

Tecnología de la Programación

Boletín de Prácticas

Curso 2020-2021

Grupos 1, 4 y 9

TABLA DE CONTENIDOS

| | |
|---|-----------|
| PROYECTO DE PROGRAMACIÓN | 3 |
| DESCRIPCIÓN DEL TRABAJO A REALIZAR..... | 3 |
| ENTREGABLES | 4 |
| FECHA Y LUGAR DE ENTREGA..... | 4 |
| CRITERIOS DE EVALUACIÓN..... | 5 |
| SESIONES DE PRÁCTICAS GUIADAS | 6 |
| SESIÓN 1. INTRODUCCIÓN AL PROYECTO DE PROGRAMACIÓN Y A CODE::BLOCKS | 6 |
| SESIÓN 2. DETECCIÓN DE COLISIONES | 6 |
| SESIÓN 3. ANIMACIONES INTERACTIVAS | 7 |
| SESIÓN 4. UNA BALA..... | 8 |
| SESIÓN 5. UNA RÁFAGA DE BALAS | 9 |
| SESIÓN 6. CONTROL DE IMPACTOS | 10 |
| SESIÓN 7. PROGRAMACIÓN MODULAR Y ABSTRACCIÓN DE DATOS..... | 10 |
| SESIÓN 8. PROGRAMACIÓN MODULAR Y ABSTRACCIÓN DE DATOS II | 11 |
| SESIÓN 9. UN EJÉRCITO DE ENEMIGOS..... | 12 |
| SESIÓN 10. GESTIÓN DE RECORDS | 13 |
| SESIÓN 11. MENÚS | 13 |
| SESIÓN 12. DOCUMENTACIÓN DEL PROYECTO | 13 |
| ¿CÓMO CONSEGUIR LA MEJOR NOTA POSIBLE? | 14 |
| ANEXO A. INTRODUCCIÓN A CODE::BLOCKS | 15 |
| CREAR UN PROYECTO | 15 |
| AÑADIR CÓDIGO FUENTE A UN PROYECTO | 16 |
| COMPILAR EL CÓDIGO FUENTE Y GENERAR LA APLICACIÓN..... | 16 |
| EJECUTAR UNA APLICACIÓN | 16 |
| DEPURAR UN PROYECTO | 17 |
| ANEXO B. INTRODUCCIÓN A LOS GRÁFICOS EN C..... | 19 |
| DIBUJANDO EN LA PANTALLA CON LAS FUNCIONES BÁSICAS | 19 |
| ANIMACIONES..... | 21 |
| APLICACIONES INTERACTIVAS..... | 22 |
| ANEXO C. DOCUMENTACIÓN DE SOFTWARE CON DOXYGEN | 24 |
| USO DE DOXYGEN DESDE CODE::BLOCKS | 25 |
| ANEXO D. INSTALACIÓN DE CODE::BLOCKS, SDL Y DOXYGEN | 26 |

Proyecto de programación

Descripción del trabajo a realizar

Las prácticas de la asignatura Tecnología de la Programación se realizan individualmente y consisten en el desarrollo de un proyecto de programación que permita al estudiante aplicar las metodologías y técnicas estudiadas en la asignatura. En concreto, **debe realizarse una aplicación que cumpla los siguientes requisitos:**

- Utilizar el lenguaje C.
- Compilar sin errores ni avisos (warnings).
- Utilizar el entorno de desarrollo integrado Code::Blocks.
- Utilizar la biblioteca de funciones gráficas SDL a través del módulo Pantalla.
- Utilizar ficheros para guardar datos relevantes de la aplicación.
- Implementar y usar uno o más TDAs simples.
- Implementar y usar uno o más TDAs contenedores con representación contigua.
- Implementar y usar uno o más TDAs contenedores con representación enlazada.
- Comentar el código y generar su documentación con la herramienta Doxygen.

A lo largo de las sesiones de prácticas presenciales se trabajarán todos los aspectos necesarios para desarrollar una aplicación que cumpla los anteriores requisitos. Se podrá entregar esta aplicación como proyecto, o utilizar los conceptos aprendidos para desarrollar una aplicación de libre elección, siempre que cumpla los mismos requisitos.

Además del código y la documentación, se deberá escribir y entregar un informe describiendo el trabajo realizado, que constará de los siguientes apartados:

- **Portada** con la siguiente información: Apellidos, Nombre, DNI, Asignatura, Grupo y Subgrupo de Prácticas, Convocatoria, Año académico y Profesor.
- **Índice de contenidos.**
- **Descripción de la aplicación:** Objetivo y funcionamiento general de la aplicación.
- **Manual de usuario:** Instrucciones para utilizar la aplicación.
- **Organización del proyecto:** Descripción de los ficheros que componen la aplicación, relaciones y dependencias que hay entre los distintos módulos, así como cualquier otro aspecto relevante del proyecto.
- **Estructuras de datos:** Especificación informal de todos los Tipos de Datos Abstractos diseñados e implementados para el proyecto.
- **Conclusiones:** Comentarios personales en relación a las dificultades encontradas, la forma de superarlas y el grado de satisfacción con la asignatura, así como sugerencias de mejora.

El **código fuente** incluido en el informe deberá estar bien formateado. Es decir, se debe cuidar el sangrado y usar un tamaño de la letra adecuado para que no se corten los renglones, o que cuando esto ocurra el código siga siendo legible y ordenado. El tipo de letra para el **código fuente** será `courier` o uno semejante de ancho fijo.

Entregables

Todo el material se entregará **comprimido en un único fichero cuyo nombre será el DNI de su autor y cuyo contenido estará distribuido en carpetas** del siguiente modo:

- **Carpeta “Código”**: Contendrá únicamente los ficheros siguientes:
 - Fichero de proyecto Code::Blocks (.cbp).
 - Ficheros de código fuente (ficheros cabecera .h y ficheros fuente *.c).
 - Otros ficheros necesarios para compilar o ejecutar la aplicación tales como imágenes, textos, datos, etc.
 - No debe contener ningún fichero .c ni .h de prueba o correspondiente a versiones anteriores del código que no se utilice en el proyecto final.
- **Carpeta “Documentación”**: Contendrá todo el contenido de la carpeta html generada por Doxygen a partir de los comentarios realizadas en el código fuente de todos los ficheros del proyecto. Recuerda NO incluir el módulo Pantalla.
- **Carpeta “Informe”**: Contendrá el informe en formato PDF.

IMPORTANTE: El código debe compilar sin avisos (warnings) ni errores de compilación en unas condiciones similares a las de la instalación de Code::Blocks que hay en los laboratorios de prácticas. Además, el programa debe estar libre de errores de ejecución.

Fecha y lugar de entrega

La entrega se realizará a través de una tarea publicada en el Aula Virtual. La fecha límite para entregar el trabajo se indicará al publicarse la tarea. En dicha tarea se indicará el momento y lugar en donde se llevará a cabo una sesión de conformidad con la entrega. En dicha sesión el estudiante tendrá la oportunidad de presentar al profesor todo el material entregado a través del aula virtual para comprobar que se incluye todo lo que se solicita en el apartado Entregables, que el proyecto compila sin errores ni avisos y, además, comienza y termina la ejecución correctamente. En caso de no cumplir estos requisitos de conformidad, el alumno dispondrá de un plazo de dos días para corregir los errores y volver a realizar la entrega.

La fecha de entrega se avisará con antelación suficiente a través del aula virtual.

Criterios de evaluación

Las prácticas se realizarán **individualmente**. Si el profesor lo considerase oportuno se pedirá al estudiante la realización de una entrevista personal para aclarar detalles del trabajo realizado. En tal caso, la no asistencia (injustificada) a esta entrevista de prácticas será motivo de suspenso.

El proyecto de programación será valorado por el profesor con una puntuación de 0 a 10. Para superar la parte práctica de la asignatura será necesario obtener una nota igual o superior a 5 sobre 10.

En la evaluación se tendrá en cuenta la presentación, claridad y completitud del informe, el uso de un lenguaje técnico apropiado, la legibilidad, organización y documentación del código fuente, y la corrección, robustez, modularidad, extensibilidad, reusabilidad y eficiencia del software.

A continuación, se muestran algunos ejemplos de aspectos relevantes que se tendrán en cuenta durante la evaluación del software:

1. Legibilidad
 - Elige los identificadores de variables y funciones de forma adecuada.
 - Documenta las funciones incluyendo precondiciones, efecto y resultados.
 - Formatea el código fuente aplicando sangrado.
2. Corrección
 - Reserva la memoria dinámica necesaria y la libera cuando ya no es útil.
 - No se producen errores en tiempo de ejecución por acceso indebido a memoria.
 - No se producen bucles infinitos ni llamadas recursivas que desborden la pila.
3. Robustez
 - Da valor inicial a las variables.
 - Controla errores debidos al mal uso de la aplicación por parte del usuario.
 - Comprueba los valores devueltos por funciones que informen sobre errores.
4. Modularidad
 - Garantiza que la representación de los TDA permanece oculta.
 - Evita dependencias entre módulos debidas al uso de variables globales.
 - Distribuye el código en módulos de forma coherente.
5. Extensibilidad y Reusabilidad
 - Crea constantes o macros en lugar de usar valores literales en el código.
 - Evita el uso de variables globales.
 - Parametriza las funciones y módulos para hacerlos más generales.
6. Eficiencia
 - Selecciona la representación más adecuada para las estructuras de datos.
 - Utiliza esquemas de inserción, acceso, recorrido y búsqueda eficientes y adecuados.
 - Aprovecha los recursos del lenguaje de programación para evitar la repetición innecesaria de cálculos o acciones previamente realizadas.

Sesiones de prácticas guiadas

Sesión 1. Introducción al proyecto de programación y a Code::Blocks

Objetivo: Por un lado, conocer cómo se van a desarrollar las prácticas, cómo se evalúan y en qué consiste el proyecto de programación. Por otro lado, hacer un primer proyecto básico usando el IDE Code::Blocks.

Tareas:

1. Lee el anexo A y realiza un proyecto en Code::Blocks que implemente la aplicación “Hola Mundo”
2. Lee el anexo D e instala el software de prácticas en tu ordenador personal.

Sesión 2. Detección de colisiones

Objetivo: Aprender a escribir y probar la corrección de funciones en C y a buscar y leer la documentación de las funciones de la biblioteca estándar de C.

Tareas:

1. Escribe el código de la siguiente función para que devuelva 1 si el punto (px,py) está dentro del rectángulo cuya esquina superior izquierda esté en (x,y) y tenga w puntos de ancho y h puntos de alto. Devolverá cero en caso contrario.

```
int dentro_rectangulo( int x, int y, int w, int h, int px, int py );
```
2. Escribe el código de la siguiente función para que devuelva la distancia euclídea entre el punto (x1,y1) y el punto (x2,y2). Para ello necesitarás utilizar las funciones sqrt y pow de la biblioteca estándar de C.

```
double distancia_punto_punto( double x1, double y1, double x2, double y2 );
```
3. Escribe el código de la siguiente función para que devuelva 1 si la circunferencia con centro en (x1,y1) y radio r1 tiene alguna parte solapada con la de centro en (x2,y2) y radio r2. Devolverá cero en caso contrario.

```
int solape_circunferencias( int x1, int y1, int r1, int x2, int y2, int r2 );
```
4. Escribe el código de la siguiente función para que devuelva 1 si el rectángulo con esquina superior izquierda en (x1,y1), ancho w1 y alto h1 tiene alguna parte solapada con el rectángulo con esquina superior izquierda en (x2,y2), ancho w2 y alto h2. Devolverá cero en caso contrario. Si tienes dudas quizá te venga bien consultar la web <http://ants.inf.um.es/staff/ilaguna/tp/tutoriales/colisiones>.

```
int solape_rectangulos( int x1, int y1, int w1, int h1, int x2, int y2, int w2, int h2 );
```
5. Escribe un programa que sirva para comprobar que las funciones anteriores son correctas. Necesitarás usar la función fabs de la biblioteca estándar de C.
6. Crea un proyecto nuevo para calcular el valor aproximado de PI mediante una simulación de Montecarlo. Para ello, escribe un programa que genere N puntos aleatorios cuyas coordenadas estén dentro de un cuadrado de lado L. Cuenta el número de puntos que caen dentro del círculo de radio L/2 centrado en el centro del cuadrado. Si llamamos a ese número C, el valor aproximado de PI será $4.0 \cdot C/N$.

Sesión 3. Animaciones interactivas

Objetivo: Aprender a escribir aplicaciones gráficas interactivas con el módulo Pantalla, y a utilizar los eventos de teclado para controlar el movimiento de un objeto gráfico.

Tareas:

1. Lee el Anexo B, y escribe un programa que muestre un cuadrado en la pantalla que se pueda mover horizontal y verticalmente usando las flechas del teclado. Este será el héroe del juego.
2. Añade al programa otro cuadrado en la pantalla que actúe de enemigo. Consigue que se mueva automáticamente por la pantalla. Para aprender qué variables necesitarás y cómo utilizarlas puedes leer los tutoriales sobre animación de la página web: <https://ants.inf.um.es/staff/jlaguna/tp/tutoriales.php>
3. Usa las funciones de la sesión 2 para que cuando el héroe toque al enemigo termine el bucle principal de la animación y con él la partida.
4. Lee la documentación del módulo pantalla que se encuentra en los recursos del aula virtual y usa las funciones `Pantalla_ImagenLee`, `Pantalla_DibujaImagen` y `Pantalla_ImagenLibera` para sustituir el rectángulo que representa al héroe por la imagen "tux.bmp". El cuadrado del enemigo puedes sustituirlo por la imagen "apple.bmp". Ambas imágenes se encuentran en los recursos del aula virtual.

Sesión 4. Una bala

Objetivo: Aprender a representar elementos complejos mediante la agrupación de otros más simples en estructuras situadas en memoria dinámica, y a definir funciones que reciban como parámetros o devuelvan punteros a estas estructuras.

Tareas:

1. Define la estructura `struct BalaRep` para representar una bala rectangular que se mueva con cierta velocidad constante. ¿Qué campos necesitará tener? Añade un campo de tipo `Imagen` para almacenar la imagen que representará gráficamente la bala. Para poder usar el tipo `Imagen` debes incluir el fichero de cabecera "Pantalla.h".
2. Define el Tipo de Datos `Bala` como un puntero a esa estructura.
3. Escribe el código de la función siguiente para que devuelva la dirección en memoria dinámica de una nueva estructura `struct BalaRep` con la posición (x,y) y velocidad (vx,vy) recibidas como parámetros. La imagen puede ser la del fichero "ubuntu.bmp" que encontrarás en los recursos del Aula Virtual.

```
Bala crea_bala ( double x, double y, double vx, double vy );
```

4. Escribe el código de la siguiente función para que libere la memoria usada por la bala `b`. Recuerda que también hay que liberar la memoria asociada a la imagen.

```
void libera_bala( Bala b );
```

5. Escribe el código de la siguiente función para que modifique la posición de la bala en función de su propia velocidad.

```
void mueve_bala( Bala b );
```

6. Escribe el código de la siguiente función para que dibuje en pantalla la representación gráfica de la bala `b`.

```
void dibuja_bala( Bala b );
```

7. Escribe el código de la siguiente función para que devuelva el valor de la coordenada horizontal de la bala `b`.

```
double get_x_bala( Bala b );
```

8. Escribe el código de la siguiente función para que devuelva el valor de la coordenada vertical de la bala `b`.

```
double get_y_bala( Bala b );
```

9. Añade al programa una variable de tipo `Bala` y asígnale `NULL`. Usando las funciones anteriores, consigue que al pulsar una tecla se cree una bala en la posición del héroe y se asigne a la variable anterior, que cuando la bala exista se mueva y se dibuje, y que antes de terminar el programa se libere su memoria.
10. Modifica el programa para que cuando la bala desaparezca por la parte superior de la pantalla se libere su memoria y la variable vuelva a valer `NULL`. Si tras hacer esto tu programa falla revisa el código para evitar usar la variable para dibujar o mover la bala cuando esta valga `NULL`.

Sesión 5. Una Ráfaga de Balas

Objetivo: Aprender a utilizar estructuras enlazadas lineales en C.

Tareas:

1. Define la estructura `struct Nodo` para representar el nodo de una lista enlazada lineal con cabecera cuyos elementos sean balas, es decir, una ráfaga de balas. Define también el alias `NodoPtr` como un puntero a dicha estructura.

2. Escribe el código de la función siguiente para que devuelva un nuevo nodo que contenga la bala recibida como parámetro.

```
NodoPtr nuevo_nodo( Bala b );
```

3. Escribe el código de la siguiente función para que inserte un nuevo nodo a la lista de balas cuya cabecera se recibe como parámetro.

```
void inserta_rafaga( NodoPtr cabecera, Bala b );
```

4. Escribe el código de la siguiente función para que libere toda la memoria de la lista de balas cuya cabecera se recibe como parámetro.

```
void libera_rafaga( NodoPtr cabecera );
```

5. Escribe el código de la siguiente función para que mueva, ejecutando la función `mueve_bala`, todas las balas incluidas en la lista de balas cuya cabecera se recibe como parámetro.

```
void mueve_rafaga( NodoPtr cabecera );
```

6. Escribe el código de la siguiente función para que dibuje, ejecutando la función `dibuja_bala`, todas las balas incluidas en la lista de balas cuya cabecera se recibe como parámetro.

```
void dibuja_rafaga( NodoPtr cabecera );
```

7. Borra todo el código relacionado con la variable de tipo `Bala` de la sesión anterior incluida la declaración de la variable que se usaba para guardar la bala. Crea una nueva variable que represente una ráfaga de balas vacía, es decir, una lista de balas vacía. No olvides crear el nodo cabecera antes de intentar añadir elementos a la lista o recorrerla para dibujar los que hayan.

8. Haz que al pulsar una tecla se cree una nueva `Bala` en la posición del héroe con cierta velocidad inicial. Tras crear la bala añádela a la ráfaga de balas.

9. En cada iteración del bucle principal se deben mover y dibujar (en este orden) todas las balas que haya en la ráfaga. De momento no te preocupes por eliminar las balas de la ráfaga que se salen de la pantalla ni de comprobar los impactos con el enemigo, esto se hará en la siguiente sesión.

10. No olvides liberar la memoria de la ráfaga de balas antes de que termine la función `main`.

Sesión 6. Control de Impactos

Objetivo: Aprender a utilizar estructuras enlazadas lineales en C.

Tareas:

1. Modifica el código de la función `mueve_rafaga` para que elimine **todas** las balas que, tras moverse, estén situadas fuera de la pantalla.
2. Escribe el código de la siguiente función para que devuelva 1 si el rectángulo con esquina superior izquierda en (x, y), anchura w y altura h se solapa con la bala b. En caso contrario devolverá 0.

```
int colision_bala( Bala b, double x, double y, double w, double h );
```

3. Escribe el código de la siguiente función para que busque y elimine la primera bala que se solape con el rectángulo con esquina superior izquierda en (x,y), anchura w y altura h. La función devolverá 1 si eliminó alguna bala y 0 en caso contrario.

```
int colision_rafaga(NodoPtr cabecera, double x, double y, double w, double h);
```

4. Usa las funciones anteriores para detectar si se ha impactado en el enemigo y, en caso afirmativo, hacer que el jugador gane un punto y crear un nuevo enemigo en otra posición. Para esto último basta con que vuelvas a generar unas coordenadas aleatorias para el mismo, así en la siguiente iteración se dibujará y moverá desde otro sitio y dará la impresión de que ha aparecido otro nuevo enemigo.
5. Modifica el programa para que el héroe sólo pueda moverse en horizontal y esté situado en la parte inferior de la pantalla.

Sesión 7. Programación modular y Abstracción de datos

Objetivo: Aprender a separar el código en módulos y a implementar TDAs en C

Tareas:

1. Organiza el código en módulos creando el módulo Colisiones en el que se incluirán las funciones realizadas en la sesión 2, y por otro lado, el TDA Bala usando las funciones de la sesión 4 junto con la función `colision_bala` de la sesión 6.
2. Añade al proyecto el TDA Bala y el módulo Colisiones y modifica el programa principal para que los pueda usar incluyendo sus ficheros de cabecera.
3. Escribe la especificación informal del TDA Bala en un documento que terminará siendo parte del informe a entregar junto con el proyecto.

Sesión 8. Programación modular y Abstracción de datos II

Objetivo: Aprender a implementar un TDA Contenedor con representación enlazada en C

Tareas:

1. Implementa el TDA Rafaga cumpliendo con la siguiente especificación informal:

Nombre: Rafaga

Descripción: Representa una colección de balas que se mueven de forma autónoma.

Domino de valores: Cualquier conjunto de balas de cualquier tamaño.

Interfaz público: Consta de las siguientes funciones

```
Rafaga crea_rafaga();
```

Esta función devuelve una nueva ráfaga vacía, es decir, sin ninguna bala, pero lista para almacenar cualquier número de balas.

```
void libera_rafaga( Rafaga r );
```

Esta función libera toda la memoria asociada a la ráfaga r y las balas que contenga.

```
void inserta_rafaga( Rafaga r, Bala b );
```

Esta función añade a la ráfaga r la bala b.

```
void mueve_rafaga ( Rafaga r );
```

Esta función mueve todas las balas contenidas en la ráfaga r y elimina **todas** las balas que, tras moverse, quedan situadas fuera de la pantalla.

```
void dibuja_rafaga ( Rafaga r );
```

Esta función muestra todas las balas contenidos en la ráfaga r.

```
int longitud_rafaga(Rafaga r );
```

Esta función devuelve el número de balas que incluye la ráfaga r.

```
int colision_rafaga( Rafaga r, double x, double y, double w, double h );
```

Esta función busca y elimina la primera bala incluida en la ráfaga r que se solape con el rectángulo con esquina superior izquierda en (x,y), anchura w y altura h. La función devolverá 1 si eliminó alguna bala y 0 en caso contrario.

Notas: La representación interna del TDA Rafaga debe usar una estructura enlazada para almacenar las balas y sus operaciones deben ser lo más eficientes posibles.

2. Modifica el programa principal para que use el TDA Ráfaga.

Sesión 9. Un ejército de enemigos

Objetivo: Aprender a implementar un TDA Contenedor con memoria contigua en C.

Tareas:

1. Implementa el TDA Ejercito cumpliendo con la siguiente especificación informal:

Nombre: Ejercito

Descripción: Representa una colección de enemigos que se mueven de forma autónoma.

Domino de valores: Cualquier conjunto de enemigos de un tamaño máximo N.

Interfaz público: Consta de las siguientes funciones

```
Ejercito crea_ejercito();
```

Esta función devuelve un nuevo ejército vacío, es decir, sin ningún enemigo, pero con capacidad para almacenar hasta N enemigos. Elige el valor de N que te parezca más adecuado.

```
void libera_ejercito( Ejercito e );
```

Esta función libera toda la memoria asociada al ejército e.

```
void inserta_enemigo ( Ejercito e, double x, double y, double w, double h );
```

Esta función crea un nuevo enemigo en las coordenadas en (x,y), anchura w y altura h, y lo añade al grupo de enemigos que forma el ejército e, siempre que aún quede espacio libre.

```
void mueve_ejercito( Ejercito e );
```

Esta función mueve todos los enemigos contenidos en el ejército e.

```
void dibuja_ejercito( Ejercito e );
```

Esta función muestra todos los enemigos contenidos en el ejército e.

```
int colision_ejercito_objeto( Ejercito e, double x, double y, double w, double h );
```

Esta función devuelve 1 si el rectángulo con esquina superior izquierda en (x,y), anchura w y altura h se solapa con el de alguno de los enemigos que contenga el ejército e.

```
int colision_ejercito_rafaga( Ejercito e, Rafaga r );
```

Esta función elimina todos los enemigos incluidos en el ejército e que colisionen con alguna de las balas de la ráfaga r. Devuelve el número de enemigos eliminados.

Notas: La representación interna del TDA Ejercito debe usar un array para almacenar los enemigos y sus operaciones deben ser lo más eficientes posible.

2. Modifica el programa principal para que se use un ejército compuesto inicialmente por varios enemigos en lugar de un único enemigo. Añade más enemigos cada cierto tiempo, por ejemplo, cada 25 iteraciones del bucle principal de la animación.
3. Haz que el jugador gane un punto por cada enemigo eliminado y pierda una vida cada vez que colisione con uno. Usa un par de variables para guardar estos datos.

Sesión 10. Gestión de records

Objetivo: Practicar el manejo de ficheros en C y la función `sprintf`.

Tareas:

1. Muestra la puntuación y las vidas restantes en pantalla con ayuda de las funciones `sprintf` de la biblioteca estándar de C y `Pantalla_DibujaTexto` de la biblioteca `Pantalla`. Consulta el manual de `sprintf`, la explicación del tema 1 de los apuntes y la documentación del módulo `Pantalla`.
2. Usa un fichero de texto para guardar la mejor puntuación conseguida. Tras cada partida se comparará la puntuación obtenida con la almacenada en el fichero. Si la obtenida es mayor que la almacenada se actualizará el fichero informando al usuario del nuevo record conseguido.

Sesión 11. Menús

Objetivo: Aprender a manejar el ratón, practicar la simulación del paso de parámetros por referencia, organizar el programa principal en funciones y reducir el uso de variables globales

Tareas:

1. Escribe el código de la función siguiente para que muestre un menú gráfico en pantalla. Tendrá, al menos, tres opciones: Jugar, Ayuda y Salir. Cada una estará enmarcada mediante un rectángulo simulando un botón. El usuario podrá elegir la opción haciendo clic con el ratón dentro del rectángulo deseado. Necesitarás las funciones `Pantalla_RatonBotonPulsado` y `Pantalla_RatonCoordenadas` de la biblioteca `Pantalla`. La función terminará devolviendo un código numérico que represente la opción elegida.

```
int menu();
```

2. Escribe el código de la función siguiente para que muestre una imagen con la explicación del juego. El bucle principal de esta animación terminará cuando se pulse una tecla concreta o se haga clic con el ratón en una zona concreta.

```
int ayuda();
```

3. Crea una función que se encargue del desarrollo de una partida y usa las funciones anteriores desde el programa principal para que al iniciar la aplicación aparezca el menú principal y, dependiendo de lo que elija el usuario se ejecute la función que muestra la ayuda, la que lleva a cabo una partida o se salga del programa.

Sesión 12. Documentación del proyecto

Objetivo: Comprender el concepto de documentación de software y su importancia, y aprender a manejar Doxygen.

Tareas:

1. Lee el anexo C para después documentar el software realizado utilizando Doxygen
2. Genera la documentación en formato HTML de todo el proyecto excepto del módulo `Pantalla`.

¿Cómo conseguir la mejor nota posible?

Haciendo correctamente todo lo descrito en los guiones de las sesiones prácticas se puede alcanzar una nota de 7 puntos. Para obtener una calificación superior se deben añadir otras características. A continuación, se muestran algunos ejemplos que, de hacerse correctamente, sumarían un punto extra cada uno hasta un máximo de 10. Si haces alguno de estos añade su hashtag en el texto al entregar la tarea a través del aula virtual:

- #MasTDAs: Crea y usa en el programa un TDA para representar al personaje principal y otro TDA para representar a un enemigo.
- #EjercitoDinamico: Modifica el TDA Ejército para que el tamaño máximo se determine mediante un parámetro en la función crea_ejercito.
- #BalaVieja: Consigue que la función colision_rafaga del TDA Rafaga elimine la bala más antigua, es decir, aquella que lleve más tiempo en la lista.
- #DoblaEnemigos: Modifica la función colision_ejercito_rafaga para que, por cada enemigo eliminado, y siempre que quede espacio suficiente, se añadan dos nuevos de menor tamaño. Haz esto sólo cuando el tamaño sea mayor que un valor predeterminado.
- #MasEnemigos: Añade al juego otros tipos de enemigos con representación gráfica y movimientos diferentes: seguir al personaje, sólo horizontal o vertical, etc.
- #Invulnerabilidad: Dota al personaje principal de invulnerabilidad durante un tiempo determinado justo después de perder una vida.
- #Animaciones: Añade animaciones o efectos especiales al personaje principal o los enemigos.
- #ImagenCompartida: Reduce la memoria usada haciendo que todos los enemigos usen la misma imagen (o imágenes) en lugar de tener cada uno su propia copia en memoria.
- #EnemigosDisparan: Haz que los enemigos también disparen.
- #LeeEjercito: Añade al TDA Ejército la siguiente función que sea capaz de crear un ejército a partir de los datos de un fichero de texto que contenga, por ejemplo, como primera línea un entero indicando cuántos enemigos contendrá el ejército descrito a continuación, y después, una fila por cada enemigo compuesta por cuatro números enteros que se correspondan con las coordenadas horizontal y vertical de la esquina superior izquierda de la posición inicial del enemigo, su anchura y su altura.

```
Ejercito lee_ejercito( char * fichero );
```

Anexo A. Introducción a Code::Blocks

Cuando se programan aplicaciones de cierto tamaño usando el paradigma de programación modular conviene disponer de herramientas que ayuden a la gestión de los múltiples ficheros que componen cada proyecto. A las herramientas que facilitan la escritura del código, la gestión de los ficheros que componen la aplicación y su compilación y depuración, se les denomina Entorno de Desarrollo Integrado (IDE).

Code::Blocks (<http://www.codeblocks.org/>) es un Entorno de Desarrollo Integrado de software libre y multi-plataforma que se distribuye bajo los términos de la Licencia General Pública de GNU. Funciona en Windows, Linux y OS X, y con él se desarrollarán las prácticas de la asignatura, las cuales se realizarán en lenguaje C.

Para cada aplicación que se vaya a desarrollar es recomendable crear una carpeta donde se guardarán todos los ficheros que la compongan. Code::Blocks facilita este proceso a través del concepto de proyecto. Durante el desarrollo de la asignatura se van a crear múltiples proyectos, así pues, conviene tenerlos todos en una única carpeta denominada, por ejemplo, “prácticas” que estaría dentro de la carpeta “TP” en una organización lógica de carpetas por asignaturas.

Además, conviene utilizar algún sistema de control de versiones o copias de seguridad para evitar problemas mayores si se pierde algún fichero. Por ejemplo, sincronizar la carpeta de prácticas con algún servicio de disco en la red, como UmuBox, es una buena idea.

Crear un proyecto

Para crear un proyecto en Code::Blocks se hace clic en la opción “New” del menú “File” y se elige “Project”. Esto hace aparecer una ventana donde se puede elegir el tipo de proyecto. En esta asignatura sólo se usará el tipo de proyecto denominado “Empty”.

Después de esto, Code::Blocks pregunta el nombre del proyecto y la carpeta donde guardarlo. Dentro de la carpeta elegida, Code::Blocks creará otra carpeta con el mismo nombre del proyecto y dentro guardará todos los ficheros del proyecto.

A continuación, Code::Blocks pregunta qué compilador se desea usar. En el caso de esta asignatura se elegirá GNU GCC. Y en la misma ventana nos deja elegir las configuraciones a utilizar en el proyecto: una de desarrollo “debug” y otra de producción “release”. Lo recomendable es dejar marcadas las dos.

Cuando el compilador genera la versión de desarrollo incluye en el código ensamblador instrucciones específicas, denominadas trazas, que permitan depurar la aplicación. Por su parte, la versión de producción no contiene esas trazas para el depurador.


Lo habitual es desarrollar la aplicación usando la configuración de desarrollo y, una vez que se ha terminado, se usa la configuración de producción para generar el ejecutable final, que, en un entorno de trabajo real, se entregaría al cliente.

Añadir código fuente a un proyecto

Hay dos formas habituales de añadir ficheros con código fuente a un proyecto. La primera consiste en crear un nuevo fichero usando el propio Code::Block. Esto se hace mediante la opción “New” del menú “File” y eligiendo “Empty file”. Al hacerlo, Code::Blocks pregunta si se desea añadir el nuevo fichero al proyecto, y si se contesta que sí, muestra una ventana para elegir el nombre del nuevo del fichero y dónde guardarlo. Lo normal será guardarlo en la propia carpeta del proyecto. Además, antes de terminar pregunta si se quiere añadir el nuevo fichero a las dos configuraciones: desarrollo y producción. A esto también se debe contestar que sí.


La otra forma de añadir ficheros tiene sentido cuando se quiere agregar ficheros fuente previamente escritos a un proyecto. Para ello se hace clic con el botón derecho del ratón sobre el nombre del proyecto y se elige la opción “Add files”. Se pueden seleccionar varios a la vez y Code::Blocks los añade todos, cada uno en su sitio. Lo más habitual es copiar primero los ficheros a la carpeta del proyecto y después añadirlos al mismo como se acaba de comentar.


Compilar el código fuente y generar la aplicación

Para generar una aplicación hay que compilar el código fuente mediante la opción “Build” del menú “Build” o haciendo clic en el botón con el icono . Para acelerar el proceso, sólo se compilan aquellos ficheros que hayan cambiado desde la última compilación exitosa. Si todo va bien, se genera un fichero objeto por cada fichero fuente y se enlazan para generar el fichero ejecutable de la aplicación. Si no, los errores o warnings encontrados aparecerán en la ventana inferior de Code::Blocks. Se debe corregir el primero y volver a compilar hasta que se obtengan 0 errores y 0 warnings.

El proyecto que se compila es el que esté activo. Es decir, el que tenga su nombre en negrita en la pestaña “Projects” de la ventana que aparece a la izquierda. Code::Blocks permite tener varios proyectos en esa ventana, pero sólo uno activo. Para activar un proyecto se hace clic con el botón derecho sobre su nombre y se elige la opción “Activate project”.


Ejecutar una aplicación

Finalmente, se ejecuta la aplicación mediante la opción “Run” del menú “Build”, o haciendo clic en el botón con el icono . El fichero ejecutable también podrá ejecutarse desde el propio Interface Gráfico del Sistema Operativo o desde el símbolo del sistema.

La opción “Rebuild” del menú “Build” realiza todo el proceso de compilación y construcción desde el principio, independientemente de que los ficheros fuentes hayan sido modificados o no. Esta misma acción se consigue haciendo clic en el botón con el icono .

Depurar un proyecto

Las herramientas de depuración han sido creadas para ayudar a encontrar errores en un programa mientras éste se está ejecutando. Un depurador (*debugger*) es una herramienta que puede controlar la ejecución de un programa, y analizar el valor de las variables del mismo. Cuando el programa se ejecuta bajo el control del depurador la ejecución se detiene automáticamente en el momento que se produce un error importante. También es posible detener la ejecución en un punto del programa elegido por el programador, al cual se le denomina punto de ruptura (*breakpoint*). Una vez que el programa está detenido, se puede visualizar el valor actual de las variables y datos del programa.

Para depurar un proyecto en Code::Blocks es necesario que la configuración activa sea la de desarrollo. La depuración del proyecto se inicia mediante la opción “Start” del menú “Debug” o haciendo clic en el botón . Si durante la ejecución se produce algún error, la línea donde se haya detectado el error aparecerá marcada en el editor de Code::Blocks, con un triángulo amarillo.





Estableciendo puntos de ruptura

Si ya se tiene una idea de donde puede estar el error, se puede usar un punto de ruptura para detener la ejecución del programa en una instrucción concreta. Para añadir un punto de ruptura, se sitúa el cursor en la línea en cuestión y se selecciona la opción “Toggle breakpoint” del menú “Debug”, o simplemente se hace clic con el botón derecho del ratón en el lateral izquierdo de la línea, y se elige la opción “Add breakpoint”. Entonces la línea queda marcada con un círculo color rojo. Después de establecer los puntos de ruptura, y antes de iniciar la depuración, es recomendable reconstruir de nuevo la aplicación mediante la opción “Rebuild” del menú “Build”.

Cuando se inicia la depuración, la ejecución del programa se detiene en el primer punto de ruptura, lo que se indica con un triángulo amarillo en la línea de ruptura.

Ejecutando paso a paso el programa

Una vez detenida la ejecución a mitad del proceso de depuración, el programa puede ejecutarse paso a paso con los botones del menú de depuración:

-  **Next line** - El depurador ejecuta hasta la siguiente línea de código del programa.
-  **Step into** - El depurador ejecuta hasta la siguiente instrucción del programa; y si la instrucción actual es una función, el depurador ejecuta hasta la primera instrucción del código de la función.
-  **Step out** - El depurador ejecuta hasta la siguiente instrucción del programa después de terminar la función actual (sube de nivel). Nótese que esta opción no debe utilizarse si la función actual es la función `main()`.
-  **Run to cursor** - El depurador continúa la ejecución del programa hasta detenerse en la línea donde se encuentra situado el cursor.

Visualizando el valor de variables


Cuando el programa está detenido a mitad de su depuración se puede consultar el valor de las variables. Esto permite comprobar que tienen el valor que esperamos, y si no, poder encontrar dónde está el error.

En el menú “Debug” está la opción “Debugging Windows” en la que se puede seleccionar que aparezca la ventana “Watches”. Esta ventana flotante se puede arrastrar a cualquier parte del interfaz gráfico de Code::Blocks para dejarla fija. En ella es posible escribir el nombre de una variable, o incluso una expresión completa, para ver su valor actual. Además, en el menú contextual que aparece al hacer clic con el botón derecho del ratón sobre el nombre de una variable en el código fuente, se puede seleccionar la opción “Watch” y la variable se añade a la ventana “Watches”

La pila de llamadas

Algunas veces resulta útil saber desde qué función se ha llamado a aquella en la que se ha detenido la ejecución del programa durante su depuración. La ventana “Call stack” muestra esta información, y es posible activarla desde la opción “Debugging Windows” del menú “Debug”.

Terminando la depuración

Es muy importante acabar la depuración antes de volver a intentar compilar el programa. De no hacerlo, Code::Blocks suele bloquearse y la única solución pasa por reiniciarlo. La depuración se acaba mediante la opción “Stop debugger” del menú “Debug” o haciendo clic en el botón .

Anexo B. Introducción a los gráficos en C

Para programar los aspectos gráficos del proyecto de programación a realizar en las prácticas de la asignatura se debe utilizar el módulo formado por los ficheros *Pantalla.h* y *Pantalla.c* que se encuentra en la sección de recursos del aula virtual. Este módulo facilita el uso de la biblioteca [SDL](#) y, aunque sólo permite utilizar parte de su funcionalidad, es suficiente para hacer el proyecto de programación. Para poder usar el módulo Pantalla, la biblioteca SDL debe estar instalada en el sistema.

Dibujando en la pantalla con las funciones básicas

Para poder usar el módulo en un proyecto, lo primero que se debe hacer es añadir los ficheros *Pantalla.h* y *Pantalla.c* al proyecto e incluir el fichero de cabecera *Pantalla.h* en los ficheros donde se vayan a utilizar sus funciones.

A continuación, se muestra cómo usar las distintas funciones del módulo empezando por un ejemplo básico: escribir un programa en el que se cree una ventana y se dibuje en ella un rectángulo. El programa se compondrá de un fichero llamado *main.c* junto con los ficheros *Pantalla.h* y *Pantalla.c*. El fichero *main.c* tendrá el siguiente código:

```
// Fichero main.c

#include "Pantalla.h"

int main( int argc, char *argv[] ) {
    Pantalla_Crea( "Ejemplo 1", 640,480 );
    Pantalla_DibujaRellenoFondo( 255,255,255, 255 );
    Pantalla_ColorTrazo( 255,0,0, 255 );
    Pantalla_ColorRelleno( 0,255,0, 255 );
    Pantalla_DibujaRectangulo( 300,220, 80,40 );
    Pantalla_Actualiza();
    Pantalla_Espera( 5000 );
    Pantalla_Libera();
    return 0;
}
```

Como muestra el programa anterior, la primera tarea a realizar es crear la ventana donde se va a mostrar el dibujo. Para ello se usa la función *Pantalla_Crea* que recibe como parámetro el título de la ventana, su anchura y su altura en pixels.

A continuación, se ha usado la función *Pantalla_DibujaRellenoFondo* para borrar toda la ventana dejándola del color que se indique con sus cuatro parámetros. El color se especifica según el modelo de color [RGB](#), es decir, mediante sus componentes roja, verde, azul, y además, el grado de transparencia (*factor alpha*). Los cuatro valores deben ser números enteros en el intervalo 0...255 siendo 0 la mínima intensidad y 255 la máxima para los componentes RGB y siendo 0 totalmente transparente y 255 totalmente opaco para el cuarto parámetro. En este caso se rellena el fondo con un color blanco, totalmente opaco.

Después se han usado las funciones `Pantalla_ColorTrazo` y `Pantalla_ColorRelleno` para establecer, respectivamente, el color de la línea que delimita las figuras (en este caso el rectángulo) y el color del interior de la figura. Para indicar el color en ambas funciones se usa el mismo formato (modelo RGB) que en la función `Pantalla_DibujaRellenoFondo`.

Una vez hecho esto, se ha dibujado la figura que se pretendía, en este ejemplo, un único rectángulo. Para conseguirlo se ha usado la función `Pantalla_DibujaRectangulo` indicando con los dos primeros parámetros la coordenada de la esquina superior izquierda del mismo y con los dos últimos su anchura y altura. Los cuatro parámetros se especifican en pixels.

Se debe tener en cuenta que la esquina superior izquierda de la ventana se corresponde con la coordenada (0,0) y la esquina inferior derecha con la coordenada (ANCHO-1, ALTO-1), siendo ANCHO y ALTO los valores usados al crear la ventana.

Si se ejecutara el programa hasta aquí se podría comprobar que, aunque la ventana aparece, y tiene el título y el tamaño adecuados, el fondo de la misma no se rellena de blanco ni aparece ningún rectángulo. La razón es que todo se ha dibujado en la memoria principal, y para poder verlo hay que copiar esos datos a la memoria asociada a la ventana en la tarjeta gráfica llamando a la función `Pantalla_Actualiza`.

Además, antes de usar la función `Pantalla_Libera` para cerrar la ventana y poder terminar el programa, se ha usado la función `Pantalla_Espera`. Esta función detiene la ejecución del programa tantos milisegundos como se le indique. En este caso 5000 ms que equivalen a 5 s.

Deteniendo la ejecución se consigue que el usuario vea el dibujo que se acaba de hacer. De no hacerlo no se vería nada, pues la ventana se cerraría inmediatamente.

Además de rectángulos también se pueden dibujar puntos, círculos, triángulos, segmentos de línea y texto. Los parámetros de estas funciones se pueden consultar en la documentación del módulo que se encuentra en el aula virtual, en formato html.

Por último, el Tipo de Datos `Imagen` y las funciones `Pantalla_ImagenLee`, `Pantalla_DibujaImagen` y `Pantalla_ImagenLibera` permiten trabajar con imágenes guardadas en ficheros en formato BMP sin compresión. En primer lugar, se lee la imagen desde el fichero y el valor devuelto por `Pantalla_ImagenLee` se almacena en una variable de tipo `Imagen`.

Al abrir la imagen se puede indicar a través del segundo parámetro de la función si se desea que el color del pixel situado en la esquina superior izquierda se interprete como transparente.

Luego, usando esa variable se dibuja la imagen en las coordenadas deseadas. Finalmente, cuando ya no se necesita usar la imagen, hay que liberar la memoria que ocupa usando la función `Pantalla_ImagenLibera`.

Animaciones

A continuación, se detalla cómo hacer animaciones. Una animación es una secuencia de [frames](#) o imágenes que se van mostrando una tras otra y cuyo contenido varía de una a otra creando sensación de movimiento.

Para hacer una animación se escribirá el código para crear la ventana y, a continuación, se debe incluir un bucle en cuyo interior se dibujará la figura en una posición determinada por el valor de una variable. Esta variable, que debe estar declarada fuera del bucle, irá cambiando de valor en cada iteración. Antes de terminar la iteración, para dejar que el ojo humano capte el frame recién pintado, se debe incluir una pausa. Y al inicio de cada iteración se borra lo que se hubiera dibujado en la iteración anterior.

El siguiente ejemplo muestra cómo hacer una animación que simule un rectángulo moviéndose horizontalmente por la pantalla.

```
// Fichero Ejemplo2.c
#include "Pantalla.h"

int main( int argc, char *argv[] )
{
    Pantalla_Crea( "Ejemplo 2", 640,480 );
    Pantalla_ColorTrazo( 255,0,0, 255 );
    Pantalla_ColorRelleno( 0,255,0, 255 );
    int x = 100;
    while ( x < 300 ) {
        x = x + 5;
        Pantalla_DibujaRellenoFondo( 255,255,255, 255 );
        Pantalla_DibujaRectangulo( x,220, 80,40 );
        Pantalla_Actualiza();
        Pantalla_Espera( 40 );
    }
    Pantalla_Libera();
    return 0;
}
```

Nótese que, en este ejemplo, la variable *x* también se usa para determinar cuándo termina el bucle. En el interior del bucle, además de actualizar el valor de *x* incrementando su valor en cada frame de la animación, se dibuja el fondo de la pantalla y después el rectángulo en la posición que indique *x*. Si no se redibujara el fondo de la pantalla en cada iteración el rectángulo se movería, pero quedaría un rastro rojo en el lateral izquierdo. El tiempo de espera entre frames está relacionado con la velocidad de la animación. Por ejemplo, si se esperasen 10 ms tras pintar cada frame se estarían dibujando algo menos de 100 fps (*frames per second*). No llegaría a 100 porque el resto de acciones incluidas en el bucle también consumen tiempo. Cuanto mayor sea el número de frames por segundo más fluida será la animación, pero más se cargará la CPU. Para conseguir una animación fluida sin sobrecargar demasiado al procesador se suele utilizar un tiempo de espera de 40ms.

Aplicaciones interactivas

Finalmente se muestra un ejemplo de aplicación interactiva, es decir, una aplicación que muestra una animación en la que se tenga en cuenta lo que el usuario haga con el ratón y el teclado. En el siguiente ejemplo, el rectángulo se dibuja en el centro de la pantalla y cuando se pulsan las teclas flecha izquierda y derecha, cambia de posición.

```
// Fichero Ejemplo3.c

#include "Pantalla.h"
#include <stdio.h>

int main( int argc, char *argv[] ) {
    Pantalla_Crea("Ejemplo 3", 640,480);
    Pantalla_ColorTrazo(255,0,0, 255);
    int x = 100;
    while ( Pantalla_Activa() ) {
        if (Pantalla_TeclaPulsada(SDL_SCANCODE_RIGHT)) {
            x = x + 5;
        }
        if (Pantalla_TeclaPulsada(SDL_SCANCODE_LEFT)) {
            x = x - 5;
        }
        Pantalla_DibujaRellenoFondo( 255,255,255, 255 );
        Pantalla_DibujaRectangulo( x, 220, 80,40 );
        Pantalla_Actualiza();
        Pantalla_Espera(40);
    }
    Pantalla_Libera();
    return 0;
}
```

Obsérvese cómo ahora la condición que controla el bucle principal de la animación es el resultado de ejecutar la función `Pantalla_Activa`. Es necesario ejecutar dicha función al principio de cada iteración para que las acciones del usuario (uso del ratón o pulsaciones del teclado) puedan ser procesadas correctamente. Esta función, además de actualizar dicha información, devuelve 1 si la ventana sigue activa ó 0 si ha sido cerrada por el usuario. Por lo tanto, el bucle acabará cuando se cierre la ventana.

Además, se ha utilizado la función `Pantalla_TeclaPulsada` para determinar si el usuario ha pulsado o no una tecla. Para indicar la tecla se utilizan los códigos de tecla (SCANCODES) predefinidos como constantes en la biblioteca SDL.

Una lista con todos los códigos de tecla se puede encontrar en la página web https://wiki.libsdl.org/SDL_Scancode. En este ejemplo, al pulsar las teclas con flecha izquierda y derecha se modifica el valor de la variable que determina la posición del rectángulo.

Así pues, para escribir un programa consistente en una animación interactiva, la estructura recomendada es la siguiente:

```
// Estructura básica para hacer animaciones interactivas

#include "Pantalla.h"

int main( int argc, char *argv[] ) {
    // Creación y configuración de la pantalla
    Pantalla_Crea(...);
    ...

    // Declaración de variables que mantengan el estado
    // de la animación
    ...

    // Bucle principal de la animación
    int terminado = 0;
    while ( Pantalla_Activa() && !terminado ) {

        // Detección de los eventos producidos por el usuario
        if ( Pantalla_TeclaPulsada(...) ) {...}
        if ( Pantalla_TeclaPulsada(...) ) {...}

        // Modificación del estado de la simulación teniendo
        // en cuenta los eventos previamente detectados
        ...

        // Dibujo de los elementos de la simulación en el frame
        ...

        // Actualizar variable terminado
        if ( <condición de terminación> ) terminado = 1;

        // Mostrar en la ventana el frame actual
        Pantalla_Actualiza();

        // Espera entre frames
        Pantalla_Espera( 40 );
    }
    // Liberar la pantalla
    Pantalla_Libera();
    return 0;
}
```

Nótese que para hacer una animación interactiva que tenga en cuenta la pulsación de diferentes teclas se deberá poner una sentencia `if` para cada una de ellas.

También es posible conocer la posición del ratón y el estado de sus botones usando las funciones: `Pantalla_RatonBotonPulsado` y `Pantalla_RatonCoordenadas`, cuyos parámetros pueden consultarse en la documentación.

Anexo C. Documentación de software con Doxygen

Al compilar un fichero fuente (.c) se crea el fichero objeto (.o) que puede ser utilizado por cualquier otro módulo. Para que otro módulo utilice las funciones definidas en este fichero sólo necesitará los ficheros .o y .h. Pero el programador que lo use necesitará la documentación de todas las funciones, para saber cómo usarlo.

En este apartado se describe una herramienta, denominada *Doxygen* (página oficial: <http://www.stack.nl/~dimitri/doxygen/>), que proporciona una forma muy sencilla de generar automáticamente dicha documentación. Doxygen es una herramienta de documentación para C y C++, y otros lenguajes de programación.

Para obtener una documentación con esta herramienta, primero hay que incluir comentarios en el propio código. Doxygen lee e interpreta los comentarios que empiezan por `/**`. Con ellos genera la documentación interpretando una serie de comandos específicos y relacionándolos con las definiciones de Tipos de Datos y funciones situadas a continuación.

Existen muchos comandos, pero los siguientes son los imprescindibles para documentar las prácticas de la asignatura:

- `\file` Nombre del fichero que se está documentando.
- `\brief` Breve descripción del módulo, función o Tipo de Datos que se está documentando.
- `\author` Información sobre los autores.
- `\version` versión del software.
- `\pre` Precondiciones de la función que se está documentando.
- `\param` Significado de un parámetro de la función que se está documentando.
- `\return` Valores devueltos por la función que se está documentando.

Para las prácticas de la asignatura se deben documentar únicamente los ficheros .h y el módulo principal (el módulo que contenga la función `main`).

Al principio de cada fichero del que se quiera generar su documentación debe añadirse un comando `\file` indicando el nombre del fichero.

En los ficheros .h cada Tipo de Datos y función debe tener un comentario que incluya un comando `\brief`. Además, para las funciones también se usarán los comandos `\pre`, `\param` y `\return` cuando corresponda. Si una función tiene varios parámetros, cada uno se documenta con un comando `\param` independiente.

En los comandos `\brief` de las operaciones de los TDAs no deben describirse aspectos relativos a la implementación interna, sino únicamente el aspecto público observable. En otras palabras, debe describirse el qué se hace pero no el cómo.

Y en el módulo principal debe incluirse el comando `\mainpage` con una descripción general de la aplicación.

Uso de Doxygen desde Code::Blocks

Code::Blocks permite configurar y ejecutar Doxygen. Para configurar Doxygen primero hay que seleccionar la opción “Extract documentation” del menú “DoxyBlocks”. Una vez hecho, se ejecuta la aplicación Doxywizard seleccionándola en el mismo menú.

La documentación debe generarse en español y en formato adecuado al lenguaje de programación C. No deben aparecer las estructuras de datos privadas, es decir, las que se definen en los módulos de los TDAs, y hay que cuidar que las tildes salgan correctamente.

De las múltiples opciones de configuración, las más importantes para conseguir todo lo dicho anteriormente son las siguientes:

- En la pestaña “Wizard”, tema “Mode” hay que marcar la opción “Optimize for C or PHP output”.
- En la pestaña “Expert”, tema “Project” hay que elegir “Spanish” en la opción “OUTPUT_LANGUAGE”.
- En la pestaña “Expert”, tema “Input” aparece la lista de ficheros sobre los que se va a generar la documentación. En ella hay que conseguir que sólo aparezcan los ficheros de cabecera del proyecto y el fichero con la función main.
- Si los acentos salen mal, la solución puede estar en la pestaña “Expert”, dentro de “Input”, en la opción INPUT_ENCODINGS. Las dos alternativas más habituales son ISO-8859-1 y UTF-8. Dependiendo de la codificación de los ficheros de código fuente se usa una u otra. Normalmente, los ficheros creados en Windows usan la codificación ISO-8859-1, y los creados en Linux o Mac la UTF-8.

Tras elegir las opciones deseadas se llega a la pestaña “Run” en la que pulsando el botón “Run Doxygen” se genera la documentación que puede verse pulsando el botón “Show HTML output”.

Si no se ha cambiado la configuración que aparece por omisión, los ficheros de la documentación se habrán generado en una carpeta llamada “html” que se habrá creado dentro de la carpeta del proyecto. En ella, el fichero “index.html” es el que da acceso a la documentación, y puede visualizarse con cualquier navegador web.

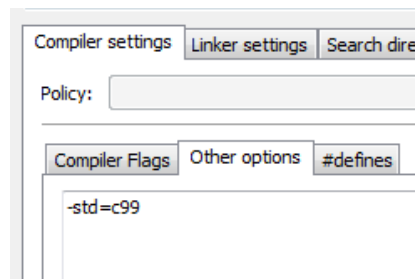
Anexo D. Instalación de Code::Blocks, SDL y Doxygen

Aunque es posible instalar cualquier versión de Code::Blocks, SDL y Doxygen, se recomienda instalar las mismas versiones que hay instaladas en los laboratorios de la Facultad. Los ficheros necesarios para instalar en Windows se pueden descargar desde la web del Centro de Cálculo:

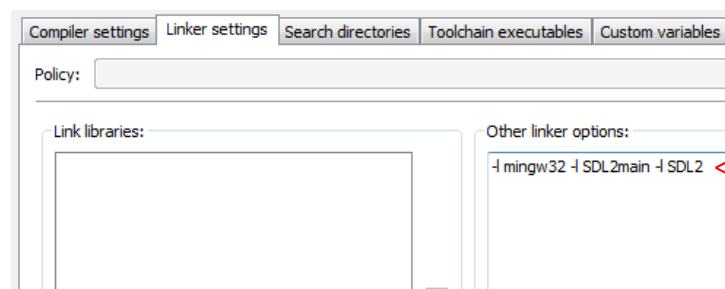
<http://wiki.inf.um.es/codeblocks/>

Sigue estas instrucciones **al pie de la letra** para instalar las tres herramientas. Es importante no confundir “contenido de carpeta” con “carpeta completa”:

1. Descarga y ejecuta el instalador de Code::Blocks: **codeblocks-13.12mingw-setup.exe**
2. Descarga y descomprime la librería SDL2: **SDL2-devel-2.0.3-mingw.tar.gz**
3. Copia el archivo **SDL2.dll** que hay en la carpeta **SDL2-2.0.3\i686-w64-mingw32\bin** a las carpetas **C:\Windows\System32** y **C:\Windows\SysWOW64**
4. Copia todo el contenido de la carpeta **SDL2-2.0.3\i686-w64-mingw32\lib** a la carpeta **C:\Archivos de programa (x86)\Codeblocks\Mingw\lib**
5. Copiar la carpeta **SDL2** que hay en **SDL2-2.0.3\i686-w64-mingw32\include** a la carpeta **C:\Archivos de programa (x86)\Codeblocks\Mingw\include**
6. Arranca Code::Blocks y haz clic en la opción “Compiler” del menú “Settings”.
7. En la pestaña “Compiler Settings” escribe `-std=c99` en la pestaña “Other options”.



8. En la pestaña “Linker Settings” escribe `-l mingw32 -l SDL2main -l SDL2` en el cuadro de texto “Other linker options”.



9. Verifica que en la pestaña “Toolchain executables” la ruta que aparece lleva al compilador MinGW incluido en Codeblocks: **C:\Program Files (x86)\CodeBlocks\MinGW** o similar.
10. Descarga y ejecuta el instalador de Doxygen: **doxygen-1.8.8-setup.exe**

Finalmente, si al compilar un programa que use gráficos aparece el error “winapifamily.h: No such file or directory” la solución consiste en sustituir el fichero **SDL_platform.h** que hay en **C:\Program Files (x86)\CodeBlocks\MinGW\include\SDL2** por el que hay en la web del Centro de cálculo.

<http://wiki.inf.um.es/codeblocks/>