fraccion f1, Fraccion f2);
        friend ostream& operator<< (ostream &o, const Fraccion &a);
        friend istream& operator>> (istream &i, Fraccion &a);
    private:
        int numerador;   ///< Entero en el que guardaremos el numerador de la fraccion
        int denominador; ///< Entero en el que guardaremos el denominador de la fraccion
        void Simplificar (void);
};
```

Fichero: **Fraccion.cpp**

```
/*************************************************************************/ /**
 *
 * @file Fraccion.cpp
 *
 * Implementación de los métodos de la clase Fraccion
 *
 * @version 1.0
 * @author Ricardo Ferris
 * @date 07/05/2010
 *
 *************************************************************************/

#include <iostream>
#include <cmath>
using namespace std;
#include "Fraccion.h"
```

```
/*****************************************************************/ /**
 *
 *  Asigna al numerador y denominador de la fraccion que hace la llamada
 *  los valores de los parametros pasados
 *
 *  @param [in]  num    Numerador a asignar
 *  @param [in]  denom  Denominador a asignar
 *
 */ /*****************************************************************/
void Fraccion::Asignar (int num, int denom)
{
   if (denom == 0)
   {
      cout << "Denominador cero. Operacion anulada";
      denominador = 1;
      numerador = 0;
   }
   else
   {
      numerador = num;
      denominador = denom;
   }

   Simplificar();
}

/*****************************************************************/ /**
 *
 *  Lee de teclado los valores del numerador y del denominador.
 *  Utiliza para hacerlo la sobrecarga del operador <<
 *
 */ /*****************************************************************/
void Fraccion::Leer (void)
{
    cin >> *this;

    return;
}

/*****************************************************************/ /**
 *
 *  Sobrecarga del operador de entrada. Método amigo de la clase fraccion
 *
 *  @param        i  stream del que leeremos la fraccion
 *  @param [out]  a  Fraccion a rellenar
 *
 */ /*****************************************************************/
istream& operator>> (istream &i, Fraccion &a)
{
    int num, denom;

    cout << "Dame numerador: ";
    i >> num;

    cout << "Dame denominador: ";
    i >> denom;

    a.Asignar (num, denom);

    return i;
}
```

```
/*****************************************************************/ /**
 *
 *  Muestra por pantalla la fraccion. utiliza para ello la sobrecarga del
 *  operador de <<
 *
 */ /*****************************************************************/
void Fraccion::Escribir (void)
{
    cout << *this;
}

/*****************************************************************/ /**
 *
 *  Sobrecarga del operador de salida. Método amigo de la clase fraccion
 *
 *  @param       o  stream en el que escribiremos la fraccion
 *  @param [in]  a  Fraccion a mostrar
 *
 */ /*****************************************************************/
ostream& operator<< (ostream &o, const Fraccion &a)
{
    o << a.numerador << " / " << a.denominador;

    return o;
}

/*****************************************************************/ /**
 *
 *  Sobrecarga del operador suma (+). Suma de dos fracciones
 *
 *  @param [in]  f  Fraccion a sumar
 *
 *  @return  Devuelve la fraccion suma resultante
 *
 */ /*****************************************************************/
Fraccion Fraccion::operator+ (Fraccion f) const
{
   Fraccion suma;

   suma.denominador = denominador * f.denominador;
   suma.numerador = numerador * f.denominador + denominador * f.numerador;
   suma.Simplificar();

   return suma;
}

/*****************************************************************/ /**
 *
 *  Sobrecarga del operador resta (-). Resta dos fracciones
 *
 *  @param [in]  f  Fraccion a restar
 *
 *  @return  Devuelve la fraccion resta resultante
 *
 */ /*****************************************************************/
Fraccion Fraccion::operator- (Fraccion f) const
{
   Fraccion resta;

   resta.denominador = denominador * f.denominador;
   resta.numerador = numerador * f.denominador - denominador * f.numerador;
   resta.Simplificar();

   return resta;
}
```

```
/*********************************************************************/ /**
 *
 *  Función que cambia de signo la fraccion
 *
 *  @return  Devuelve la fraccion cambiada de signo
 *
 */ /*********************************************************************/
Fraccion Fraccion::Menos (void) const
{
   Fraccion f;

   f.numerador = -numerador;
   f.denominador = denominador;

   return f;
}

/*********************************************************************/ /**
 *
 *  Función que nos dice el valor real de una fraccion
 *
 *  @return  Devuelve el valor 'float' de la fraccion
 *
 */ /*********************************************************************/
float Fraccion::Evaluar (void)
{
   return float (numerador) / denominador;
}

/*********************************************************************/ /**
 *
 *  Método que nos dice si dos fracciones son iguales. Método amigo de
 *  la clase fraccion
 *
 *  @retval  true   Si las dos fracciones son iguales
 *  @retval  false  Si las dos fracciones no son iguales
 *
 */ /*********************************************************************/
bool operator== (Fraccion f1, Fraccion f2)
{
   return f1.numerador == f2.numerador && f1.denominador == f2.denominador;
}

/*********************************************************************/ /**
 *
 *  Método privado que pone la fracción en su forma canonica.
 *
 */ /*********************************************************************/
void Fraccion::Simplificar (void)
{
    int max, i;

    if (abs (denominador) > abs (numerador) )
       max = abs (denominador);
    else
       max = abs (numerador);

    i = 2;
    while (i <= max)
    {
       if ( ( (numerador % i) == 0 ) &&
            ( (denominador % i) == 0 ) )
       {
          numerador /= i;
```

```
            denominador /= i;
        }
        else
            i++;
    }

    return;
}
```