

PRÁCTICAS DE PROGRAMACIÓN en C±

José A. Cerrada Somolinos
Manuel E. Collado Machuca
Ismael Abad Cardiel
Rubén Heradio Gil



Editorial universitaria
Ramón Areces

UNED

PRÁCTICAS
DE PROGRAMACIÓN
en C[±]

JOSÉ A. CERRADA SOMOLINOS

Catedrático de Lenguajes y Sistemas Informáticos (UNED)

MANUEL E. COLLADO MACHUCA

Catedrático de Lenguajes y Sistemas Informáticos (UPM)

ISMAEL ABAD CARDIEL

Profesor Colaborador de Lenguajes y Sistemas Informáticos (UNED)

RUBÉN HERADIO GIL

Profesor Ayudante de Lenguajes y Sistemas Informáticos (UNED)

PRÁCTICAS DE PROGRAMACIÓN en C±



Editorial universitaria
Ramón Areces



Índice general

1	Introducción	1
1.1	Prácticas de la asignatura	3
1.2	Organización del texto	3
1.3	Cómo utilizar el libro	6
2	Instalación del entorno de prácticas	9
2.1	Code::Blocks ^{C±}	9
2.2	Requisitos de Code::Blocks ^{C±}	9
2.3	Instalación del entorno	10
3	El entorno a vista de pájaro	13
3.1	Cómo poner en marcha Code::Blocks ^{C±}	13
3.2	Cómo escribir un programa	13
3.3	Cómo generar un programa ejecutable	15
3.4	Code::Blocks ^{C±} al rescate	17
3.4.1	Errores de precompilación C±	18
3.4.2	Errores de compilación	19
3.4.3	Errores de ejecución	20
4	Primera práctica: impresión de datos personales	25
4.1	Enunciado de la práctica	25
4.2	Autocorrección de la práctica	27
5	Conceptos básicos del entorno	33
5.1	Estructura general del entorno	34
5.2	Gestión de archivos	36
5.2.1	Creación de nuevos archivos	36
5.2.2	Almacenamiento de archivos	39
5.2.3	Opciones para abrir archivos almacenados	40

5.3 Opciones de edición	42
5.3.1 Introducción	42
5.3.2 Acciones de edición de texto	46
5.3.3 Apariencia del texto	49
5.3.4 Autocompletado de texto	49
5.3.5 Balanceo de delimitadores de texto	50
5.3.6 Otros comentarios básicos	51
5.4 Generación de archivos ejecutables	52
5.5 Análisis de programas C±	57
5.6 Tratamiento de errores en C±	62
5.6.1 Errores relacionados con la función <code>main</code>	62
5.6.2 Errores relacionados con el uso de expresiones	63
5.6.3 Errores relacionados con la declaración de constantes	65
5.6.4 Errores relacionados con la declaración de subprogramas	65
5.6.5 Errores relacionados con la declaración de variables .	67
5.6.6 Errores relacionados con el uso de <code>for</code> , <code>if</code> , <code>switch</code> , <code>while</code> y <code>dowhile</code>	68
5.6.7 Otros errores	68
6 Segunda práctica: rombos	71
6.1 Enunciado de la práctica	71
6.2 Metodología de desarrollo del algoritmo	74
6.2.1 División del rombo en triángulos	74
6.2.2 División de un triángulo en líneas	77
6.3 Autocorrección de la práctica	78
7 Depuración de programas	85
7.1 Conceptos básicos sobre la depuración de programas	86
7.2 Ejemplo de depuración simple con mensajes	87
7.3 Operaciones de depuración	90
7.4 Ventanas de depuración e información	94
7.5 Ejemplo de depuración	100
8 Tercera práctica: calendario	105
8.1 Enunciado	105
8.2 Metodología de desarrollo del algoritmo	106
8.3 Autocorrección de la práctica	107

8.3.1 Proceso de autocorrección	107
8.3.2 Casos de prueba	111
8.4 Entrega telemática de las prácticas	113
9 Manejo de proyectos con el entorno	115
9.1 Un simulador del juego “Piedra, Papel o Tijera”	115
9.2 Solución del ejemplo	116
9.3 Escritura de la solución con <i>Code::Blocks^{C±}</i>	122
9.4 Organización de un proyecto en MS Windows	129
9.5 Menús para la gestión de proyectos	130
10 Cuarta práctica: programación modular	133
10.1 Normativa	133
10.2 Ejemplo de enunciado de la cuarta práctica	134
11 Esto es sólo el principio	139
11.1 Complementos de <i>Code::Blocks^{C±}</i>	139
11.2 Complemento de búsqueda <i>ThreadSearch</i>	141
11.3 Esquemas de edición	142
11.4 Navegación por el código	144
11.5 Configuración avanzada del entorno	146
Bibliografía	151

Índice de cuadros

5.1 Apariencia del texto en Code::Blocks ^{C±}	49
5.2 Errores relacionados con la función main	63
5.3 Errores relacionados con el uso de expresiones	64
5.4 Errores relacionados con la declaración de constantes	65
5.5 Errores relacionados con la declaración de subprogramas . .	66
5.6 Errores relacionados con la declaración de variables	67
5.7 Errores relacionados con el uso for , if , switch , while y dowhile	68
5.8 Otros errores comunes en C±	70

Capítulo 1

Introducción

La importancia de C y sus sucesores (C++, C# y Java) en el desarrollo de programas es incuestionable. La figura 1.1 muestra el ranking que TIOBE SOFTWARE publicó en marzo de 2010. Según este ranking, el lenguaje de programación más utilizado hoy día es Java (17,5% de los proyectos informáticos), seguido de C (17,28%). C++ y C# ocupan la cuarta y la sexta posición (9,61% y 4,23% respectivamente). Es decir, el 46,62% de las aplicaciones informáticas se realizan con algún lenguaje de “la familia C”.

Lamentablemente, el coste de la flexibilidad y potencia de estos lenguajes es su enorme complejidad. Dada la dificultad de su aprendizaje, consideramos que estos lenguajes no son el mejor vehículo para iniciarse en la programación. Los autores del presente libro, hemos hecho un esfuerzo por seleccionar el núcleo de la familia de lenguajes C que creemos más adecuado pedagógicamente para los alumnos de la asignatura de *Fundamentos de Programación* que se imparte en el primer año de Grado en Informática de la Universidad Nacional de Educación a Distancia (UNED). El resultado de este esfuerzo es el lenguaje C±.

Este libro es el complemento práctico del texto básico de la asignatura (consulte la referencia bibliográfica [1] al final de este libro), donde se explica detalladamente el lenguaje C±.

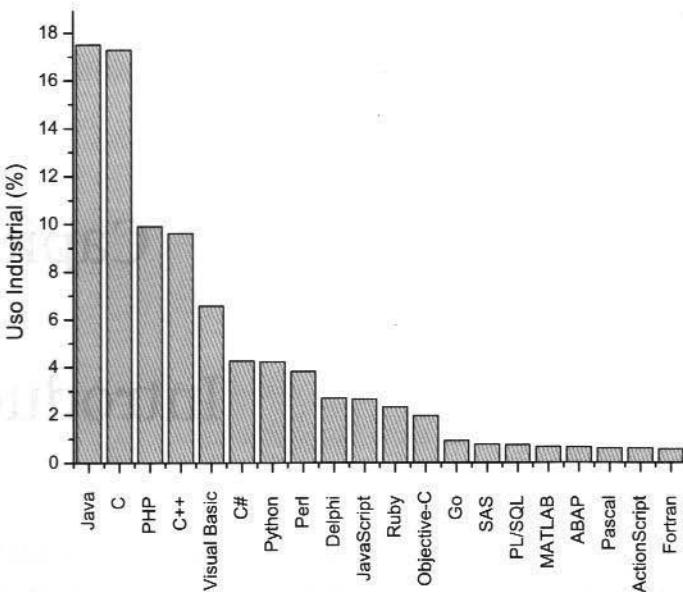


Figura 1.1: Uso industrial de los lenguajes de programación

Para dar soporte a los usuarios de C \pm , hemos creado una versión específica del entorno de programación Code::Blocks¹, al que nos referiremos como Code::Blocks $C\pm$, que facilita la edición, compilación y depuración de programas C \pm . El presente libro describe cómo instalar y sacar el máximo partido al nuevo entorno de programación para C \pm .

Aunque el libro puede utilizarse por cualquier persona que desee iniciarse en la programación de ordenadores, está especialmente orientado a alumnos de la UNED. Es decir, cuida de manera especial los aspectos específicos de la enseñanza a distancia, potenciando el autoaprendizaje del alumno. Los conceptos se introducen progresivamente, poco a poco, para que el alumno pueda ir avanzando a su ritmo. El texto propone tres prácticas de dificultad creciente, donde el alumno debe aplicar los conceptos teóricos de C \pm . Así, se refuerza lo aprendido y se motiva al alumno para querer aprender más. Además, se incluyen las pautas metodológicas para resolver las prácticas y otros problemas comunes de la

¹La versión original de Code::Blocks es código libre y puede obtenerse en [6].

programación. Por último, *Code::Blocks^{C±}* incluye la capacidad de corregir automáticamente las prácticas. De este modo, se logra una gran interactividad alumno-entorno que redunda en un proceso de autoaprendizaje más ágil.

Esperamos que nuestro esfuerzo contribuya a un sólido aprendizaje de los fundamentos de la programación imperativa y sirva de plataforma para profundizar en el lenguaje de programación C y en sus sucesores C++, C# y Java.

1.1. Prácticas de la asignatura

La asignatura de *Fundamentos de Programación* incluye:

1. Tres prácticas que se mantienen todos los cursos académicos y se corrigen de forma automática con *Code::Blocks^{C±}*.
2. Una última práctica que será diferente cada curso académico y se publicará en la web de la asignatura [5].

La figura 1.2 resume esquemáticamente la correspondencia entre las prácticas propuestas en este libro y los temas del texto básico [1] de la asignatura.

1.2. Organización del texto

El libro se estructura en los siguientes capítulos:

- **Capítulo 2: Instalación del entorno de prácticas.** Describe la instalación de *Code::Blocks^{C±}* y da una visión panorámica de su uso.
- **Capítulo 3: El entorno a vista de pájaro.** Este capítulo va dirigido a los lectores que nunca han utilizado un entorno de desarrollo de software y describe, a través de un ejemplo, un paseo rápido por las funcionalidades de *Code::Blocks^{C±}* que se utilizan con mayor frecuencia cuando se escriben programas C±.

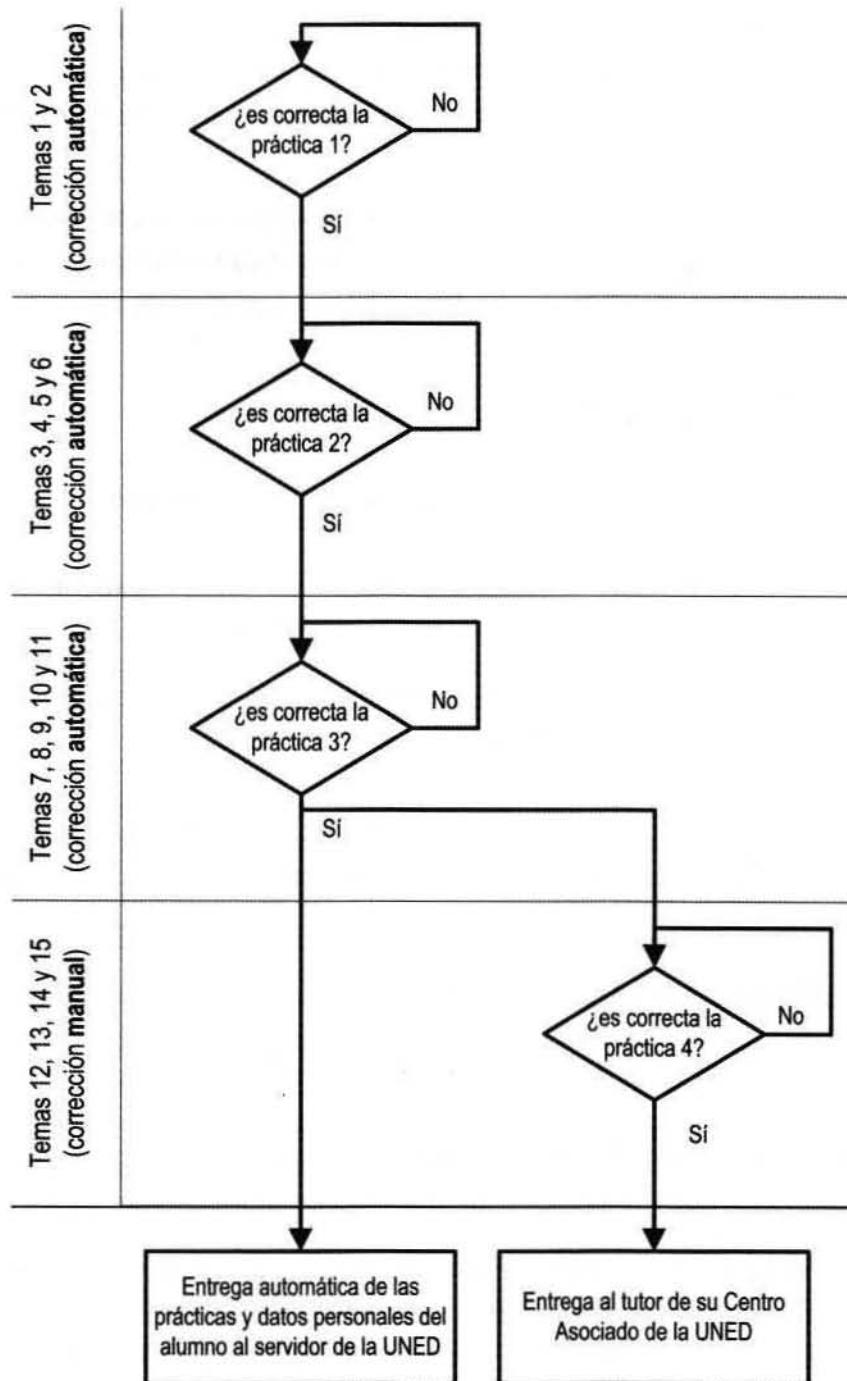


Figura 1.2: Relación entre las prácticas y los temas del texto básico de la asignatura [1]

- **Capítulo 4: Primera Práctica: “Datos Personales”.** Incluye el enunciado de la primera práctica. Se trata de un problema sumamente sencillo que tiene como objetivo familiarizar al alumno con `Code::BlocksC±`. Además, el capítulo describe el proceso de corrección automática de errores para la primera práctica.
- **Capítulo 5: Conceptos básicos del entorno.** Explica los fundamentos de `Code::BlocksC±`, cubriendo las siguientes áreas:
 1. Gestión de archivos.
 2. Edición de programas.
 3. Compilación de programas C±.
 4. Tratamiento de errores.
 5. Ejecución de programas.
- **Capítulo 6: Segunda práctica: “Rombos”.** De dificultad creciente respecto a la primera práctica, este capítulo propone otra práctica que exige el uso de nuevos elementos del lenguaje C±. En el capítulo se dan las pautas metodológicas para resolver la práctica, incidiendo en cuáles son las consecuencias negativas de no reflexionar suficientemente antes de codificar una solución y en cuál es la forma correcta de pensar para resolver problemas complejos.
- **Capítulo 7: Depuración de programas.** Como suele decirse “errar es humano”. Una estimación conservadora considera que se cometen 15 errores de programación por cada 1.000 líneas de código. No es de extrañar, por tanto, que cuando se comienza a programar por primera vez o se aprende un nuevo lenguaje se cometan muchos errores. Este capítulo describe cómo detectar y corregir errores de programación con `Code::BlocksC±`.
- **Capítulo 8: Tercera práctica: “Calendario”.** Incluye el enunciado de la tercera práctica, las pautas metodológicas sobre cómo resolver la práctica, una descripción de su proceso de corrección automática y sobre su entrega telemática al equipo docente de la UNED. Además,

con el objetivo de que el alumno reflexione sobre la variedad de problemas que contempla la práctica y aprenda los fundamentos de la prueba de programas, el capítulo incluye una breve introducción sobre la técnica de pruebas basada en la partición de clases de equivalencia.

- **Capítulo 9: Manejo de proyectos con el entorno.** Describe cómo implementar programas modulares utilizando proyectos.
- **Capítulo 10: Cuarta práctica: programación modular.** Tiene como objetivo que el alumno ejercite la descomposición modular de programas como herramienta para resolver problemas complejos. Frente al resto de las prácticas, esta última tiene las siguientes peculiaridades:
 1. Es diferente cada año. Los enunciados se publicarán en la web de la asignatura [5]. A modo de ejemplo, el capítulo incluye un enunciado de este tipo de práctica.
 2. No se corrige automáticamente, sino que el alumno deberá entregarla al tutor de su centro asociado para su evaluación.
- **Capítulo 11: Esto es sólo el principio.** Este manual trata de cubrir la prestaciones mínimas de Code::Blocks^{C±}para realizar programas. Sin embargo, los entornos de programación modernos suelen incluir muchas más posibilidades. Es interesante que el lector indague por su cuenta y experimente con las opciones disponibles. Este capítulo presenta algunas prestaciones adicionales de Code::Blocks^{C±}y resume los menús donde el lector puede iniciar sus propias investigaciones.

1.3. Cómo utilizar el libro

El libro se ha escrito con tres perfiles de lectores en mente:

1. Lector A. Estudiante que nunca ha utilizado un entorno de programación. Desea aprender progresivamente los fundamentos de C±

y de Code::Blocks^{C±}. Aconsejamos que estudie todos los capítulos según su orden de aparición en el libro.

2. Lector B. Estudiante que está familiarizado con otros entornos de programación (Visual Studio, Eclipse, etc). Desea centrar su estudio en C±. Creemos que puede abordar la resolución de las prácticas directamente. Por supuesto, si tiene dudas puntuales sobre Code::Blocks^{C±} puede dirigirse a los capítulos correspondientes.
3. Lector C. Estudiante que desea utilizar este libro como un manual de referencia de Code::Blocks^{C±}.

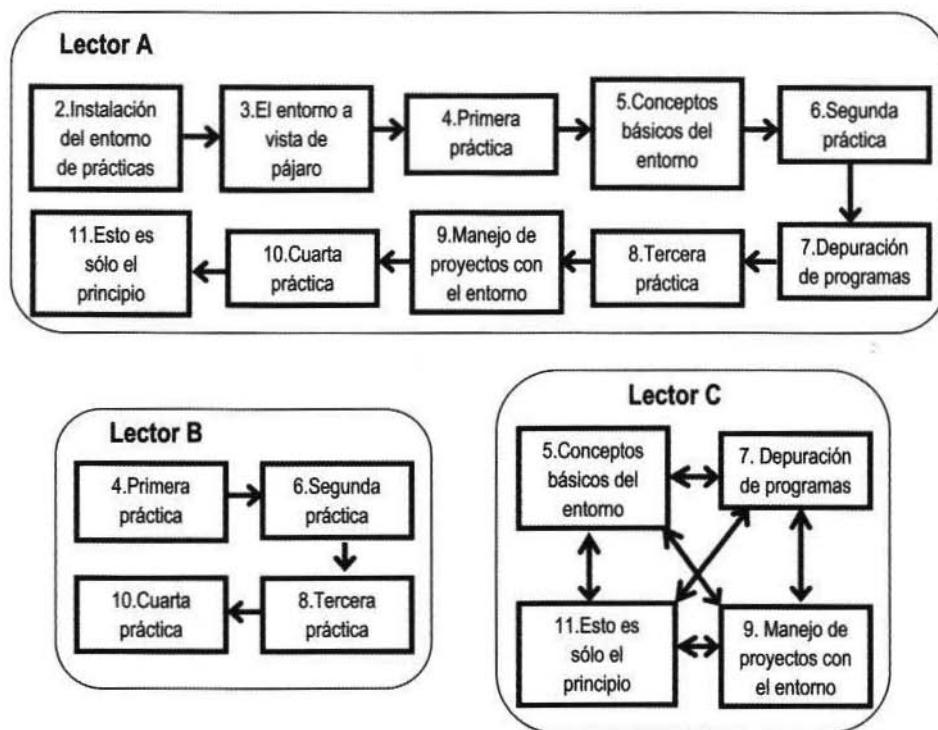


Figura 1.3: Distintos itinerarios de aprendizaje para los lectores del libro

Capítulo 2

Instalación del entorno de prácticas

ESTE capítulo describe los requisitos que debe cumplir nuestro ordenador para utilizar **Code::Blocks^{C±}**, así como los pasos necesarios para su instalación.

2.1. **Code::Blocks^{C±}**

Code::Blocks es un entorno integrado de desarrollo¹ orientado principalmente a los lenguajes C y C++. Además, soporta la integración de muchos otros lenguajes, como C# ó Python. Code::Blocks puede ejecutarse en diversos sistemas operativos (Microsoft Windows, Unix/Linux y Apple Mac) utilizando la librería de objetos gráficos multiplataforma wxWidget².

Code::Blocks^{C±}es una versión específica de Code::Blocks para C± desarrollada por los autores de este libro.

2.2. **Requisitos de Code::Blocks^{C±}**

Los requisitos necesarios para ejecutar **Code::Blocks^{C±}**son:

¹Este tipo de entornos suelen denominarse IDEs (*Integrated Development Environments* en inglés).

²La versión original de wxWidget es una librería C++ multiplataforma de código libre que puede obtenerse en [7].

1. Utilizar como sistema operativo Microsoft Windows. Se ha verificado el buen funcionamiento de **Code::Blocks^{C±}** con:
 - Los sistemas operativos Windows XP, Windows Vista y Windows 7 (incluyendo las versiones de finales del año 2009).
 - Las Arquitecturas x32 y x64.
2. 512 Mbytes de memoria RAM.
3. Al menos 200 Mbytes de espacio libre en el disco duro donde se instalará **Code::Blocks^{C±}**.
4. Para completar el proceso de autocorrección y entrega de las tres primeras prácticas es **imprescindible** tener el entorno con conexión a Internet.

2.3. Instalación del entorno

El programa de instalación del entorno está disponible en la web oficial de la asignatura [5]. Dicho programa está comprimido en el archivo **cmasmenos.exe**.

Guardaremos el archivo en una carpeta temporal de nuestro ordenador, por ejemplo en: **C:\instalacion_cmasmenos**

A continuación, descomprimimos el archivo haciendo doble clic sobre **cmasmenos.exe**. Aparecerá una ventana que nos indica que la descompresión va a realizarse en **C:\instalacion_cmasmenos**. Pulsamos **Instalar** (ver figura 2.1).

Finalizada la descompresión, ejecutaremos el archivo **instalar.exe**. Aparecerá la ventana de la figura 2.2³, donde podremos especificar las siguientes opciones de instalación:

- Carpeta de destino donde deseamos instalar **C±**.

³Las versiones más recientes de Windows nos pueden solicitar la confirmación de la ejecución del programa de instalación según la configuración que se utilice del sistema.

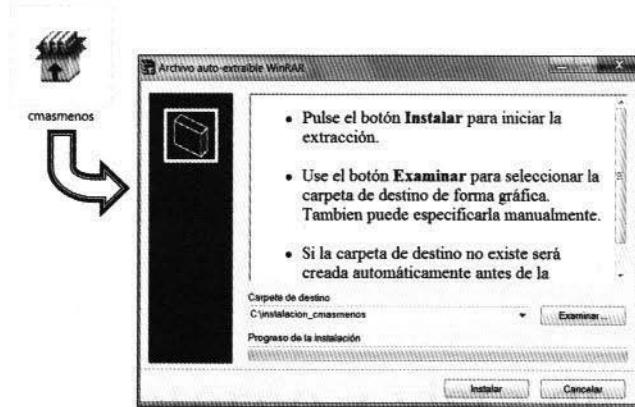


Figura 2.1: Descompresión del archivo de instalación

- ¿Deseamos que el instalador cree un acceso directo para el entorno en el escritorio de Windows?
- ¿Deseamos que se ejecute el entorno al terminar la instalación?

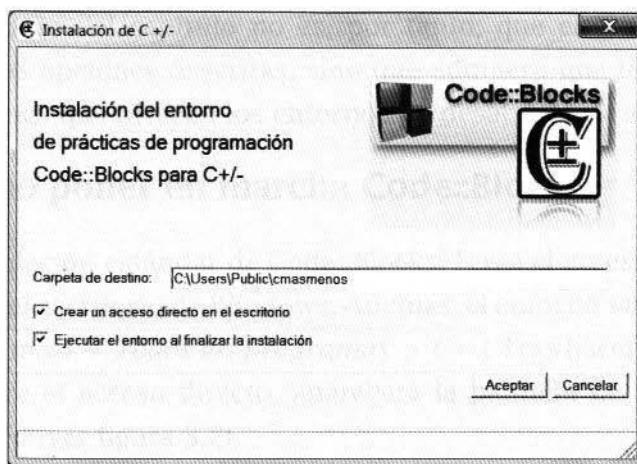


Figura 2.2: Instalación del entorno de prácticas

Pulsando **Aceptar** comenzará la instalación, que puede tardar unos minutos. Finalizada la instalación, podemos eliminar la carpeta temporal C:\instalacion_cmasmenos utilizando el explorador de Windows.

Capítulo 3

El entorno a vista de pájaro

ESTE capítulo va dirigido a los lectores que nunca han utilizado un entorno de desarrollo de software. El capítulo describe un paseo rápido por las funcionalidades de **Code::Blocks^{C±}** que se utilizan con mayor frecuencia cuando se escriben programas C±. Estas funcionalidades, y muchas otras, se explicarán en detalle en los próximos capítulos. El objetivo prioritario del capítulo no es, por tanto, que el alumno entienda totalmente las opciones descritas, sino que adquiera una idea general de las prestaciones que ofrecen los entornos de desarrollo de software.

3.1. Cómo poner en marcha **Code::Blocks^{C±}**

La instalación estándar de **Code::Blocks^{C±}** crea el acceso directo de la figura 3.1 en el escritorio de Windows. Además, el entorno también es accesible desde **Inicio → Todos los programas → C++**. Tras hacer doble clic con el ratón sobre el acceso directo, aparecerá la pantalla de bienvenida de **Code::Blocks^{C±}**(ver figura 3.2).

3.2. Cómo escribir un programa

La forma más sencilla de empezar a escribir un programa es pulsar sobre el ícono C±, que se encuentra en la esquina superior izquierda de la ventana de **Code::Blocks^{C±}**. La figura 3.3 resalta la ubicación de esta

opción. Al hacer clic sobre el icono C \pm , crearemos un nuevo fichero en blanco sobre el que escribiremos nuestro programa (ver figura 3.4).



Figura 3.1: Acceso directo de Code::Blocks^{C \pm} situado en el escritorio de Windows

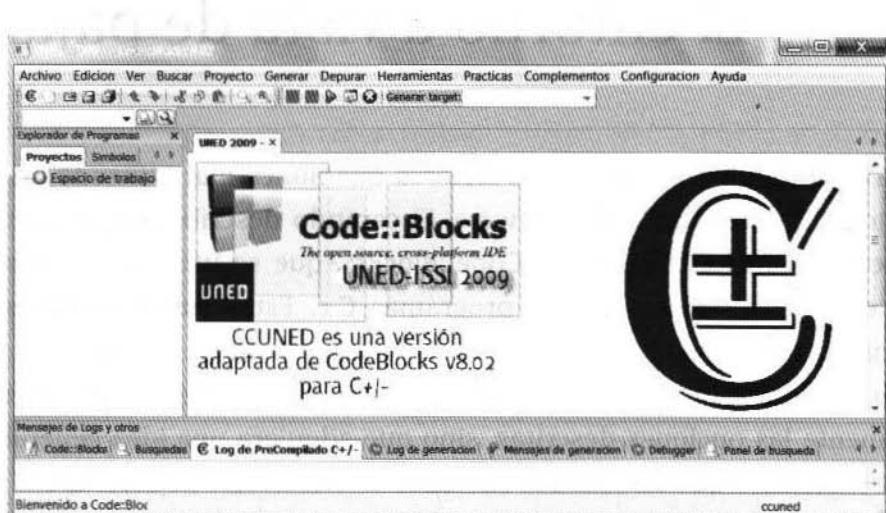


Figura 3.2: Ventana de bienvenida de Code::Blocks^{C \pm}

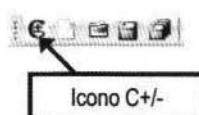


Figura 3.3: Creación de un fichero vacío

A continuación, escribiremos el código de nuestro primer programa en el área de edición, tal como se indica en la figura 3.5.

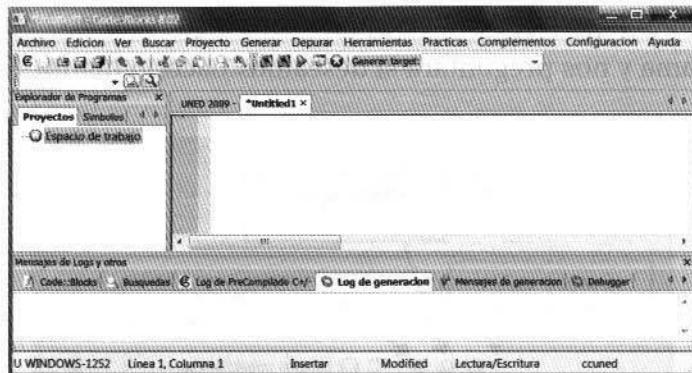


Figura 3.4: Área de edición de un nuevo programa

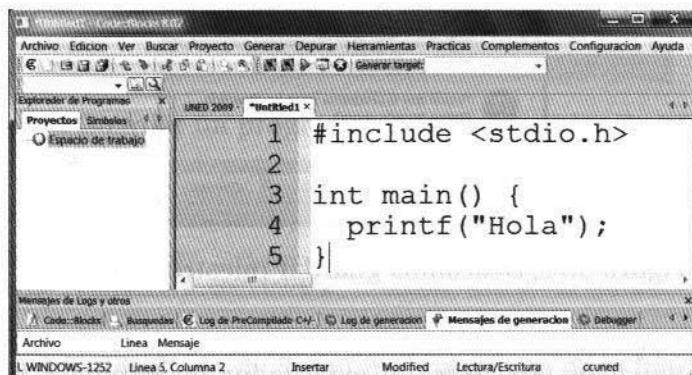


Figura 3.5: Código de un programa muy sencillo

3.3. Cómo generar un programa ejecutable

Un computador sólo entiende el *lenguaje máquina*, es decir, “programas escritos como largas sucesiones de ceros y unos”. Por tanto, es incapaz de ejecutar directamente el código C± que acabamos de escribir. Para ello, deberemos traducir nuestro programa C± a su correspondiente versión en código máquina. Si hacemos clic sobre el icono **Ejecutar** (ver figura 3.6), Code::BlocksC± hará la traducción por nosotros y solicitará al computador que ejecute el código máquina resultante.

En primer lugar, Code::Blocks^{C±} detectará que aún no hemos guardado el programa y nos preguntará si deseamos hacerlo (figura 3.7). Pulsaremos **Si**

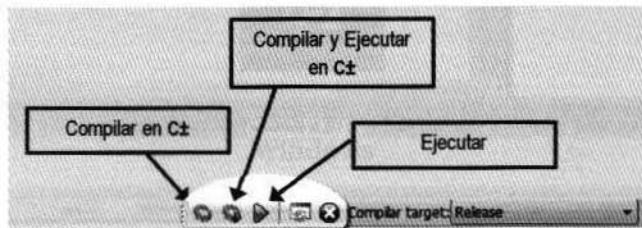


Figura 3.6: Generación y ejecución de programas en Code::Blocks^{C±}

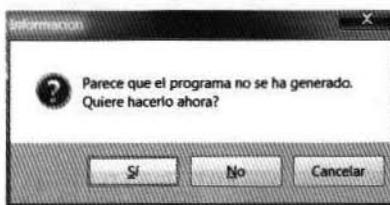


Figura 3.7: ¿Deseamos guardar nuestro programa?

A continuación, se abrirá el explorador de Windows para que indiquemos dónde queremos guardar el programa y qué nombre deseamos ponerle. En la figura 3.8, se indica que el programa debe llamarse *hola*, que es de tipo *Ficheros de C+/- Code::Blocks*, y que debe guardarse en **>Equipo>Disco Local (C:)>Mis Ejemplos**. Tras especificar la información anterior, pulsaremos el botón **Guardar**

Como nuestro programa es correcto, Code::Blocks^{C±} lo traducirá a código máquina. A continuación, se abrirá una nueva ventana donde se ejecutará el programa. El resultado se muestra en la figura 3.9:

- Se ha escrito *Hola* en la pantalla (línea 1 de la figura 3.9).
- La función *main* ha devuelto 0; este resultado numérico indica que la ejecución del programa ha terminado sin ningún contratiempo (línea 2 de la figura 3.9).

- El programa ha tardado en completar su ejecución 0,063 segundos (línea 2 de la figura 3.9).

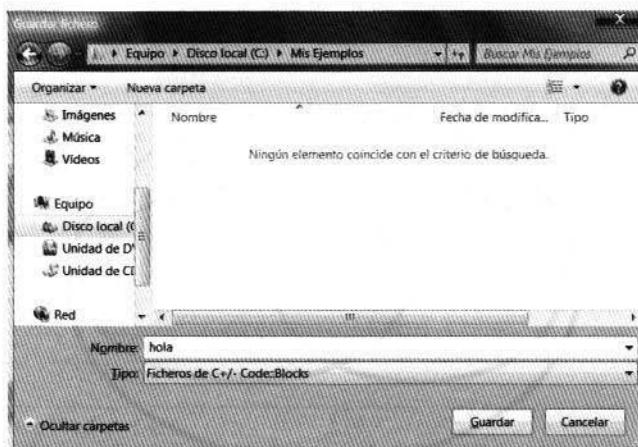


Figura 3.8: Guardar un programa con el explorador de Windows

```
1 Hola
2 El proceso devolvio 0 (0x0) tiempo de ejecucion : 0.063 s
3 Pulsar cualquier tecla para continuar.
```

Figura 3.9: Resultado de ejecutar el programa de la figura 3.5

3.4. Code::Blocks^{C±} al rescate

Escribir un programa informático de principio a fin sin cometer ningún error es prácticamente imposible. Además, a medida que aumenta el tamaño de los programas, crece exponencialmente la probabilidad de cometer errores, así como la cantidad de esfuerzo necesario para detectar los errores y corregirlos. Por ello, desde los comienzos de la informática se ha investigado sobre mecanismos que faciliten la detección automática de errores de programación. Afortunadamente, Code::Blocks^{C±}incorpora muchos de estos mecanismos, que se tratarán en detalle en los capítulos 5 y 7. En esta sección, nos limitaremos a introducir los tres grandes tipos de errores que pueden cometerse al escribir programas C± y cuáles son las prestaciones/limitaciones de Code::Blocks^{C±}para subsanarlos.

3.4.1. Errores de precompilación C \pm

C \pm es un subconjunto de C++ definido en el libro básico de la asignatura [1]. La figura 3.10 muestra la relación de C \pm con los lenguajes C y C++.

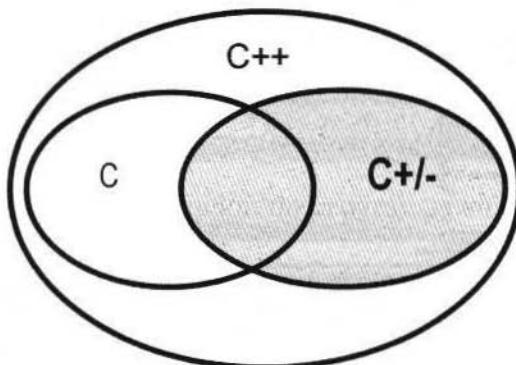


Figura 3.10: Relación entre los lenguajes C, C++ y C \pm

El primer filtro que Code::Blocks^{C \pm} pasa a nuestros programas es un precompilador que trata de buscar errores léxicos y gramaticales según las reglas de C \pm . Puesto que C \pm es más restrictivo que C++, un programa correcto en C++ puede tener errores en Code::Blocks^{C \pm} .

Por ejemplo, imaginemos que escribimos el programa incorrecto de la figura 3.11.

```
1 main() {
2     printf("Hola");
3 }
```

Figura 3.11: Versión errónea del programa hola

Al pulsar **Ejecutar**, en la pestaña *Log de precompilado C \pm* del área *Mensajes de Logs y otros*, aparecerá el mensaje de error de la figura 3.12. Si nos fijamos en la figura 3.13, donde se amplía el mensaje de error, vemos que nuestro fallo se produce en la Línea 1, donde “Falta int antes del main”. Puesto que C \pm exige que la función main siempre devuelva un valor entero, corregiremos el problema modificando la línea 1 (ver figura 3.14).

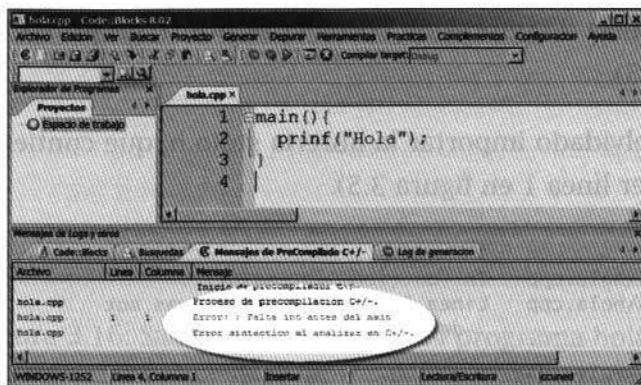


Figura 3.12: La traducción se ha interrumpido por un error de precompilación

Inicio de precompilador C++/-.
 C:\Mis Ejemplos\hola.cpp: Linea 1: Columna 1: Falta int antes del main
 Error sintactico al analizar en C++/-.

Figura 3.13: Detalle del error sintáctico mostrado en la figura 3.12

```

1 int main() {
2     printf("Hola");
3 }

```

Figura 3.14: Resolución del error notificado en la figura 3.13

3.4.2. Errores de compilación

Tras pasar el precompilador de C±, Code::Blocks^{C±} utiliza un compilador de C++ para garantizar que el programa satisface las restricciones léxicas, sintácticas y semánticas de C++.

Volviendo a nuestro ejemplo, si tratamos de ejecutar el programa de la figura 3.14, en la pestaña *Mensajes de generacion* del área *Mensajes de Logs y otros* aparecerá el nuevo mensaje de error de la figura 3.15. Dentro de la función main, concretamente en la línea 2 del programa, estamos usando un identificador que no ha sido declarado previamente: printf. El mensaje nos ayuda a detectar los siguientes dos problemas, resueltos en la figura 3.5:

1. La sentencia de impresión correcta es `printf`, no `prinf` (ver línea 4 en figura 3.5).
2. Hemos olvidado importar la librería `stdio.h`, que contiene la función `printf` (ver línea 1 en figura 3.5).

```
C:\Mis Ejemplos\hola.cpp  In function 'int main()':
C:\Mis Ejemplos\hola.cpp  Linea 2: error: 'prinf' was not
                           declared in this scope
==== Generacion finalizada: 1 errores, 0 warnings ====

```

Figura 3.15: Error en la figura 3.14 debido a la sentencia `printf`

3.4.3. Errores de ejecución

Por desgracia, la acción conjunta del precompilador de C± y el compilador de C++ no es infalible. Hay errores que no pueden detectarse con un análisis estático del código, sino que hay que esperar a que el programa empiece a funcionar para que los problemas afloren. Lamentablemente, la capacidad de `Code::Blocks`^{C±} para localizar automáticamente estos problemas es muy limitada.

3.4.3.1. Errores sobre los que `Code::Blocks`^{C±} no ofrece ninguna ayuda

Supongamos, por ejemplo, que deseamos desarrollar un programa `hola10` que imprima por pantalla la palabra “Hola” diez veces, en diez líneas distintas (ver figura 3.16).

Si escribimos el programa de la figura 3.17, al pulsar **Ejecutar** el código pasará los filtros del precompilador de C± y el compilador de C++. Sin embargo, al ejecutar el programa se producirá el resultado erróneo de la figura 3.18.

Aunque el programa es correcto en el sentido de que satisface todas las reglas de C±, es incorrecto en el sentido de que incumple su objetivo (lo que en *ingeniería del software* se conoce como *las especificaciones del*

```
Hola  
Hola  
Hola  
Hola  
Hola  
Hola  
Hola  
Hola  
Hola  
Hola
```

Figura 3.16: Salida esperada para el programa hola10

```
1 #include <stdio.h>  
2  
3 int main() {  
4     for (int k=1; k<=10; k++) {  
5         printf("Hola");  
6     }  
7 }
```

Figura 3.17: Primera versión de hola10

```
HolaHolaHolaHolaHolaHolaHolaHola  
El proceso devolvio 0 (0x0) tiempo de ejecucion : 0.067 s  
Pulsar cualquier tecla para continuar.
```

Figura 3.18: Resultado producido por el programa de la figura 3.17

cliente). Como el lector puede imaginarse, “Code::Blocks^{C±} no hace magia”: el entorno carece de la capacidad telepática de adivinar cuál es el propósito que el programador tiene en mente cuando se pone a codificar. Así, tendremos que esforzarnos para localizar manualmente el error y subsanarlo (ver figura 3.19).

```
1 #include <stdio.h>  
2  
3 int main() {  
4     for (int k=1; k<=10; k++) {  
5         printf("Hola\n");  
6     }  
7 }
```

Figura 3.19: Versión correcta de hola10

3.4.3.2. Manual de estilo de C±

Aunque el programa de la figura 3.19 es correcto, resulta difícil de leer porque no se han respetado las reglas de encolumnado recomendadas en el manual de estilo de C± (ver apéndice B del libro básico [1]).

Si seleccionamos **Complementos → Formateador de código (AStyle)**, Code::Blocks^{C±} formateará adecuadamente el código, produciendo la figura 3.20.

```

1 #include <stdio.h>
2
3 int main() {
4     for (int k=1; k<=10; k++) {
5         printf("Hola\n");
6     }
7 }
```

Figura 3.20: Formateado automático de la figura 3.19

3.4.3.3. Una última bala en la recámara

Imaginemos que deseamos realizar un programa `holaN`, que pregunte por el número de veces que debe imprimirse la palabra “Hola”. El código del programa, que aparece en la figura 3.21 parece correcto. Sin embargo, su ejecución no imprimirá “Hola” las veces solicitadas...

```

1 #include <stdio.h>
2
3 int main() {
4     int rep;
5
6     printf( "Repeticiones: " );
7     scanf( "%d", &rep );
8     for (int k=1; k<=rep; k++) {
9         printf("Hola\n");
10    }
11 }
```

Figura 3.21: Primera versión de `holaN`

Además de errores, Code::Blocks^{C±} puede producir advertencias (*warnings* en inglés). Mientras que un error siempre apunta a un fallo, una

advertencia simplemente es un indicio de un posible problema. Es decir, ante un programa correcto, Code::Blocks^{C±} puede equivocarse y producir advertencias.

En nuestro ejemplo, Code::Blocks^{C±} producirá el mensaje de advertencia de la figura 3.22, que nos informa de que hemos cometido un error en la línea 7 del programa. Aunque el mensaje es algo críptico, si nos fijamos en dicha línea, veremos que hemos olvidado el símbolo & delante del identificador rep (ver figura 3.23).

```
C:\Mis Ejemplos\hola.cpp  In function 'int main()':
C:\Mis Ejemplos\hola.cpp  Linea 7: warning: format argument is not
                           a pointer (arg 2)

== Generacion finalizada: 0 errores, 1 warnings ==
```

Figura 3.22: Advertencia de error para el programa de la figura 3.21

```
1 #include <stdio.h>
2
3 int main() {
4     int rep;
5
6     printf( "Repeticiones: " );
7     scanf( "%d", &rep );
8     for (int k=1; k<=rep; k++) {
9         printf("Hola\n");
10    }
11 }
```

Figura 3.23: Versión correcta de holaN

Como hemos apuntado anteriormente, las advertencias son un arma de doble filo. Utilizando un símil judicial, en la notificación de errores, Code::Blocks^{C±} se comporta como un juez benévolos que, de vez en cuando, absuelve a delincuentes, pero asegura que todos los criminales que acaban en la cárcel son culpables. Sin embargo, en la notificación de advertencias, Code::Blocks^{C±} se comporta como un juez demasiado severo y, consecuentemente, algunos inocentes pueden ser declarados culpables erróneamente. Resumiendo, las advertencias de Code::Blocks^{C±} tienen dos efectos, uno positivo y otro negativo:

1. Efecto positivo: **Code::Blocks^{C±}** es potencialmente capaz de detectar nuevos errores.
2. Efecto negativo: **Code::Blocks^{C±}** puede equivocarse y producir advertencias que no deben tenerse en cuenta.

Capítulo 4

Primera práctica: impresión de datos personales

EN este capítulo se recoge el enunciado de la primera práctica. Se trata de una práctica sumamente sencilla que tiene como objetivo familiarizar al alumno con el entorno de programación de C±. Todas las prácticas propuestas en este libro pueden corregirse automáticamente con el entorno de programación. El capítulo incluye una descripción detallada del proceso de detección automática de errores para la primera práctica.

4.1. Enunciado de la práctica

La primera práctica consiste en imprimir en pantalla los datos personales del alumno incluidos en la cabecera. Estos datos son: NOMBRE, PRIMER APELLIDO, SEGUNDO APELLIDO, DNI y EMAIL.

El código fuente de todas las prácticas debe incluir una cabecera que contenga los datos personales del alumno. La figura 4.1 muestra el formato de dicha cabecera. Como puede observarse:

1. De acuerdo con la sintaxis de C±, la cabecera es un comentario delimitado por los símbolos /* y */.

2. La cabecera ocupa 7 líneas e incluye los 5 campos obligatorios: NOMBRE, PRIMER APELLIDO, SEGUNDO APELLIDO, DNI y EMAIL. El valor de cada campo se delimita por el símbolo #.

```
1  ****
2  * NOMBRE: #Juan Antonio#
3  * PRIMER APELLIDO: #Fernandez#
4  * SEGUNDO APELLIDO: #Gonzalez#
5  * DNI: #99999999#
6  * EMAIL: #jantonio.gonzalez@mimail.com#
7  ****
```

Figura 4.1: Ejemplo de cabecera válida para las prácticas 1, 2 y 3

La ejecución de esta primera práctica debe imprimir en pantalla los datos personales del alumno y que coincidan con los datos incluidos en la cabecera.

Por ejemplo, si la práctica incluye la cabecera de la figura 4.1, el resultado de su ejecución será la figura 4.2. Como puede observarse, el valor de cada campo se imprime en una línea distinta.

```
1 Juan Antonio
2 Fernandez
3 Gonzalez
4 9999999
5 jantonio.gonzalez@mimail.com
```

Figura 4.2: Resultado de ejecutar la primera práctica (suponiendo que incluye la cabecera de la figura 4.1)

La impresión por pantalla de los datos personales debe coincidir de forma estricta con los datos contenidos en la cabecera del código fuente. No serán resultados correctos si existen diferencias en el uso de mayúsculas/minúsculas, por ejemplo no será correcto usar en la cabecera el nombre 'JOSE ANTONIO' y luego imprimir el valor 'Jose Antonio'. De la misma forma tampoco se dará como correcta la solución que use como DNI el valor en la cabecera '012345' e imprima la salida '12345'.

4.2. Autocorrección de la práctica

Una vez editado el código fuente de la práctica 1, comprobaremos su corrección seleccionando ***Prácticas → Comprobar la práctica 1*** en nuestro entorno de programación para C \pm .

La acción de autocorrección iniciará las operaciones de comprobación de la primera práctica. Una vez iniciada la acción se abrirá una nueva pestaña, junto a la utilizada para la edición del código fuente, donde se visualizará el resultado de la corrección (ver figura 4.3).

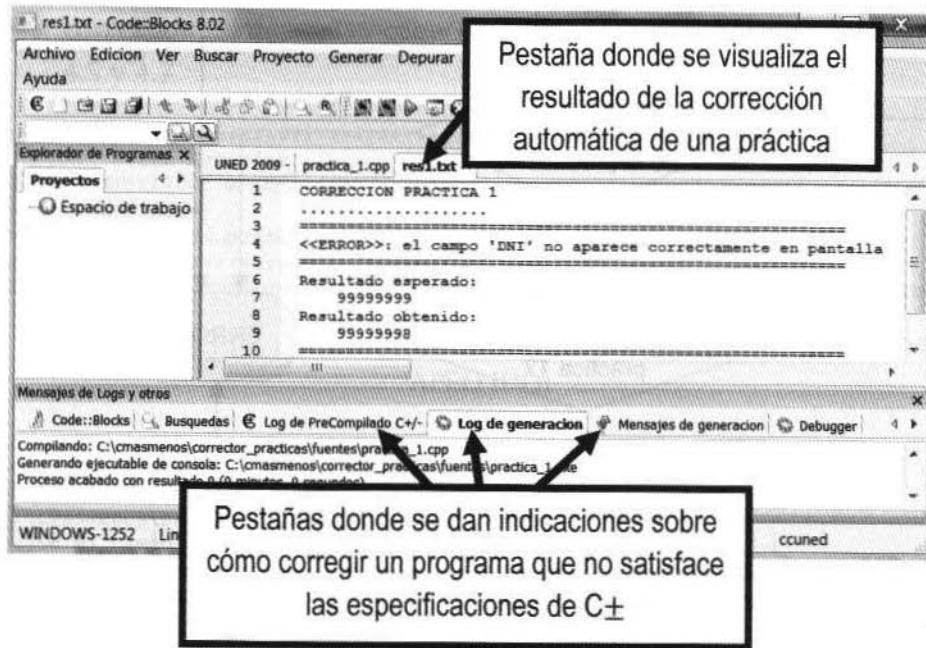


Figura 4.3: Resultado de la corrección automática de las prácticas

La figura 4.4 resume esquemáticamente las comprobaciones que realiza el corrector automático.

El proceso de autocorrección consta de los siguientes pasos:

1. ¿El programa 'Práctica 1' es válido según el lenguaje C \pm ?

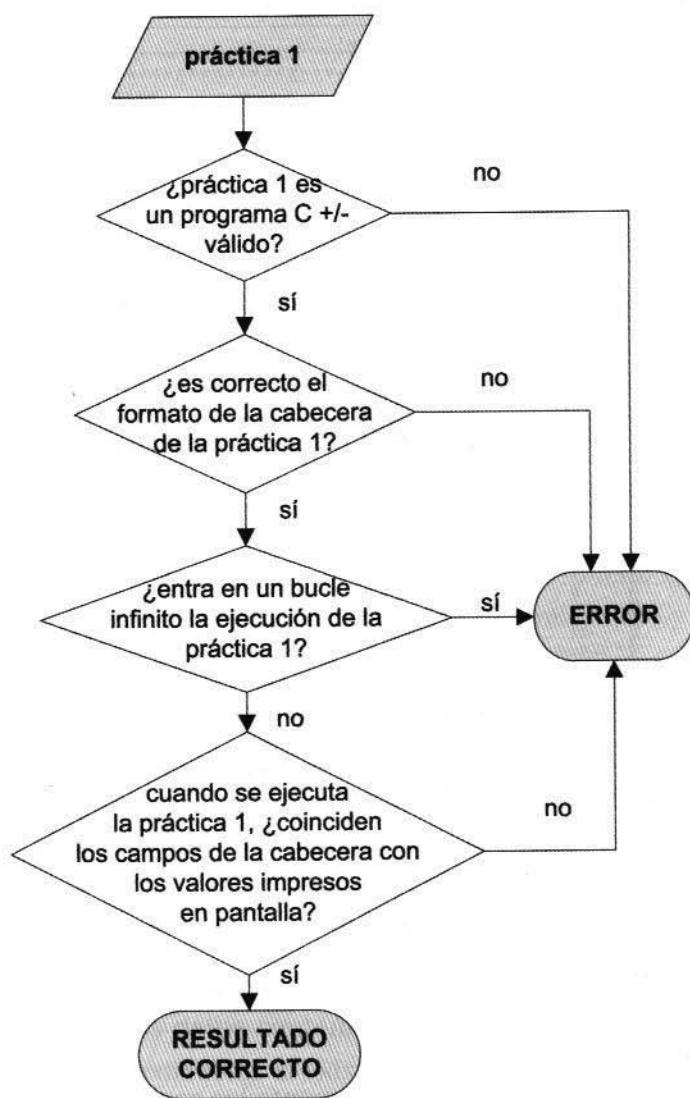


Figura 4.4: Corrección automática de la práctica 1

Si el código fuente de la práctica 1 no satisface las restricciones sintácticas y semánticas de C \pm aparecerá un mensaje de error y no se podrá obtener el programa ejecutable de la práctica. Para solventarlo, conviene consultar los mensajes explicativos que aparecen en la parte inferior de la pantalla (pestañas *Log de PreCompilado C \pm* , *Log de generacion* y *Mensajes de generacion* de la figura 4.3).

2. ¿Es correcto el formato de la cabecera de la práctica 1?

Si la cabecera de la práctica es incorrecta se generará un mensaje de error. Por ejemplo, las figuras 4.6 y 4.8 muestran los mensajes correspondientes a las cabeceras erróneas incluidas en las figuras 4.5 y 4.7.

```
*****  
* NOMBRE: #Juan Antonio#  
* PRIMER APELLIDO: #Fernandez#  
* DNI: #99999999#  
* EMAIL: #jantonio.gonzalez@mimail.com#  
*****/
```

Figura 4.5: Ejemplo de cabecera incorrecta (falta el campo SEGUNDO APELLIDO)

```
=====  
<<ERROR>>: Error en la cabecera del fichero  
'C:\cmasmenos\practicas\practica_1.cpp',  
revise el campo 'SEGUNDO APELLIDO'  
=====
```

Figura 4.6: Mensaje de error provocado por la cabecera de la figura 4.5

```
*****  
* NOMBRE: Juan Antonio  
* PRIMER APELLIDO: #Fernandez#  
* SEGUNDO APELLIDO: #Gonzalez#  
* DNI: #99999999#  
* EMAIL: #jantonio.gonzalez@mimail.com#  
*****/
```

Figura 4.7: Ejemplo de cabecera incorrecta (el valor del campo NOMBRE no está delimitado por el símbolo #)

```
=====
<<ERROR>>: Error en la cabecera del fichero
'C:\cmasmenos\practicas\practica_1.cpp',
revise el campo 'NOMBRE'
=====
```

Figura 4.8: Mensaje de error provocado por la cabecera de la figura 4.7

3. ¿Entra en un bucle infinito la ejecución de la práctica 1?

El corrector automático comprueba que la ejecución de la práctica no entra en un bucle infinito, es decir, no se queda colgada. Por ejemplo, el bucle `while` de la figura 4.9 imprimirá el carácter ‘‘0’’ una cantidad infinita de veces. Ante este error, el corrector producirá el mensaje de la figura 4.10.

```
*****  
* NOMBRE: #Juan Antonio#  
* PRIMER APELLIDO: #Fernandez#  
* SEGUNDO APELLIDO: #Gonzalez#  
* DNI: #99999999#  
* EMAIL: #jantonio.gonzalez@mimail.com#  
*****  
  
#include <stdio.h>  
  
int main()  
{  
    int dni;  
  
    dni = 0;  
    while (dni < 1) {  
        printf("%d\n",dni);  
    }  
}
```

Figura 4.9: Práctica cuya ejecución entra en un bucle infinito

```
=====  
<< ERROR >>  
-----  
Se ha detectado un bucle infinito. Revise su codigo  
=====
```

Figura 4.10: Mensaje de error provocado por la ejecución de un bucle infinito

4. Cuando se ejecuta la práctica 1, ¿coinciden los campos de la cabecera con los valores impresos en pantalla?

El entorno verifica si se satisface el enunciado de la práctica. Por ejemplo, la práctica de la figura 4.11 es incorrecta porque el NOMBRE que imprime en pantalla (Jacinto) no coincide con el NOMBRE que figura en la cabecera (Juan Antonio).

```
*****  
* NOMBRE: #Juan Antonio#  
* PRIMER APELLIDO: #Fernandez#  
* SEGUNDO APELLIDO: #Gonzalez#  
* DNI: #99999999#  
* EMAIL: #jantonio.gonzalez@mimail.com#  
*****/  
  
#include <stdio.h>  
  
int main()  
{  
    printf("Jacinto\n");  
}
```

Figura 4.11: Ejemplo que no satisface el enunciado de la primera práctica

Además, la práctica no imprime el valor del resto de los campos. Ante estos fallos, el corrector producirá el mensaje de la figura 4.12.

```
1 ======  
2 <<ERROR>>: en pantalla deberian aparecer 5 lineas:  
3 ======  
4     (1) nombre  
5     (2) primer apellido  
6     (3) segundo apellido  
7     (4) dni  
8     (5) email  
9 ======
```

Figura 4.12: Mensaje de error que informa de que el ejemplo de la figura 4.11 no satisface el enunciado de la primera práctica

Si la práctica es correcta, se generará automáticamente el fichero `practica_2.cpp`, que incluirá la cabecera de la segunda práctica. Dicho fichero se localizará en la misma carpeta que la primera práctica.

Capítulo 5

Conceptos básicos del entorno

EN este capítulo se incluye la descripción del entorno de programación **Code::Blocks^{C±}**. Se introducen los elementos que forman parte del mismo y cómo pueden utilizarse para realizar las tareas básicas de programación.

La figura 5.1 resume esquemáticamente el contenido del capítulo, que comienza con una presentación de los elementos que forman parte de **Code::Blocks^{C±}**.

A continuación, se comentan las operaciones relacionadas con la gestión de archivos de código C±: crear un nuevo programa, abrir un programa previamente escrito o guardar las modificaciones realizadas. También, y tratándose de una de las funciones básicas de un entorno de programación, se describe todo lo relacionado con la edición del código: tratamiento de texto, acciones rápidas sobre el texto, resaltado de código, etc.

Después, se describen las operaciones relacionadas con la generación de código ejecutable: la comprobación de la corrección del código fuente de un programa y el paso de dicho código fuente a código ejecutable.

Para finalizar, hemos incluido un apartado de comentarios básicos relacionados con la forma de proceder a la hora de resolver algunos de los errores más comunes que pueden cometerse al programar con C \pm .

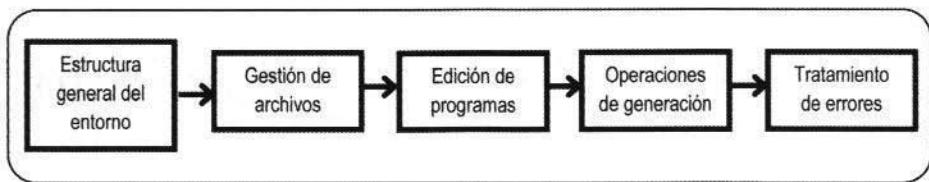


Figura 5.1: Estructura conceptual del capítulo

5.1. Estructura general del entorno

El entorno Code::Blocks $C\pm$ incluye un espacio de trabajo organizado en cinco partes, tal y como se muestra en la figura 5.2:

1. **Área de menús y botoneras:** permite acceder a todas las funciones que se pueden realizar en el entorno. En concreto, Code::Blocks $C\pm$ dispone de:
 - Doce menús: *Archivo, Edición, Ver, Buscar, Proyecto, Generar, Depurar, Herramientas, Prácticas, Complementos, Configuración y Ayuda*.
 - Cinco botoneras: *Principal, Compiler, Debugger, Thread Search y Code Completion*. El objetivo de estos botones es proporcionar una forma rápida para la realización de las operaciones más comunes sin necesidad de pasar por la navegación de los menús.
2. **Área de gestión del espacio de trabajo:** presenta la organización de los recursos (archivos y símbolos) que se manejan en un momento dado en el entorno Code::Blocks $C\pm$.
3. **Área de edición:** aquí realizaremos las operaciones de tratamiento de texto. Dicha área está organizada para que podamos disponer de

diferentes archivos abiertos de forma simultánea con la navegación por pestañas.

4. **Área de mensajes:** nos servirá para obtener la información sobre las operaciones que realizamos: mensajes generales del entorno, operaciones de generación, operaciones de depuración y búsquedas.
5. **Barra de estado:** nos dará información sobre el archivo activo, la línea y columna en la que se posiciona el cursor, la operación de tratamiento de texto activa, etc.

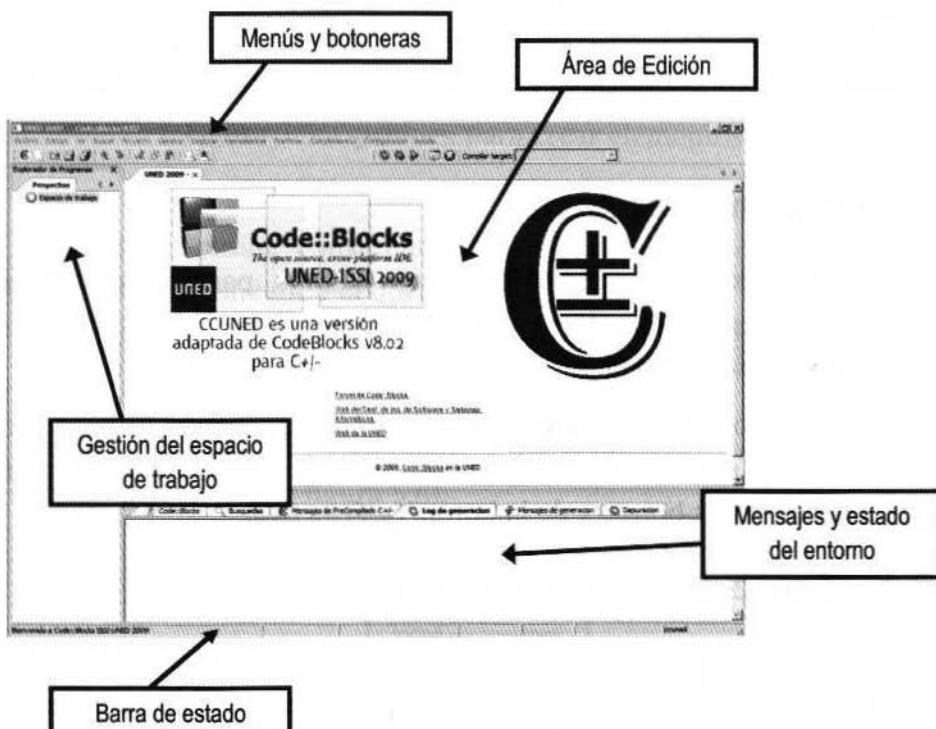


Figura 5.2: Página de bienvenida de Code::Blocks^{C±}

La figura 5.3 resume la secuencia básica de acciones cuando se utiliza Code::Blocks^{C±}.

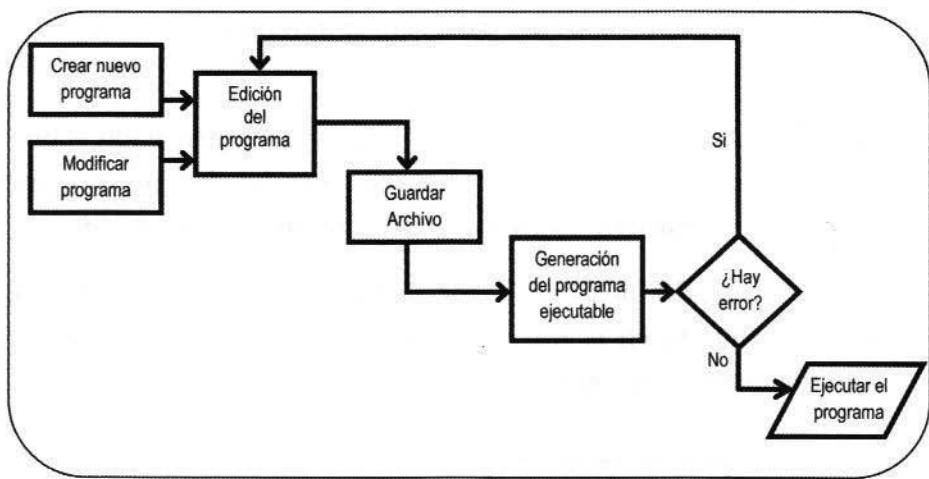


Figura 5.3: Secuencia básica de acciones cuando se utiliza Code::Blocks^{C±}

5.2. Gestión de archivos

Code::Blocks^{C±} incluye las opciones básicas para la gestión de archivos de código fuente: la creación de archivos, el almacenamiento de los programas escritos y la apertura de los archivos almacenados.

Tal y como se ha comentado previamente, podemos utilizar estas funciones en el entorno de dos formas: desde las opciones del menú y desde determinados botones de la botonera *Principal*.

5.2.1. Creación de nuevos archivos

La primera operación que se debe conocer es cómo crear un nuevo archivo. Desde la opción **Archivo→Nuevo** (ver figura 5.4), accedemos a las cuatro opciones disponibles (ver figura 5.5):

1. Fichero vacío
2. Proyecto
3. Archivo
4. Desde plantilla de usuario

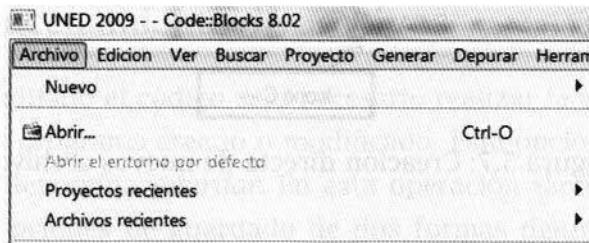
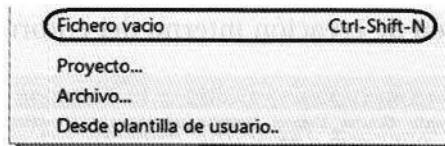
Figura 5.4: *Archivo → Nuevo*

Figura 5.5: Opciones para crear elementos

En este capítulo, consideraremos únicamente la opción de crear nuevos archivos utilizando la opción de ***Fichero vacío***.

La combinación rápida de teclas para realizar esta misma operación corresponde con: ***Ctrl+Shift+N***

Esta misma opción es accesible desde la botonera *Principal* marcando el segundo ícono (ver figura 5.6).

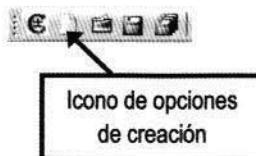


Figura 5.6: Creación de nuevos archivos

También se ha incluido en esta misma botonera *Principal*, el primer ícono correspondiente a C± (ver figura 5.7). Este botón permite la creación directa de un fichero vacío de C± sobre el área de trabajo (la misma operación antes comentada, pero en un único paso).

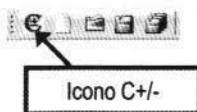


Figura 5.7: Creación directa de nuevos archivos

En cualquiera de los casos mencionados, el resultado de esta acción es el mismo. El entorno creará (no almacenará) un fichero vacío con identificador `*SinNombreN1` en el área de edición, donde N será el ordinal correspondiente al orden de creación interno del entorno (ver figura 5.8).

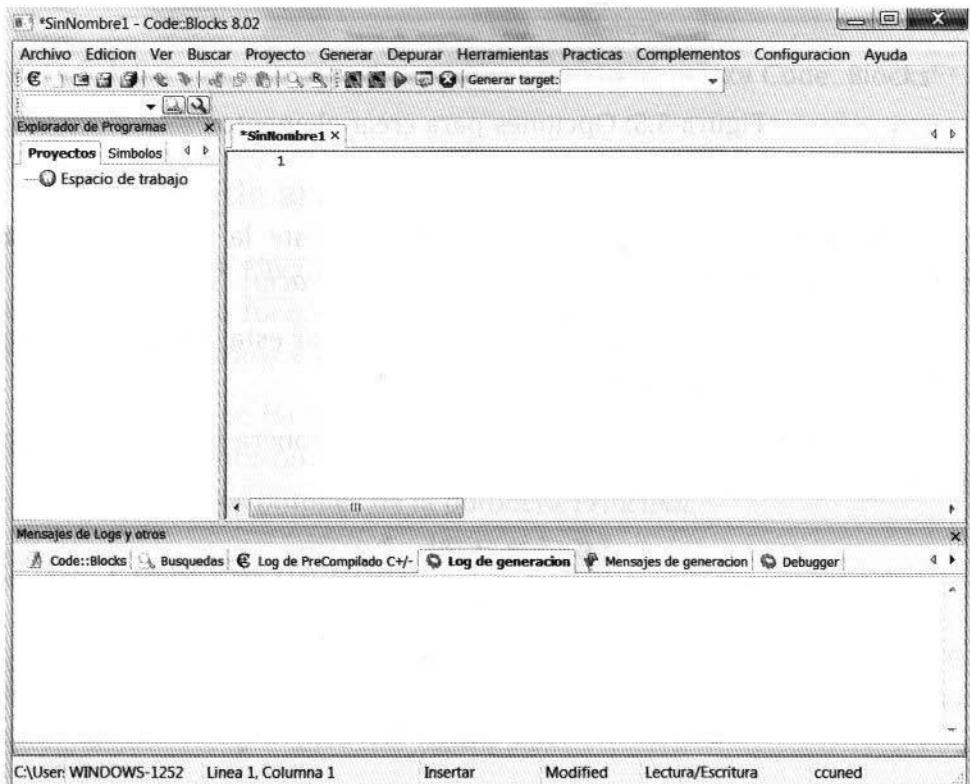


Figura 5.8: Archivo vacío sin nombre

¹Un asterisco a la izquierda del nombre del archivo significa que se han realizado modificaciones en el archivo que no se han guardado en disco.

5.2.2. Almacenamiento de archivos

Una vez editado el código será necesario realizar la acción de almacenamiento del programa creado o modificado. Esta opción de la gestión de ficheros se denomina *guardar*. En esta operación también podremos acceder a las opciones de guardado de dos formas distintas para obtener al mismo resultado: una por el menú **Archivo** y otra por la botonera **Principal**.

En el menú **Archivo** encontramos tres alternativas de guardado (ver figura 5.9):

- **Guardar** : almacenará el archivo seleccionado en el área de edición sin necesidad de un diálogo adicional. En caso de no tener nombre, se mostrará el diálogo de navegación de archivos del sistema para identificar el archivo que se va a almacenar (ver figura 5.10).
- **Guardar como** : iniciará el diálogo basado en la navegación del sistema de archivos para almacenar el archivo seleccionado.
- **Guardar todos los archivos** : almacenará todos los archivos abiertos en el área de edición que no estén almacenados. En caso de existir archivos que no tengan nombre, se mostrará por cada uno de ellos un diálogo de navegación que nos permitirá elegir el nombre y el directorio donde almacenarlos.

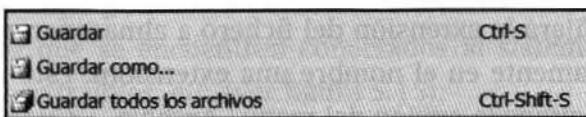


Figura 5.9: Opciones para almacenar en el menú **Archivo**

En la botonera **Principal** encontramos dos iconos que se corresponden con las mismas acciones que la primera y la tercera antes comentadas: *guardar* y *guardar todos* (ver figura 5.11).

Además del directorio en el que almacenaremos los ficheros, en el diálogo para guardar podemos seleccionar el nombre y el tipo del fichero

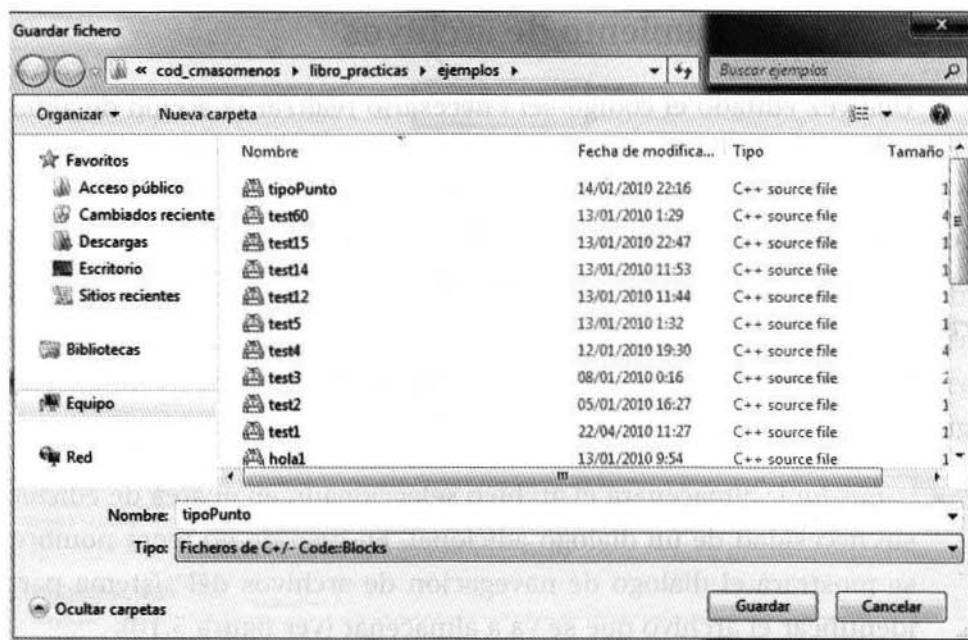


Figura 5.10: Diálogo para guardar un fichero

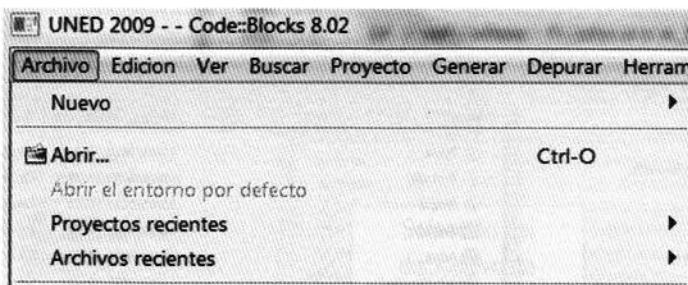


Figura 5.11: Iconos para guardar un fichero

(esta selección fijará la extensión del fichero a almacenar, a no ser que se indique explícitamente en el nombre una extensión). Este mismo diálogo se iniciará si elegimos la opción del menú **Archivo → Guardar como**.

5.2.3. Opciones para abrir archivos almacenados

Este último apartado sobre gestión de archivos está dedicado a las opciones disponibles para abrir un archivo ya existente. Como en los casos anteriores, el entorno proporciona dos alternativas para realizar esta tarea: por menú y por la botonera *Principal*. Desde el mismo menú de **Archivo** accedemos a la opción **Abrir ...** (ver figura 5.12).

Figura 5.12: *Archivo → Abrir ...*

En este caso, el botón de la botonera *Principal* que se ha incorporado para esta acción es el tercero (ver figura 5.13).

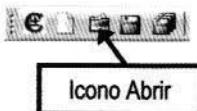


Figura 5.13: Iconos para abrir ficheros

Estas dos acciones desencadenan la apertura de un diálogo de selección del archivo que se quiere abrir de acuerdo a las características de navegación del sistema de ficheros que se tenga configurado en cada sistema operativo del tipo Windows (ver figura 5.14).

En este diálogo podremos acceder a los diferentes dispositivos de almacenamiento que se encuentren conectados al sistema: discos duros internos, discos extraíbles, etc. (ver figura 5.15).

Por defecto, este diálogo se iniciará con un filtro aplicado a los ficheros que se muestran. El filtro que debe considerar el usuario en estas opciones iniciales es el de *Ficheros de C± Code::Blocks* que correspondería con una máscara *.cpp.

Si se desea utilizar otro tipo de archivos, podemos modificar el filtro que se aplica. Los filtros existentes en el entorno son para ficheros de tipo: C++, C± Code::Blocks, de cabecera Code::Blocks, de proyecto Code::Blocks, de entorno de trabajo Code::Blocks, de recursos Windows y XML. Además,

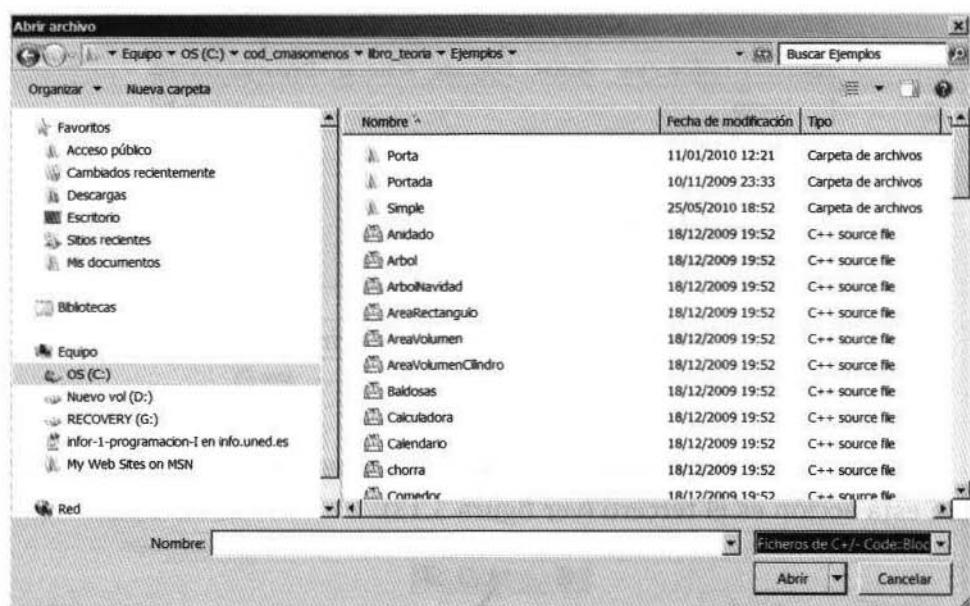


Figura 5.14: Diálogo de navegación del sistema de ficheros

pueden visualizarse todos los ficheros de una carpeta si no se utiliza ningún filtro.

Otro mecanismo de gran utilidad son los accesos rápidos a los archivos con los que se ha trabajado últimamente y ya se han cerrado. Según se trabaja, el entorno acumula una lista de estos archivos y los pone a disposición del usuario a través de la opción **Archivo → Archivos recientes** (ver figura 5.16).

5.3. Opciones de edición

5.3.1. Introducción

Todas las opciones de edición se encuentran en el menú **Edición** y, aquellas que son más importantes, están incorporadas en la botonera **Principal**.

Todas las acciones que realizamos al editar se aplicarán en el área de edición del entorno (ver figura 5.17).

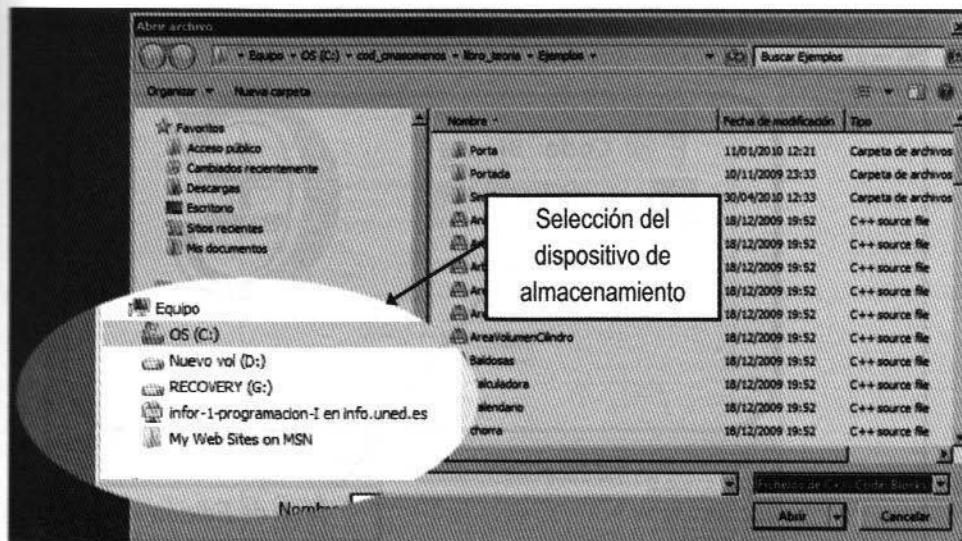


Figura 5.15: Selección y navegación por los dispositivos de almacenamiento

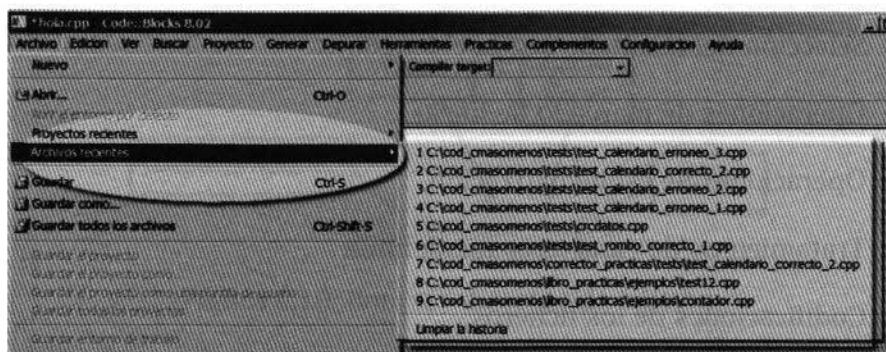


Figura 5.16: Lista de archivos recientes

Las operaciones que vamos a poder realizar en el entorno son (ver figura 5.18):

- Ejecución de operaciones: deshacer/rehacer.
- Operaciones de selección de texto: copiar/cortar/pegar.
- Navegación Cabecera/Fuente en ficheros de lenguajes tipo C++.

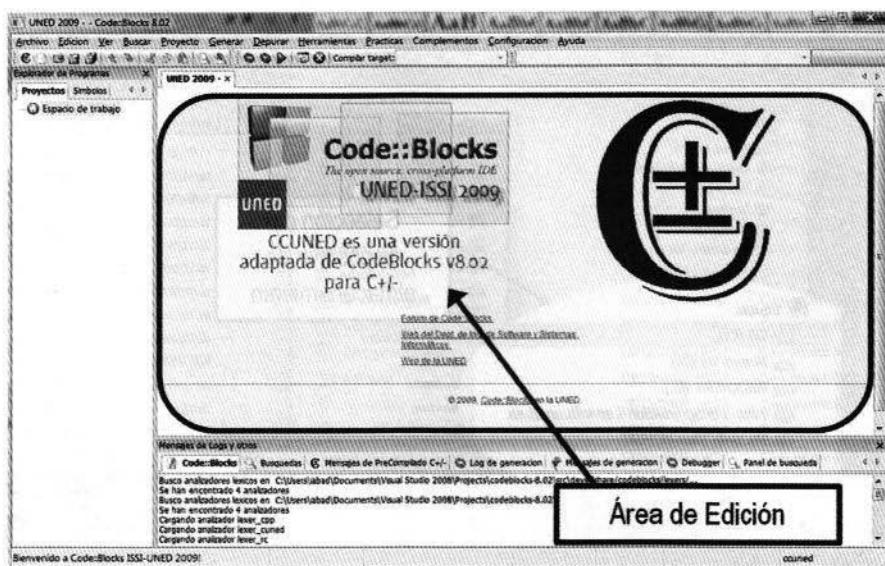


Figura 5.17: Área de edición

- Selección del resultado según el lenguaje.
- Operaciones de marcado de líneas.
- Operaciones de esquematización.
- Tratamientos de caracteres fin de línea.
- Codificaciones de los ficheros.
- Acciones particulares: operaciones con párrafos o líneas completas, zoom/no zoom de texto, tipos de letra.
- Gestión de comentarios.
- Ayudas a la escritura de código: autocompletar y balanceo de delimitadores.

Entre todas estas opciones, las más utilizadas de la botonera *Principal* son las acciones de ejecución de operaciones y las de selección de texto (ver figura 5.19).

Deshacer	Ctrl-Z
Rehacer	Ctrl-Shift-Z
Cortar	Ctrl-X
Copiar	Ctrl-C
Pegar	Ctrl-V
Cambiar entre fichero fuente y fichero de cabecera F11	
Modo de resultado de lenguaje	▶
Marcadores	▶
Esquematización	▶
Modo del fin del linea	▶
Codificación del fichero	▶
Comandos especiales	▶
Seleccionar todo	Ctrl-A
Comentar	Shift-Ctrl-C
Descomentar	Shift-Ctrl-X
Altera el estado de comentado o no	
Comenta con abrir-cerrar	
Comentar con caja	
Auto-completar	Ctrl-J
Ir a la llave, parentesis o corchete correspondiente	Ctrl-Shift-B

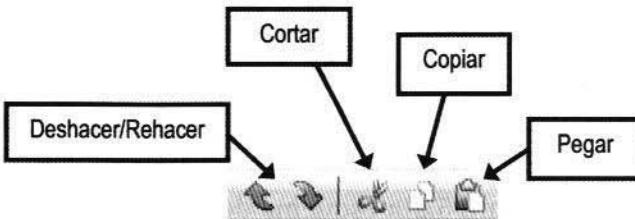
Figura 5.18: Menú de *Edición*

Figura 5.19: Botones directos de edición

En este apartado comentaremos únicamente las operaciones de edición que se usan con mayor frecuencia en el trabajo habitual del programador.

5.3.2. Acciones de edición de texto

Las operaciones de edición de texto corresponden a las acciones nativas² que están disponibles en los sistemas Windows. Estas acciones están disponibles sólo si se desencadena una acción de selección de texto en el área de edición.

El esquema de funcionamiento básico para la edición consiste en una operación de selección de los elementos con los que se quiere operar (ver figura 5.20). A continuación se debe elegir cuál es la operación a realizar: copiar o cortar. Si se quiere volver a situar los elementos en el área de edición, entonces posicionaremos el cursor en el destino y se realizará la operación de pegar. Si se ha elegido la operación de cortar y no se realiza otra acción se habrá realizado una operación de borrado.

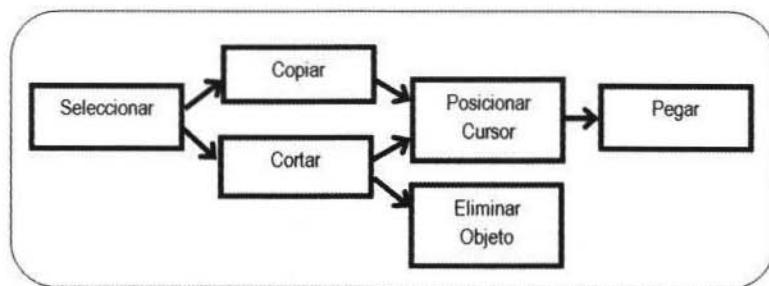


Figura 5.20: Esquema de edición

La edición de texto se realiza de la misma forma que en el resto del entorno Windows (por lo tanto, su funcionamiento está sujeto a la configuración que se haya elegido en cada entorno operativo concreto) con las siguientes operaciones:

- **Selección de texto:** si se utiliza como dispositivo el ratón, pulsaremos el botón izquierdo para marcar el inicio a seleccionar y arrastraremos de arriba hacia abajo y de izquierda a derecha, para soltar el botón al final de la selección. También se puede realizar esta ope-

²Eventos que se resuelven por parte del sistema operativo directamente sin la intervención de las aplicaciones.

ración con el teclado realizando la siguiente secuencia de acciones: primero llevaremos el cursor hasta el punto de inicio que queremos seleccionar, y en este punto pulsaremos la tecla **Δ** (Mayúscula) y sin dejar de pulsarla moveremos el cursor por el texto hasta el final de la selección (ver figura 5.21).

```
1 // ****
2 /* Procedimiento: CurvaC
3 ****
4 #include <stdio.h>
5 /*
6 */
7 Parámetros globales
8 /**
9 const int ANCHO = 32;
10 const int ALTO = 19; /* la pantalla */
11
12
13
```

A screenshot of a code editor window showing a portion of a C++ program. Lines 9 and 10 are highlighted with a thick, semi-transparent oval selection. Line 9 contains the declaration `const int ANCHO = 32;` and line 10 contains `const int ALTO = 19; /* la pantalla */`. The rest of the code is visible but not selected.

Figura 5.21: Selección de texto

- **Copiar/cortar:** una vez seleccionado el texto podemos realizar una de las dos operaciones disponibles. La acción de *copiar* guardará en un contenedor, denominado *portapapeles*, el texto seleccionado y no realizará ninguna otra acción ni modificación sobre el área editada. La acción de *cortar* guardará en el mismo portapapeles el texto seleccionado, pero eliminará el texto seleccionado y dejará el cursor en el inicio de la selección.
- **Pegar:** una vez se ha seleccionado la acción que se quiere hacer, copiar o cortar, el texto almacenado en el portapapeles puede utilizarse en otro punto del texto que estamos editando. Esta acción es lo que se conoce como *Pegar*. Si un texto se ha cortado y no se pega, es equivalente a una acción de borrado.

Existen varias vías alternativas para Copiar/Cortar/Pegar:

1. Desde el menú de **Edición** de Code::Blocks^{C+}, podemos encontrar las acciones de **Cortar**, **Copiar** y **Pegar**.

2. Desde la botonera *Principal* pulsando con botón izquierdo (esto es con carácter general porque se debe recordar los comentarios sobre particularizaciones de la configuración de sistemas Windows) en los botones antes comentados.

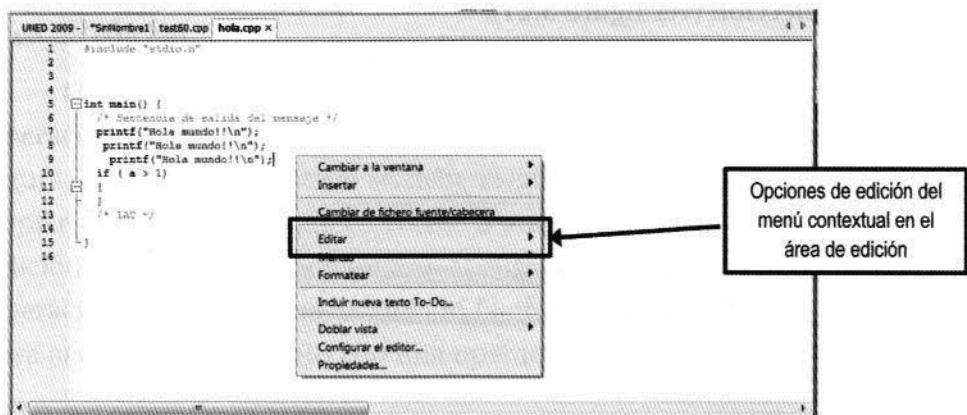


Figura 5.22: Menú *Edit* de contexto en la edición

3. Con las combinaciones de teclas que son nativas de los sistemas operativos Windows para este tipo de acciones:

- Copiar: ***Ctrl+C***
- Cortar: ***Ctrl+X***
- Pegar: ***Ctrl+V***
- Seleccionar todo: ***Ctrl+A***³

4. Con el menú contextual de Edición que obtenemos al colocarnos sobre el área de Edición y pulsar con el botón derecho del ratón (ver figura 5.22)

³En los sistemas con Windows 7, la acción nativa de Windows para seleccionar todo es ***Ctrl+E***. Sin embargo, en el entorno Code::Blocks C++ se mantiene la combinación ***Ctrl+A*** para la selección completa del texto de un archivo.

5.3.3. Apariencia del texto

La configuración del entorno por defecto se ha fijado para utilizar un resaltado de texto según los lenguajes tipo C++ (ver cuadro 5.1).

Color texto	Representa
Negro	Código del programa. Es el valor por defecto que se usará en caso de ser un elemento no definido
Gris	Comentarios en el código
Rosa	Literales numéricos
Azul	Palabras reservadas del lenguaje y para que se destaque más se aplica tipo de letra negrita
Azul	Literales cadena
Naranja	Literales de tipo carácter
Rojo	Operadores y delimitadores

Cuadro 5.1: Apariencia del texto en *Code::Blocks*^{C±}

5.3.4. Autocompletado de texto

El objetivo básico de las opciones de autocompletado de texto es ayudar al programador a la hora de escribir el texto poniendo a su disposición las alternativas de código que pueden resultar más interesantes en el momento de la escritura. En la configuración inicial de *Code::Blocks*^{C±} se ha fijado que la presentación de alternativas se realice a partir del cuarto carácter escrito, por lo que no existirán propuestas por debajo de este número de caracteres. Un ejemplo de funcionamiento lo tenemos al escribir el programa más sencillo que comentábamos en el capítulo 2. Al escribir el identificador `main` obteníamos como alternativa correcta disponible la función global `main` (ver figura 5.23).

Si en lugar de tener una única alternativa, tuviéramos varias, el entorno nos propondría las opciones disponibles. Por ejemplo, si antes del `main`, comenzamos a escribir `type`, obtenemos tres alternativas (ver figura 5.24).

```

3
4 #include <stdio.h>
5
6 int main
7 / (→) main
8 /
9 Parámetros globales
10
11

```

Figura 5.23: Edición con autocompletar

5.3.5. Balanceo de delimitadores de texto

El balanceo de delimitadores de texto nos permite visualizar si la apertura y cierre de bloques se ha realizado correctamente. Los delimitadores para los que se puede aplicar el balanceo son las llaves delimitadoras de bloques, los paréntesis utilizados en expresiones o declaraciones y los corchetes usados en las formaciones. El balanceo representa el estado de los delimitadores de forma visual para indicar si son o no correctos. Con la configuración inicial de Code::Blocks^{C++} los delimitadores correctamente balanceados se muestran en color negro sobre fondo azul y los erróneos en color negro sobre fondo rojo (ver figura 5.25).

```

2
3 #include "stdio.h"
4
5 type|
6 ↗+ typedef
7 ↗+ typeid
8 ↗+ typename
9
10

```

Figura 5.24: Ejemplo de alternativas múltiples con autocompletar

Estas opciones de balanceo se activan o desactivan al colocarnos con el cursor sobre el carácter delimitador que queremos comprobar.

Además, el entorno incluye una opción de navegar entre delimitadores. Esta opción se encuentra al final del menú de **Edición** y puede

```

14 const int NumLin = 20; /* Número de líneas de la página*/
15 const int NumCol = 40; /* Número de caracteres por linea */
16
17 typedef char Pagi[NumLin][NumCol];
18
19 Pagi pp;
20
21 void PaginaEnBlanco() {
22     /* Rellena la página con blancos */
23
24     for (int lin = 0; lin <= NumLin-1; lin++) {
25         for (int col = 0; col <= NumCol-1; col++) {
26             pp[lin][col] = ' ';
27         }
28     }
29 }
30
31

```

Figura 5.25: Delimitador de llave correcto

iniciarse con la combinación de teclas ***Ctrl+Shift+B***. Esta opción es de mucha utilidad en los casos en los que se está comprobando que se han escrito los bloques de forma correcta y estos ocupan más líneas que las que están visibles en el área de edición. Esta acción iniciada con el cursor en uno de los delimitadores nos llevará el cursor hasta el carácter final del bloque. Sólo funcionará en el caso de que el delimitador elegido esté correctamente balanceado (esto es, aparezca en azul).

5.3.6. Otros comentarios básicos

El área de edición permite manejar varios ficheros abiertos de forma simultánea, tal y como se muestra en la figura 5.26. Estos ficheros abiertos se presentan en forma de pestañas. En el caso de que haya una cantidad de ficheros abiertos mayor de la que se puede mostrar en pantalla, se habilitarán las flechas de navegación entre pestañas para poder acceder a todos los ficheros abiertos.

Además, Code::Blocks^{C++} proporciona otros dos mecanismos de información a la edición. Por una parte se muestra en todo momento el número de línea en el que estamos trabajando. Por otra, en la parte inferior se dispone de una barra de información de estado del entorno. En esta barra tenemos disponible el nombre del fichero abierto, la codificación del fichero que

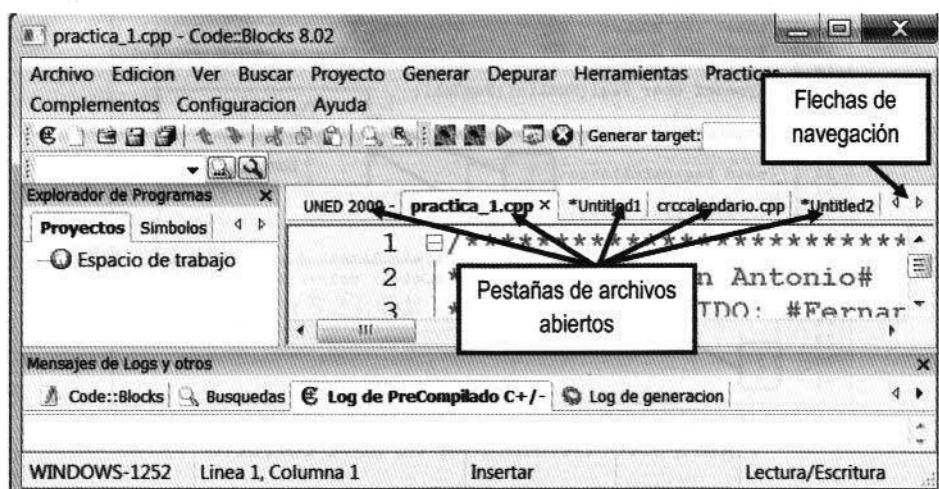


Figura 5.26: Edición con varios ficheros abiertos

manejamos, la posición del cursor: línea y columna, la operación activa, el estado del fichero y el modo del fichero (ver figura 5.27).

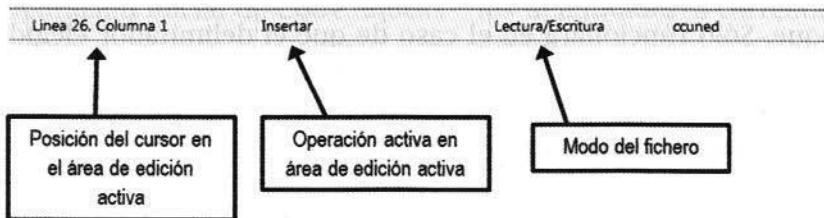


Figura 5.27: Fragmento de la barra de estado de Code::Blocks^{C±}

5.4. Generación de archivos ejecutables

Las operaciones de generación son aquellas que nos permiten realizar las acciones de compilación y montaje de ejecutables a partir de los códigos de los ficheros fuente. La figura 5.28 muestra el esquema básico para generar ficheros ejecutables a partir de código fuente.

A continuación, se describe cada una de las acciones representadas en la figura 5.28:

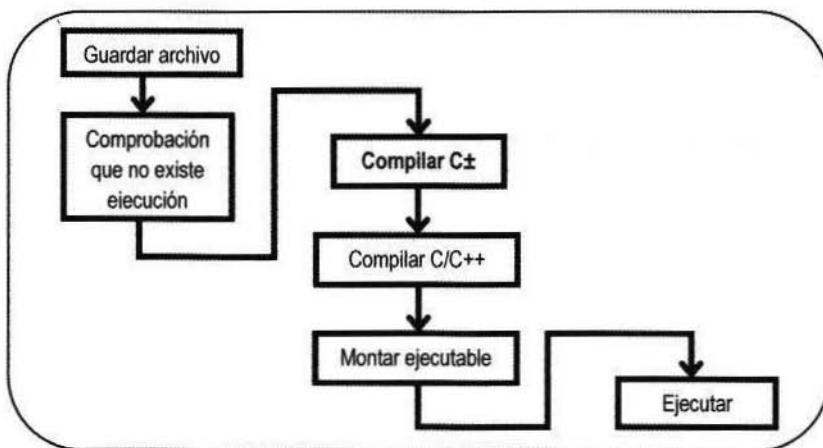


Figura 5.28: Esquema de acciones de generación

1. Comprobación de que el fichero que se quiere compilar ha sido guardado. En caso de no haberse guardado con anterioridad, el entorno guardará automáticamente el fichero con el mismo nombre que tenga. Si hemos creado un fichero nuevo y no los hemos almacenado, el entorno no permitirá realizar las operaciones de generación hasta que no se elija el nombre del fichero. Ante esta situación, las opciones de generación aparecerán como no habilitadas, tal y como se muestra en la figura 5.29.
2. Después de guardar el fichero, se comprueba que el programa que se desea ejecutar no está ya en ejecución. En caso de que ya se encuentre en ejecución, se detiene la operación y se envía el correspondiente mensaje de error.
3. Si se han pasado estas dos comprobaciones iniciales, el proceso inicia las operaciones de compilación. Lo primero que se comprueba es que el fichero sea correcto según la definición de sentencias de C±. Este proceso generará los correspondientes mensajes que se produzcan utilizando como ventana de salida la pestaña *Mensajes de precompilado C±*, figura 5.30.

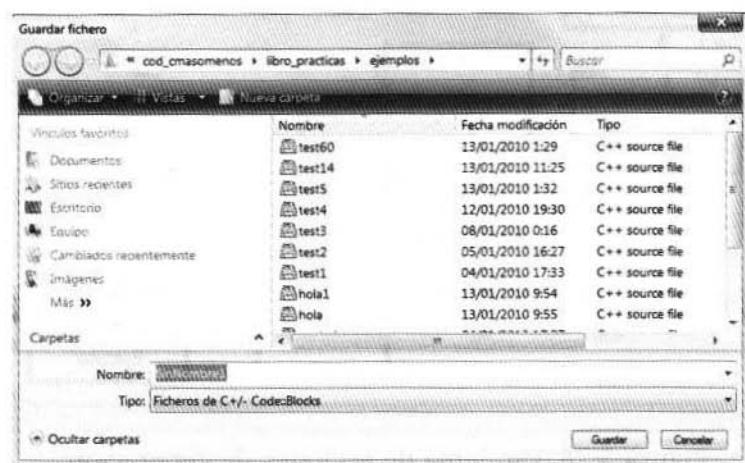


Figura 5.29: Guardar antes de generar

```

1 #include <stdio.h>
2
3 int main()
4 {
5     printf("HOLA\n");
6 }
7

```

Figura 5.30: Mensajes de precompilado C++

4. C± es un subconjunto de C++. Si las comprobaciones del lenguaje C± son correctas entonces se ejecuta la llamada al compilador de C++. El compilador de C++ procesará el código fuente. La salida de dicho proceso se mostrará en las pestañas de mensajes: *Log de generación* y *Mensajes de generación*.

La pestaña *Log de generación* muestra los mensajes tal y como los genera la operación de compilación.

La pestaña *Mensajes de generación* incluye además la referencia al fichero que se compila y la línea a la que se asocia cada mensaje, tal y como se muestra en la figura 5.31.

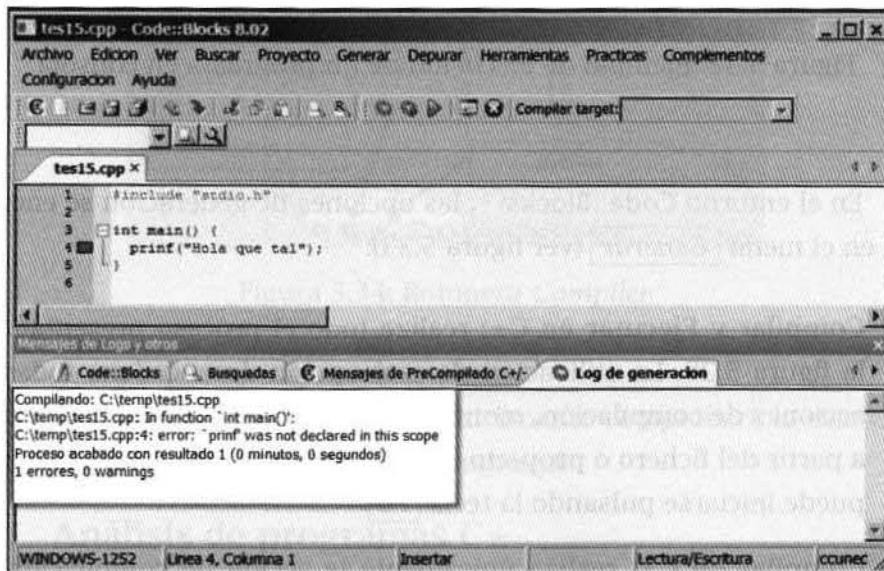


Figura 5.31: Mensajes en *Log de generación*

5. A continuación, se procede a montar o ensamblar los ficheros compilados, produciendo como resultado un fichero ejecutable. En este caso, los mensajes se muestran en las mismas pestañas que en el paso anterior.

6. Finalmente, se ejecuta el programa creado. La configuración inicial de Code::Blocks^{C±} utilizará una consola auxiliar para la ejecución de los programas, tal y como se muestra en la figura 5.32.

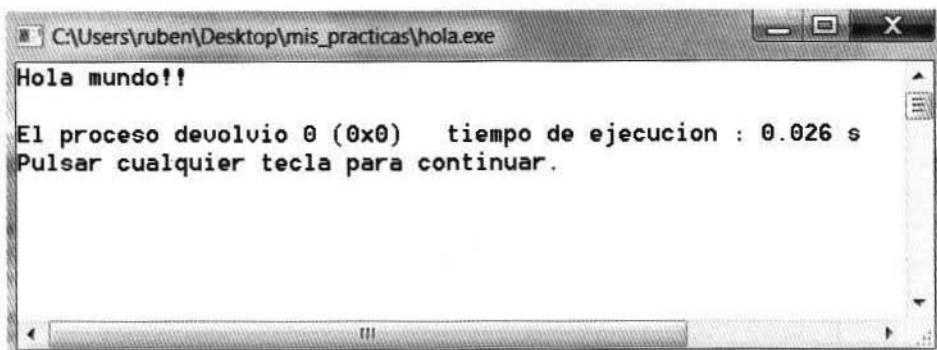


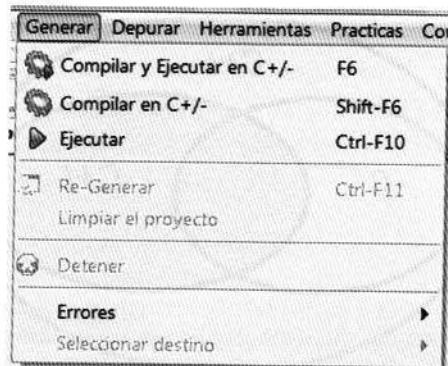
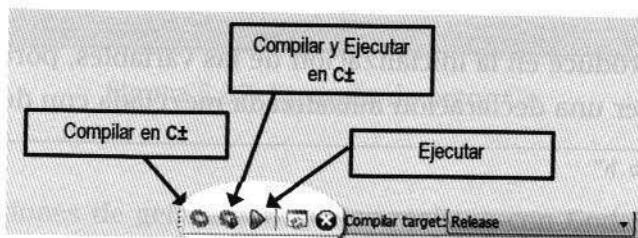
Figura 5.32: Ejemplo de ejecución de un programa en consola

En el entorno Code::Blocks^{C±}, las opciones de generación se encuentran en el menú **Generar** (ver figura 5.33):

1. **Compilar y Ejecutar en C±**: realiza todo el proceso presentado en la figura 5.28. La utilización de esta opción desencadena todas las acciones de compilación, montaje y, si todo es correcto, la ejecución a partir del fichero o proyecto abierto. Alternativamente, esta opción puede iniciarse pulsando la tecla **F6**.
2. **Compilar en C±**: realiza únicamente la acción de compilación del fichero activo o del proyecto activo.
3. **Ejecutar**: inicia la ejecución del fichero (solo si previamente ha sido creado con alguna de las opciones anteriores) o del proyecto activo.

Estas opciones también están disponibles en la botonera *Compiler* (ver figura 5.34).

La opción del menú **Compilar en C±** realizará únicamente las operaciones de generación (compilar y montar el ejecutable) pero no iniciará el programa.

Figura 5.33: Menú *Generar*Figura 5.34: Botonera *Compiler*

La opción del menú *Ejecutar* únicamente comprobará la existencia del ejecutable asociado y lanzará el programa.

5.5. Análisis de programas C \pm

C \pm es un subconjunto de C++ definido en el libro básico de la asignatura [1]. La figura 5.35 muestra la relación de C \pm con los lenguajes C y C++.

Puesto que C \pm es más restrictivo que C++, un programa correcto en C++ puede producir errores en Code::Blocks^{C \pm} . Por ejemplo, el programa de la figura 5.36 es correcto en C++. Sin embargo, presenta dos errores en C \pm . El primero se debe a la función global `main()`, ya que en C \pm es necesario que se indique que la función devuelve un valor entero `int`. El

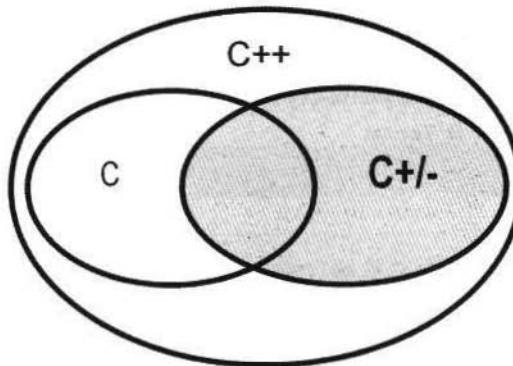


Figura 5.35: Relación entre los lenguajes C, C++ y C \pm

segundo se produce en la inicialización de las variables, porque en C \pm no podemos hacer una declaración inicializada mezclada con declaraciones.

```

1 #include "stdio.h"
2
3 main() {
4     int a, b=1, contador=1, total;
5
6     do {
7         a=1;
8         do {
9             total = b*a;
10            printf("%d x %d = %d \n", b, a, total);
11            a++;
12        } while (a <= 10)
13        b++;
14    } while (b<=10)
15 }
```

Figura 5.36: Ejemplo de programa correcto en C++, pero erróneo en C \pm

En la pestaña de *Mensajes de PreCompilado C±* del área de *Mensajes de Logs y otros* se muestran los dos errores, por ejemplo el segundo se muestra en la figura 5.37

El entorno Code::Blocks^{C \pm} incluye, como paso inicial de generación, la comprobación del programa según las normas de C \pm . Para hacer este análisis se ha introducido una fase de preprocesso, anterior a la compilación propiamente dicha. La fase de preprocesso es OBLIGATORIA en cualquiera

```

1 #include "stdio.h"
2
3 int main() {
4     int a, b=1, contador=1, total;
5
6     do {
7         a=1;
8         do {
9             total = b*a;
10            printf("%d x %d = %d \n", b, a, total);
11            a++;
12        } while (a<=10)
13        b++;
14    } while (b<=10);
15 }

```

Figura 5.37: Ejemplo de error C \pm

de las operaciones de generación que se realicen con **Code::Blocks**^{C \pm} . Se debe tener claro que no se realizará ninguna compilación que previamente no haya sido comprobada según las normas de C \pm .

El preprocesso de un programa consta de dos fases:

1. **Comprobación de los elementos léxicos:** se verifica que todas las “palabras” del programa aparecen en el “diccionario” de C \pm . Los errores léxicos más habituales están relacionados con la utilización de símbolos incorrectos. Por ejemplo, si declaramos una variable con una letra “ñ”. Otro error habitual de este tipo se produce si no cerramos una cadena de caracteres al olvidar las comillas de finalización de un **string**, tal y como se muestra en la llamada al procedimiento **printf** del ejemplo del programa 5.38.

Cuando el preprocessador de C \pm se encuentra con la expresión “Hola que tal”, no es capaz de catalogarla como cadena y, por tanto, no puede hacerla corresponder con ningún tipo de “palabra” válida contenida en el “diccionario” de C \pm , y se genera el error que se muestra en la figura 5.39. Al tratarse de una cadena a la que le falta

```

1 #include "stdio.h"
2
3 int main() {
4     printf("Hola que tal");
5 }
```

Figura 5.38: Error Léxico C± por tener el literal cadena sin cerrar

cerrar las comillas el mensaje nos indica que se ha leido un elemento incorrecto.

2. **Comprobación de la sintaxis:** se asegura que el programa respeta la gramática de C±, es decir, que las “palabras” se combinan correctamente. Por ejemplo, el código del ejemplo 5.40 muestra un programa sintácticamente incorrecto: aunque `while`, `true`, `printf...` son palabras correctas, pero no están bien estructuradas porque `printf(HOLA);` debería delimitarse con llaves al estar dentro de un `while`.

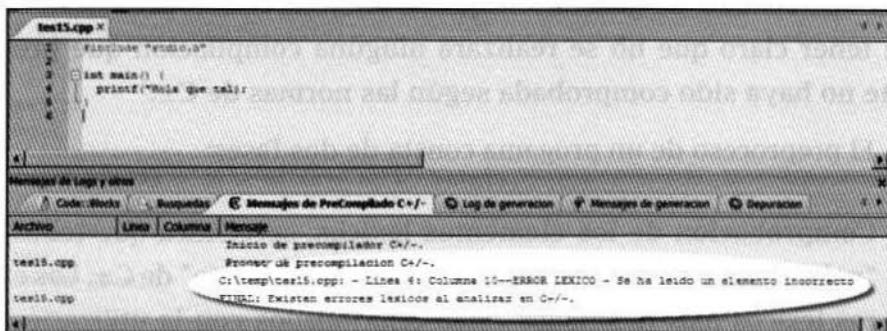


Figura 5.39: Ejemplo de mensaje de error léxico C±

```

1 #include "stdio.h"
2
3 int main() {
4     while (true) printf("Hola que tal \n");
5 }
```

Figura 5.40: Ejemplo de error C±

En este caso el mensaje que se muestra indica que el error se debe a que falta el delimitador del inicio del bloque de sentencias del `while`, tal y como se muestra en la figura 5.41.

```

1 #include <stdio.h>
2
3 /* HOLA 2000
4 PRACTICA */
5
6 int main() {
7     while (true) printf("HOLA\n");
8 }
9

```

Mensajes de Log y otros
Code::Blocks > Búsquedas > Mensajes de PreCompilado C++ > Log de generación > Mensajes de generación > Depuración

Archivo	Línea	Columna	Mensaje
			Inicio de precompilador C++.
Hola.cpp			Proceso de precompilación C++.
Hola.cpp	7	17	Error: : Uso del WHILE incorrecto - Falta llave en el inicio del bloque de sentencias
Hola.cpp			Error sintáctico al analizar en C++.

Figura 5.41: Mensaje de error C±

El analizador sintáctico si no puede determinar con seguridad el error sintáctico que se produce generará un error denominado *Elemento no esperado*. Este mensaje se asocia a aquellos casos en los que se produce un error de la estructura que resulta totalmente inesperado. Por ejemplo si escribimos una sentencia dentro de una asignación, como por ejemplo en el código 5.42

```

1 #include "stdio.h"
2
3 int main() {
4     x = while()
5 }

```

Figura 5.42: Error C± de *Elemento no esperado*

Y los mensajes que se obtienen quedarían tal y como se muestra en la figura 5.43.

The screenshot shows the Code::Blocks IDE interface. In the top panel, there is a code editor window titled "UNED 2009 - *Hola.cpp". The code contains a simple C++ program with a syntax error. Below the code editor is a "Messages" tab. The "Messages" tab has several tabs: "Code::Blocks", "Búsquedas", "Messages de PreCompilado C/C++", "Log de generación", "Messages de generación", and "Depuración". The "Messages de generación" tab is active. It displays the following log entries:

- Inicio de precompilador C/C++.
- Acceso de precompilación C/C++.
- Hola.cpp 11 14 Error: - Elemento no esperado. Se esperaba: - ! ~ { * + bool char CharLitera
- Hola.cpp 11 14 Error: - Elemento no esperado. Se esperaba: - ! ~ { * + bool char CharLitera

A red arrow points from the text "Elemento no esperado" in the error message to the word "while" in the code editor, indicating where the error occurred.

Figura 5.43: Mensaje de error de *Elemento no esperado*

5.6. Tratamiento de errores en C \pm

En este apartado queremos incluir algunas recomendaciones a la hora de resolver los errores más comunes que se producen en el uso de Code::Blocks^{C \pm} . Estos comentarios se han organizado a partir de los conceptos básicos sobre el uso de la generación y de las particularidades comentadas en relación con la generación en C \pm . Aunque haremos especial énfasis en los errores relacionados con C \pm , también describiremos muchos de los errores de programación más frecuentes en lenguajes como C y C++.

5.6.1. Errores relacionados con la función main

El cuadro 5.2 resume los errores más comunes relacionados con la función `main`. Éstos suelen producirse al usar una declaración incompleta. Puede olvidarse, por ejemplo, escribir la palabra reservada `int` para indicar que devuelve un valor de este tipo. O también, si el lector está familiarizado con los lenguajes tipo C, podría intentar hacer uso de argumentos en la función `main()` habituales en esta clase de lenguajes: `argc` y `argv`, como en la figura 5.44.

Es necesario recordar que, en C±, `main()` debe emplearse obligatoriamente como función que devuelve un valor entero y que no tiene parámetros.

Error	Descripción
falta <code>int</code>	La cabecera de programa en C± es: <code>int main()</code>
la estructura principal no tiene argumentos	En C± el programa principal no tiene argumentos. No pueden usarse <code>argc</code> ni <code>argv</code>

Cuadro 5.2: Errores relacionados con la función `main`

```

1  #include <stdio.h>
2
3
4  int main(int argc, char *argv[])
5      printf("Hola MUNDO \n");
6  }
7

```

Mensajes de Log y otros

Code::Blocks Busquedas Mensajes de PreCompilado C++ Log de generación Mensajes de generación Depuración

Archivo	Línea	Columna	Mensaje
Hola.cpp			Inicio de precompilador C++.
Hola.cpp	4	10	Proceso de precompilación C++.
Hola.cpp			>Error: El programa principal debe tener argumentos en C++.
Ejemplos\Hola.cpp			Error sintáctico al analizar en C++.
			default Línea 4. Columna 1 Insertar

Figura 5.44: Ejemplo de error debido a declaración de argumentos en `main`

5.6.2. Errores relacionados con el uso de expresiones

El cuadro 5.3 resume los errores relacionados con el uso de expresiones. Los errores más habituales se suelen producir al no completar las expresiones de forma correcta. Por ejemplo al escribir expresiones que no tengan sentido o cometer errores al olvidar algún operando u operador, tal y como se muestra en la figura 5.45

Lo habitual es que el preproceso C \pm genere un mensaje relacionado con los operadores afectados por errores en la expresión o que se genere un error de tipo genérico con el mensaje “Elemento no esperado” y una colección de los símbolos esperados como en el ejemplo incluido en la figura 5.45. El tratamiento pasará por revisar la expresión programada.

Error	Descripción
el operador ...	Error relacionado con la sintaxis de algún operador en C \pm
después de ... no no se puede usar ...	En el lenguaje C \pm las expresiones pueden construirse todo lo complejas que se quiera pero siempre respetando la estructura de los operadores y teniendo en cuenta el orden de evaluación
en la expresión	Error genérico relacionado con la sintaxis de la expresión

Cuadro 5.3: Errores relacionados con el uso de expresiones

```

UNED 2009 - Hola.cpp x
1 #include <stdio.h>
2
3
4 int main()
5 {
6     int a;
7
8     a = 5 ! a;
9
10    printf("Hola MUNDO \n");
11 }
12

```

Mensajes de Log y otros

Archivo	Línea	Columna	Mensaje
Hola.cpp			Inicio de precompilador C \pm .
Hola.cpp	8	11	Proceso de precompilación C \pm .
Hola.cpp			Error: Análisis C \pm . Elemento no esperado. Se esperaba: - / * % &) + , , / ; ; [] ...
Hola.cpp			Error sintáctico al analizar en C \pm .

Ejemplos/Hola.cpp default Línea 1. Columna 1 Insertar Lectura/Escritura

Figura 5.45: Ejemplo de error en el uso de una expresión

5.6.3. Errores relacionados con la declaración de constantes

El cuadro 5.4 resume los errores relacionados con la declaración de constantes. Éstos suelen deberse a que hemos olvidado alguno de los elementos necesarios en la declaración: el tipo de la constante, su identificador o su valor. La figura 5.46 muestra un error de este tipo.

Error	Descripción
... falta valor ...	Se ha declarado una constante sin valor
... falta tipo ...	Se ha declarado una constante sin tipo
...falta identificador ...	Se ha declarado una constante sin identificador

Cuadro 5.4: Errores relacionados con la declaración de constantes

The screenshot shows the Code::Blocks IDE interface. In the top editor window, there is a code file named "Hola.cpp". The code contains the following C++ code:

```

1 #include <stdio.h>
2
3
4 int main()
5 {
6     int a;
7     const a = 6;
8
9     printf("Hola MUNDO \n");
10 }
11
12

```

A cursor is positioned at the line "const a = 6;" where the identifier "a" is highlighted in red. Below the editor, the status bar displays the path "E:\ejemplos\Hola.cpp".

In the bottom panel, the "Mensajes de PreCompilado C++" tab is selected, showing the following error message:

Archivo	Línea	Columna	Mensaje
Hola.cpp			Inicio de precompilador C++..
Hola.cpp	7	13	Proceso de precompilación C++..
Hola.cpp			Línea 7: Columna 13: Declaración de constante INCORRECTA - Falta tipo de la constante
Hola.cpp			Error sintáctico al analizar en C++..

Figura 5.46: Ejemplo de error en la declaración de constantes

5.6.4. Errores relacionados con la declaración de subprogramas

El cuadro 5.5 resume los errores relacionados con la declaración de subprogramas. La figura 5.47 muestra un ejemplo de esta clase de error: el

tipo de la constante `pepe` no está definido en la cabecera del subprograma `aux`.

Error	Descripción
... sobra punto y coma ...	En el lenguaje C± utilizaremos un punto y coma después de la declaración de un subprograma. Sin embargo en la realización deberá comenzar el bloque de implementación sin punto y coma
... falta tipo o identificador...	En el caso de un subprograma con argumentos estos deben incluir el tipo y el identificador del argumento

Cuadro 5.5: Errores relacionados con la declaración de subprogramas

El preproceso C± genera en estos casos un error sobre la parte de la declaración afectada por lo que el tratamiento pasará por revisar que todos los elementos están correctamente incluidos.

The screenshot shows a Code::Blocks IDE window. The code editor displays the following C++ code:

```

UNED 2009 - *Hola.cpp *
1 #include <stdio.h>
2
3 void aux(const pepe);
4
5 int main()
6 {
7     printf("Hola MUNDO \n");
8 }
9

```

The line `void aux(const pepe);` is highlighted in red, indicating a syntax error. Below the editor, the 'Messages' tab is selected, showing the following log output:

Archivo	Línea	Columna	Mensaje
Hola.cpp	3	20	Inicio de precompilador C+/-.
Hola.cpp			Proceso de precompilación C+/-.
Hola.cpp			Línea 3: Columna 20: Falta el identificador o el tipo del argumento de la constante
Hola.cpp			Error sintáctico al analizar en C+/-.

Figura 5.47: Ejemplo de error en la declaración de subprogramas

5.6.5. Errores relacionados con la declaración de variables

El cuadro 5.6 resume los errores relacionados con la declaración de variables. Por ejemplo, en la figura 5.48 se comete el error de inicializar una variable en una lista de declaración.

En estos casos, el preproceso genera los mensajes de error sobre las variables afectadas y debemos repasar que se cumplen los requisitos que se han establecido en C± para la declaración de variables.

Error	Descripción
... iniciar variable ...	En el lenguaje C± en caso de usar una lista de declaración de variables no podremos iniciar su valor
... declaración variable ...	En el lenguaje C± no se pueden declarar variables anónimas. La declaración de variables exige el uso de tipos predefinidos o tipos previamente declarados

Cuadro 5.6: Errores relacionados con la declaración de variables

```

1 #include <stdio.h>
2
3 int a, b=1, c;
4
5 int main()
6 {
7     printf("Hola MUNDO \n");
8 }

```

The screenshot shows the UNED 2009 - Hola.cpp window in Code::Blocks. The code contains a declaration list in the main() function where variable 'a' is initialized. A red box highlights this line. The status bar at the bottom displays the error message: "Error : No se puede iniciar una variable en una lista de declaración".

Figura 5.48: Ejemplo de error en la declaración de variables

5.6.6. Errores relacionados con el uso de **for**, **if**, **switch**, **while** y **dowhile**

El cuadro 5.7 resume los errores relacionados con el uso de las estructuras **for**, **if**, **switch**, **while** y **dowhile**. Típicamente, estos errores proceden de las restricciones impuestas en C \pm que no existen en C++. Por ejemplo en C \pm todos los bloques van entre delimitadores llaves, mientras que en C++ pueden no usarse si el bloque tiene una única sentencia, como por ejemplo en la figura 5.49.

Generalmente, Code::Blocks^{C \pm} nos indicará la sentencia donde se ha producido el error. En algunos casos, también se pueden generar mensajes de tipo general sobre todo relacionados con la declaración en un bloque ejecutivo o de elementos no esperados.

Error	Descripción
... Falta llave ...	En el lenguaje C \pm , todas las estructuras (for , if ...) deben usar el delimitador llave aunque sólo contengan una sentencia
... Falta cerrar ...	Error si se ha olvidado cerrar las expresiones condicionales de las estructuras o el delimitador llave de cierre del bloque

Cuadro 5.7: Errores relacionados con el uso **for**, **if**, **switch**, **while** y **dowhile**

5.6.7. Otros errores

El cuadro 5.8 resume otros errores comunes en C \pm . Además, incluimos la siguiente lista de errores frecuentes en C \pm :

- En las sentencias **try**, **throw** y **catch** deben tenerse en cuenta las restricciones de C \pm que, sin embargo, no existen en C++. Por ejemplo,

The screenshot shows the Code::Blocks IDE interface. The main window displays the code file `Hola.cpp` with the following content:

```
UNED 2009 - | Hola.cpp |
1 #include <stdio.h>
2
3
4 int main()
5 {
6     while (true) printf("Hola MUNDO \n");
7 }
8
```

The cursor is at the end of the opening brace of the `main()` function. Below the code editor, the status bar shows the message "Error sintáctico al analizar en C++". The bottom part of the interface shows the "Messages" tab of the IDE's message window, which contains the following log entries:

Archivo	Línea	Columna	Mensaje
Hola.cpp			Inicio de precompilador C++.
Hola.cpp			Proceso de precompilación C++.
Hola.cpp	6	17	Error al analizar el código. Falso final en el inicio de la sentencia while.
Hola.cpp			Error sintáctico al analizar en C++.

Figura 5.49: Ejemplo de error en el uso de la sentencia `while`

si hay captura después del `try` entonces sólo se puede usar un `catch` y no varios.

- Utilización de una variable sin inicializar.
- Utilizar un índice de formación mayor que la longitud de la formación.
- Omitir un punto y coma, o un final de llave.
- Utilizar un puntero sin inicializar.
- Utilizar una asignación `=`, cuando lo que realmente deseamos expresar es una comparación `==`.
- Sobreescribir u omitir el terminador nulo de una cadena.
- Especificar los valores de variables en un `scanf()` en lugar de sus direcciones.
- Fallar al declarar el tipo de retorno de una función.
- Omitir un `break` en una sentencia `case` (la ejecución continúa en los `cases` sucesivos).

- Utilizar **break** para salir de un bloque de código asociado con una sentencia **if** (el **break** sirve para salir de bloques de código con un **for**, **switch** o **while**).

Error	Tratamiento
Falta punto y coma	Revisar la línea en la que se produce el mensaje para incluir el punto y coma.
No se puede declarar en el bloque ejecutivo	En C± no se pueden mezclar las declaraciones con las sentencias del bloque ejecutivo. Se debe revisar donde se está declarando en la parte ejecutiva. Esta declaración se debe mover a un bloque declarativo apropiado.
Error en lista	Error que se produce al declarar listas de items en valores o definiciones de tipos. Se debe revisar que la lista se ha escrito correctamente según C±
Error en expresión	Caso genérico de error en las expresiones. Se debe revisar cuál es la expresión errónea.
Elemento no esperado, se esperaba ...	Caso genérico de error. Este error se mostrará cuando Code::Blocks ^{C±} no puede proporcionar más información sobre la situación errónea que se produce. Al mostrar este error, se indica también cuáles son los elementos que se esperaban recibir y no han llegado. Estos elementos esperados pueden ser una pista importante para localizar el contexto en el que se produce el error y poder tratarlo.

Cuadro 5.8: Otros errores comunes en C±

Capítulo 6

Segunda práctica: rombos

ESTE capítulo presenta el enunciado de la segunda práctica. De dificultad creciente respecto a la primera práctica, requiere el uso de nuevos elementos del lenguaje C_±: variables, expresiones de condición, uso de esquemas de selección e iteración... En el capítulo se dan las pautas metodológicas para resolver la práctica, incidiendo en cuáles son las consecuencias negativas de no reflexionar suficientemente antes de codificar una solución y en cuál es la forma correcta de pensar para resolver problemas complejos. Por último, el capítulo incluye una descripción detallada del proceso de corrección automática de la segunda práctica.

6.1. Enunciado de la práctica

Esta práctica consiste en realizar un programa que imprima por pantalla rombos concéntricos dibujados con los caracteres '@', 'o' y '.'.

El programa solicitará, como dato de entrada, la longitud del lado del rombo más externo (es decir, su número de caracteres). De fuera hacia dentro, el primer rombo estará formado por caracteres '@', el segundo por caracteres '.', el tercero por caracteres 'o', el cuarto nuevamente por caracteres '.', el quinto por caracteres '@', el sexto también por caracteres '.', el séptimo por caracteres 'o' y así sucesivamente.

Las figuras 6.1, 6.2, 6.3, 6.4, 6.5 y 6.6 muestran los resultados de ejecutar el programa para lados de longitud 1, 2, 3, 4, 5 y 10.

```
¿Lado del Rombo?1  
@
```

Figura 6.1: Ejecución del programa para lado = 1

```
¿Lado del Rombo?2  
@  
@. @  
@
```

Figura 6.2: Ejecución del programa para lado = 2

```
¿Lado del Rombo?3  
@  
@. @  
@. o . @  
@. @  
@
```

Figura 6.3: Ejecución del programa para lado = 3

```
¿Lado del Rombo?4  
@  
@. @  
@. o . @  
@. o . o . @  
@. o . @  
@. @  
@
```

Figura 6.4: Ejecución del programa para lado = 4

Como se puede observar, el rombo siempre debe quedar ajustado a la izquierda de la pantalla y estar separado con una línea en blanco después de la pregunta ¿Lado del Rombo?

```
¿Lado del Rombo?5
```

```
@  
@.  
@.o.  
@.o.o.  
@.o.@.o.  
@.o.o.  
@.o.  
@.  
@
```

Figura 6.5: Ejecución del programa para lado = 5

```
¿Lado del Rombo?10
```

```
@  
@.  
@.o.  
@.o.o.  
@.o.@.o.  
@.o.@.o.o.  
@.o.@.o.o.o.  
@.o.@.o.o.o.o.  
@.o.@.o.o.o.o.o.  
@.o.@.o.o.o.o.o.o.  
@.o.@.o.o.o.o.o.o.  
@.o.@.o.o.o.o.o.  
@.o.@.o.o.o.  
@.o.  
@.  
@
```

Figura 6.6: Ejecución del programa para lado = 10

El tamaño del rombo más grande será de 20 caracteres de lado. El programa no deberá imprimir nada para tamaños cero o negativo, ni para tamaños superiores a 20.

El programa debe pedir ¿Lado del Rombo? sólo una vez en cada ejecución. Si el lado es erróneo, la ejecución del programa finalizará. Es decir, nunca debe haber un bucle indefinido que solicite un valor de entrada repetidamente hasta que sea correcto.

6.2. Metodología de desarrollo del algoritmo

Para la elaboración de cualquier algoritmo es muy recomendable pensar sobre papel en la solución. Si en lugar de reflexionar nos abalanzamos sobre el ordenador para programar la primera idea que se nos venga a la cabeza, acabaremos invirtiendo mucho tiempo en detectar y solventar las carencias del programa inicial hasta obtener una solución satisfactoria. En lugar de dedicar el tiempo en pulir un mal programa, deberemos invertirlo en diseñar un buen programa desde el principio.

El problema del rombo es suficientemente complejo para que sea difícil encontrar una solución directa del mismo. Si tratamos de resolverlo en un sólo paso, tratando de pensar a la vez cómo imprimir el número adecuado de rombos concéntricos según la longitud del lado del rombo más externo, cómo seleccionar los caracteres correspondientes, cómo situar el rombo justificado a la izquierda, etc. nos veremos abrumados¹. Por el contrario, conviene abordar la construcción del algoritmo mediante la técnica de refinamientos sucesivos², descomponiendo de forma progresiva el problema general en subproblemas más sencillos hasta alcanzar problemas elementales fáciles de resolver. La figura 6.7 ilustra la descomposición que desarrollaremos en las secciones 6.2.1 y 6.2.2.

6.2.1. División del rombo en triángulos

En lugar de imprimir directamente los rombos, descompondremos su impresión en figuras geométricas más sencillas. Concretamente, en triángulos.

$$\text{Imprimir rombos} \rightsquigarrow \begin{cases} 1) \text{ Imprimir triángulos superiores} \\ 2) \text{ Imprimir triángulos inferiores} \end{cases}$$

Por ejemplo, las figuras 6.8 y 6.9 muestran el resultado de los subproblemas 'Imprimir triángulos superiores' e 'Imprimir triángulos inferiores' para una lado de tamaño 10.

¹Los psicólogos consideran que la capacidad mental humana para trabajar simultáneamente con varios conceptos es muy limitada: tan sólo 7 ± 2 conceptos.

²ver el tema 4 del libro [1].

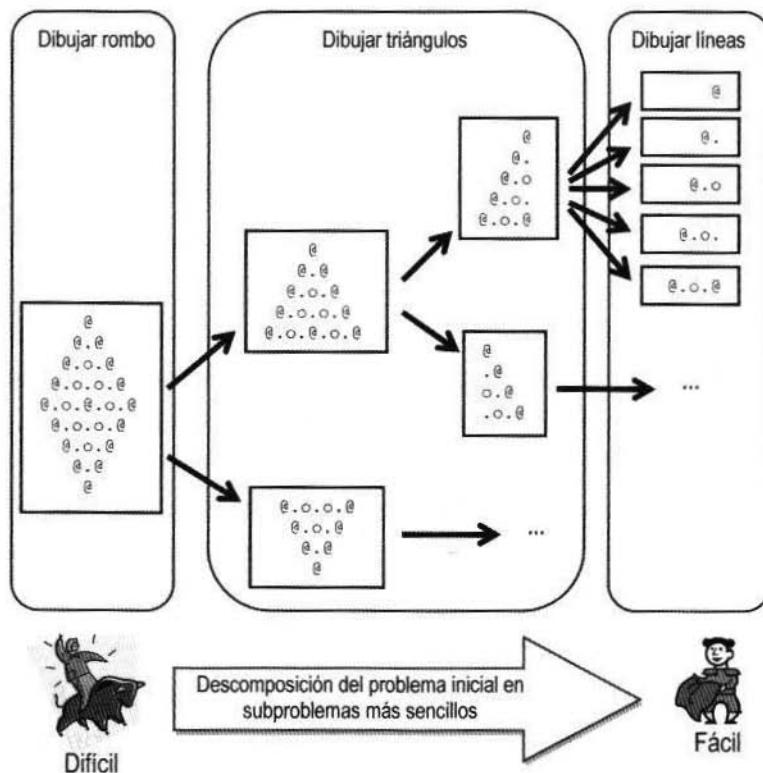


Figura 6.7: Aplicación de refinamientos sucesivos al problema del rombo

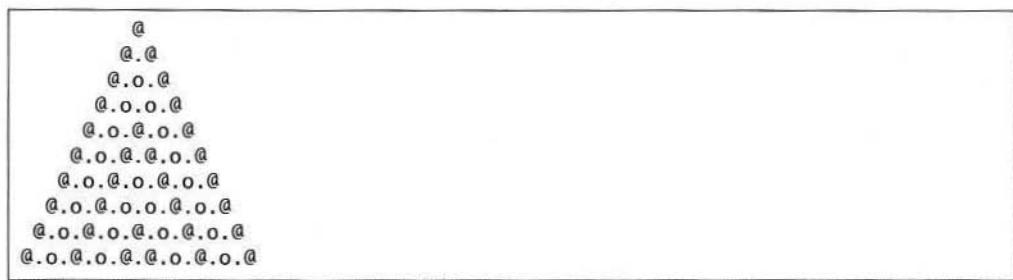


Figura 6.8: Resultado de 'Imprimir triángulos superiores' para lado = 10

```

@.o@.o@.o@.o@
@.o@.o.o@.o@.
@.o.o@.o@.o@.
@.o.o@.o@.o@.
@.o@.o@.o@.
@.o.o@.
@.o@.
@.o@.
@.

```

Figura 6.9: Resultado de ‘Imprimir triángulos inferiores’ para lado = 10

Gracias a la técnica de refinamientos sucesivos, en cada paso centraremos nuestra atención en un subproblema en concreto e ignoraremos el resto de los subproblemas. Por ejemplo, cuando resolvamos imprimir triángulos superiores sólo pensaremos en cómo realizar esta tarea, evitando pensar en cómo imprimir triángulos inferiores. Además, la descomposición propuesta aprovecha la simetría del rombo. Así, el esfuerzo invertido en implementar imprimir triángulos superiores podrá reutilizarse para construir imprimir triángulos inferiores.

De nuevo, puede aprovecharse la simetría de la figura 6.8 y utilizar la técnica de refinamientos sucesivos para descomponer imprimir triángulos superiores en dos nuevos subproblemas:

1) Imprimir triángulos superiores ~~

~~ { 1.1) Imprimir mitad izquierda triángulos superiores
 ~~ { 1.2) Imprimir mitad derecha triángulos superiores

Las figuras 6.10 y 6.11, muestran el resultado de ‘Imprimir mitad izquierda triángulos superiores’ e ‘Imprimir mitad derecha triángulos superiores’ para una lado de tamaño 10.

Comparando las figuras 6.10 y 6.11, observamos que la mitad derecha tiene una línea menos de altura que la izquierda (9 líneas en lugar de 10 líneas). Además, la secuencia de los caracteres ‘@’, ‘o’ y ‘.’ es inversa a la de la mitad izquierda. También se debe observar que la nueva secuencia inversa de esta mitad derecha debe comenzar por el siguiente carácter de la posición central. Hay que tener en cuenta que la columna central ya está incluida en la mitad izquierda.

6.2.2. División de un triángulo en líneas

El subprograma imprimir mitad izquierda triángulos superiores puede descomponerse en dos tareas elementales:

1.1) Imprimir mitad izquierda triángulos superiores \rightsquigarrow

$\rightsquigarrow \left\{ \begin{array}{l} 1.1.1) \text{ Imprimir espacios en blanco} \\ 1.1.2) \text{ Imprimir secuencia de caracteres} \end{array} \right.$

Si nos fijamos en la figura 6.10, observamos que ‘Imprimir espacios en blanco’ debe imprimir 9 blancos en la primera línea, 8 en la segunda, 7 en la tercera, y así sucesivamente hasta no imprimir ningún blanco en la décima línea. Por lo tanto, el número de blancos depende del índice de la línea y del tamaño del lado del rombo:

$$\text{número de blancos} = \text{lado del rombo} - \text{índice de la línea}$$

1	@
2	@.
3	@.o
4	@.o.
5	@.o.@
6	@.o.@.
7	@.o.@.o
8	@.o.@.o.
9	@.o.@.o.@
10	@.o.@.o.@.

Figura 6.10: Resultado de ‘Imprimir mitad izquierda triángulos superiores’ para lado = 10

Veamos cómo imprimir la secuencia de caracteres. La última línea de la figura 6.10 es “@.o.@.o.@.”. La primera posición de la línea corresponde al carácter ‘@’, la segunda a ‘.’, la tercera a ‘o’, la cuarta a ‘.’ y, a partir de aquí, se repite el patrón: la quinta posición corresponde al carácter ‘@’, la sexta a ‘.’, la séptima a ‘o’, la octava a ‘.’, etc. Tal y como se indica en la figura 6.12, esta secuencia cíclica puede implementarse utilizando el operador % de resto de la división.

```

1 @
2 .@
3 o.@
4 .o.@
5 @.o.@
6 .@.o.@
7 o.@.o.@
8 .o.@.o.@
9 @.o.@.o.@

```

Figura 6.11: Resultado de 'Imprimir mitad derecha triángulos superiores' para lado = 10

$$\text{posición \% } 4 = \begin{cases} 0 \rightarrow '@' \\ 1 \rightarrow '.' \\ 2 \rightarrow 'o' \\ 3 \rightarrow '@' \end{cases}$$

0	1	2	3
@	.	o	.

Figura 6.12: Uso del operador %, de resto de la división, para obtener los caracteres de una línea del rombo

6.3. Autocorrección de la práctica

Todas las prácticas deben tener la misma cabecera. Para evitar errores, recomendamos que el alumno edite el código fuente de la segunda práctica a partir del fichero `practica_2.cpp`, generado automáticamente tras la corrección de la práctica 1.

Una vez editado el código de la práctica 2, comprobaremos su corrección seleccionando `Prácticas → Comprobar la práctica 2` en nuestro entorno de programación para C±. La figura 6.13 resume esquemáticamente las comprobaciones que realiza el corrector automático.

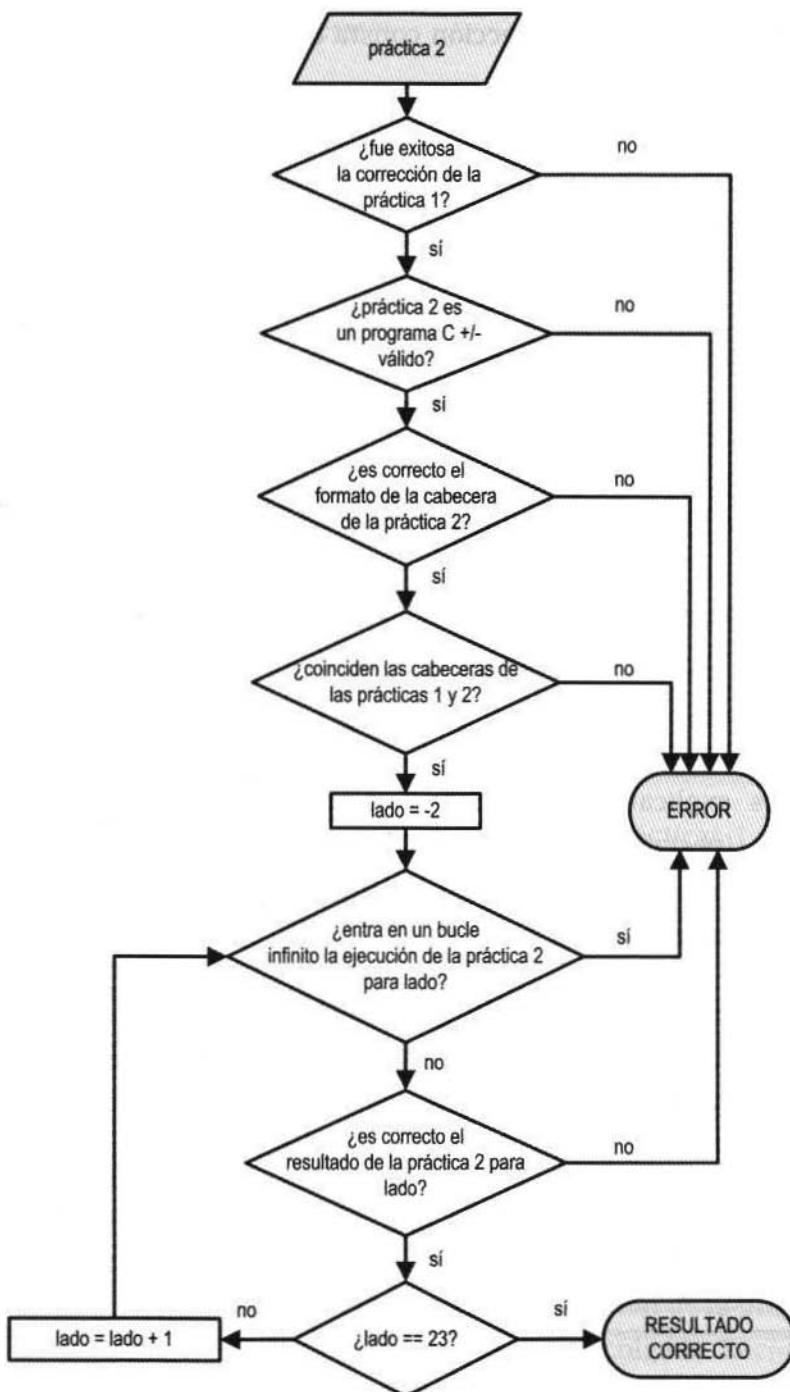


Figura 6.13: Corrección automática de la práctica 2

El proceso de autocorrección consta de los siguientes pasos:

1. ¿Fue exitosa la corrección de la práctica 1?

Es condición indispensable para corregir la práctica 2 que la práctica 1 haya superado el proceso de corrección automática. En caso contrario se visualizará el mensaje de error de la figura 6.14.

```
=====
<<ERROR>>: antes de corregir esta practica,
debe pasar la correccion de la practica 1
=====
```

Figura 6.14: Mensaje de error: se ha intentado corregir la práctica 2 sin que la práctica 1 pasara el proceso de corrección automática

2. ¿El programa 'Práctica 2' es válido según el lenguaje C±?

Si el código fuente de la práctica 2 no satisface las restricciones sintácticas y semánticas de C± aparecerá un mensaje de error. Para solventarlo, conviene consultar los mensajes explicativos que aparecen en la parte inferior de la pantalla (pestañas *Log de PreCompilado C±*, *Log de generacion* y *Mensajes de generacion*) tal y como se comentaba para la primera práctica en la figura 4.4).

3. ¿Es correcto el formato de la cabecera de la práctica 2?

Si la cabecera de la práctica es incorrecta se generará un mensaje de error. Por ejemplo, la figura 6.15 muestra el mensaje correspondiente a la cabecera errónea incluida en las figura 6.16.

```
=====
<<ERROR>>: Error en la cabecera del fichero
'C:\cmasmenos\practicas\practica_2.cpp',
revise el campo 'PRIMER APELLIDO'
=====
```

Figura 6.15: Mensaje de error provocado por la cabecera de la figura 6.16

4. ¿Coinciden las cabeceras de las prácticas 1 y 2?

```
*****  
* NOMBRE: #Juan Antonio#  
* SEGUNDO APELLIDO: #Fernandez#  
* DNI: #99999999#  
* EMAIL: #jantonio.gonzalez@mimail.com#  
*****/
```

Figura 6.16: Ejemplo de cabecera incorrecta (falta el campo PRIMER APELLIDO)

Por ejemplo, si la figura 6.17 es la cabecera de la primera práctica y la figura 6.18 la cabecera de la segunda práctica, se visualizará el mensaje de error de la figura 6.19 porque el SEGUNDO APELLIDO de la primera práctica (Gonzalez) no coincide con el de la segunda (Martinez).

```
1 *****  
2 * NOMBRE: #Juan Antonio#  
3 * PRIMER APELLIDO: #Fernandez#  
4 * SEGUNDO APELLIDO: #Gonzalez#  
5 * DNI: #99999999#  
6 * EMAIL: #jantonio.gonzalez@mimail.com#  
7 *****/
```

Figura 6.17: Ejemplo de cabecera válida para las prácticas 1, 2 y 3

```
*****  
* NOMBRE: #Juan Antonio#  
* PRIMER APELLIDO: #Fernandez#  
* SEGUNDO APELLIDO: #Martinez#  
* DNI: #99999999#  
* EMAIL: #jantonio.gonzalez@mimail.com#  
*****/
```

Figura 6.18: Cabecera de la práctica 2

5. ¿Entra en un bucle infinito la ejecución de la práctica 2?

El corrector automático comprueba que la ejecución de la práctica no entra en un bucle infinito, es decir, no se queda colgada. Recordemos que la práctica debe pedir **¿Lado del Rombo?** sólo una vez para cada ejecución. Nunca debe tener un bucle indefinido pidiendo el lado múltiples veces.

6. ¿Es correcto el resultado de la práctica 2 para lado?

El entorno verifica si se satisface el enunciado de la práctica. Para ello, se ejecuta el código para lados de tamaño -2, -1, 0, ..., 23. Por ejemplo, la figura 6.20 incluye un fragmento del mensaje de error producido por una práctica incorrecta. Aunque la práctica funciona bien para lado = 2, produce un resultado erróneo para lado = 3.

```
=====
<<ERROR>>: en la cabecera de esta practica
el campo 'SEGUNDO APELLIDO' es distinto del
de las practicas anteriores
=====
```

Figura 6.19: Mensaje de error: los valores del SEGUNDO APELLIDO de las prácticas 1 y 2 no coinciden (ver figuras 6.17 y 6.18)

Si la práctica es correcta, se generará automáticamente el fichero `practica_3.cpp`, que incluirá la cabecera de la tercera práctica. Dicho fichero se localizará en la misma carpeta que la primera y segunda práctica.

Rombo correcto (para lado = 2):

```
@  
@.  
@
```

Rombo obtenido:

```
@  
@.  
@
```

Correcto!

<< ERROR >>

Rombo correcto (para lado = 3):

```
@  
@.  
@..@  
@.  
@
```

Rombo obtenido:

```
@  
@.  
@.x.  
@.  
@
```

Figura 6.20: Práctica incorrecta: funciona bien para lado = 2, pero produce un resultado erróneo para lado = 3

Capítulo 7

Depuración de programas

ESTE capítulo está dedicado a la depuración de programas con `Code::BlocksC±`. Como introducción, se presentan los conceptos teóricos básicos relacionados con la depuración de programas. En primer lugar, se explica cómo depurar un programa incluyendo sentencias de impresión `printf` y qué inconvenientes plantea este método. A continuación, se presentan las alternativas de depuración disponibles en `Code::BlocksC±` y cómo manejarlas. También se describen las distintas formas de visualizar la información relacionada con la depuración de un programa. Por último, se muestra cómo aplicar estos conceptos a un caso práctico. Los contenidos del capítulo se resumen esquemáticamente en la figura 7.1.

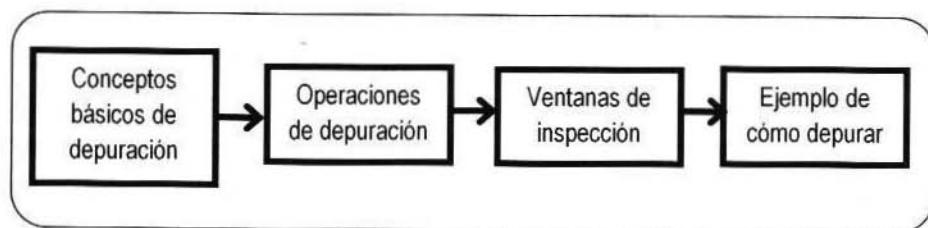


Figura 7.1: Estructura conceptual del capítulo

7.1. Conceptos básicos sobre la depuración de programas

La depuración (*debugging* en inglés) tiene como objetivo localizar los errores que hemos cometido al escribir un programa. La depuración de un error suele requerir los siguientes pasos:

- Reconocer que ese error existe (un programa puede contener errores que jamás serán detectados, por ejemplo, sentencias dentro de sentencias de condiciones que nunca se ejecutarán).
- Aislar la fuente del error.
- Identificar la causa del error.
- Determinar una solución para el error.
- Aplicar la solución.
- Comprobar que la solución resuelve el problema.

La depuración de un programa puede realizarse rudimentariamente incluyendo nuevas sentencias que impriman por pantalla mensajes sobre el estado de las variables (la sección 7.2 incluye un ejemplo de este tipo de estrategia). Este método, además de engorroso (hay que incluir nuevas sentencias `printf` por todo el código), es muy peligroso (podemos olvidar eliminar las nuevas sentencias una vez que hemos detectado el problema). A este respecto, Bruce Eckel cuenta como anécdota en su libro “Piensa en C++” que en un proyecto, donde su equipo de trabajo estaba desarrollando un programa para “electrodomésticos inteligentes”, alguien olvidó eliminar un mensaje de depuración. Como Eckel comenta, la detección de un error puede ser muy laboriosa y a medida que pasa el tiempo, los programadores tendemos a ser cada vez “más creativos” en los mensajes de depuración que imprimimos por pantalla. Meses después de vender el programa, algunos clientes de la empresa de Eckel se quejaron de que sus electrodomésticos les insultaban...

Para evitar este tipo de problemas, existen herramientas de ayuda conocidas como depuradores (*debuggers* en inglés) que permiten ejecutar un programa, detener el programa con pausas controladas, volver a comenzarlo o reanudar la ejecución desde la pausa, ejecutarlo por partes, ver o cambiar los valores de las variables, etc. Code::Blocks^{C±}incorpora el depurador de GNU para C/C++. Este depurador dispone de cuatro funciones principales para depurar los programas:

1. Iniciar el programa parametrizado.
2. Detener el programa ante condiciones determinadas.
3. Examinar el estado del programa detenido.
4. Cambiar las condiciones con el programa parado.

El punto en el que se detiene la ejecución de un programa se conoce como *punto de ruptura* o *de inturrupción* (*breakpoint* en inglés). Podemos definir un punto de ruptura como un punto de parada intencionada y controlada durante la ejecución de un programa. En un punto de ruptura podemos comprobar el estado de la memoria, las variables, los archivos, etc.

Podemos establecer un punto de ruptura en cualquier lugar ejecutable del código: asignaciones, sentencias de control, llamadas a subprogramas, etc. Sin embargo, no podemos poner puntos de ruptura en las partes no ejecutables del código: declaraciones, definiciones de tipos, comentarios, etc.

7.2. Ejemplo de depuración simple con mensajes

En este apartado se ha incluido un ejemplo sencillo sobre los conceptos relacionados con la depuración de programas. En concreto, y para iniciarse en este tipo de operaciones, se ha utilizado la estrategia de depuración con mensajes, que consiste en introducir mensajes que nos den información sobre las variables que se manejan en un programa. Para comenzar, vamos a partir de un programa que es válido en C± (ver figura

7.2). Este programa tiene como objetivo resolver la operación factorial de un valor dado por el usuario del programa. Si lo editamos y generamos, comprobaremos que es correcto en C[±] y que podemos crear el correspondiente ejecutable sin tener ningún tipo de error.

```
#include <stdio.h>

int main() {

    int num, factorial;
    printf ("Calcular el factorial de: ");
    scanf ("%d", &num );

    for (int i=1; i<num; i++) {
        factorial=factorial*i;
    }
    printf("El factorial de %d es %d",num,factorial);
}
```

Figura 7.2: Programa ‘factorial’

Sin embargo, al ejecutar el programa vamos a comprobar que el resultado obtenido no es el factorial esperado. Por ejemplo, si ejecutamos el programa y queremos calcular el valor del factorial de 6, se obtiene como resultado el mostrado en la figura 7.3.

¿Qué está ocurriendo? ¿por qué no se obtiene el resultado esperado? Code::Blocks^{C[±]}sólo es capaz de detectar automáticamente *errores de compilación* (en la sección 5.6 se describieron muchos de estos errores). Sin embargo, existe otro tipo de errores que sólo pueden detectarse una vez que el programa se ejecuta. Para detectar estos *errores de ejecución* utilizaremos las técnicas de depuración que se explican en este capítulo.

Para comprobar cómo se modifican los valores de nuestro programa, vamos a introducir una sentencia de impresión `printf` de los valores de las tres variables que se utilizan: `i`, `num` y `factorial` de forma que el código del programa queda como se muestra en la figura 7.4 (el nuevo `printf` está en las líneas 10-12).

La posición en la que situamos los mensajes de depuración dependerá de nuestras intenciones a la hora de buscar el error. Tenemos que tener en cuenta qué es lo que estamos buscando y cómo lo vamos a en-

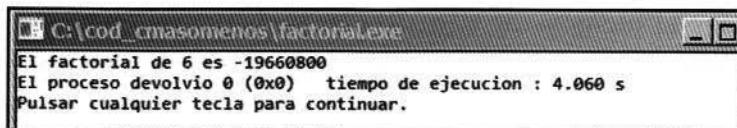


Figura 7.3: Resultado de ejecutar el programa 'factorial' de la figura 7.2

```

1 #include <stdio.h>
2
3 int main() {
4
5     int num, factorial;
6     printf ("Calcular el factorial de: ");
7     scanf ("%d", &num );
8
9     for (int i=1; i<num; i++) {
10        /* -- Sentencia de depuración -- */
11        printf("Factorial de %d,Iteracion %i,Acumulado %d \n",num,i,factorial);
12    }
13    factorial=factorial*i;
14 }
15 printf("El factorial de %d es %d",num,factorial);
16 }
```

Figura 7.4: Programa 'factorial' con mensajes de depuración

contrario. En el ejemplo, hemos detectado que el valor total no tiene sentido por lo que buscamos cómo cambian todas las variables involucradas en el cálculo. En este caso la salida que se obtiene en el programa es la mostrada en la figura 7.5

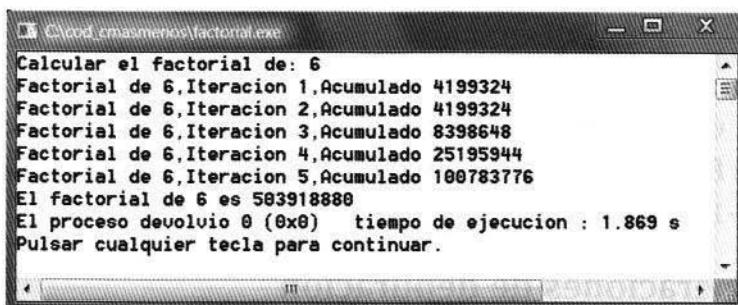


Figura 7.5: Resultado de ejecutar el programa 'factorial' de la figura 7.4

Gracias al mensaje de depuración, podemos ver que el valor de la variable `num` es correcto, que el índice de la sentencia `for` también es correcto y que el error se produce en el valor de la variable `factorial`. Esta variable tiene un valor inicial en el primer ciclo de la sentencia `for` de `2.147.348.480`. Es decir, estamos utilizando la variable `factorial` sin haberla inicializado adecuadamente. Para resolver este error, tenemos que utilizar una sentencia de inicialización para `factorial` con el valor que nos interese. Dado que la intención de la función es calcular el factorial, inicializaremos la variable a 1, tal y como se muestra en las líneas 9-11 de la figura 7.6.

```
1 #include <stdio.h>
2
3 int main() {
4
5     int num, factorial;
6     printf ("Calcular el factorial de: ");
7     scanf ("%d", &num );
8
9     /* -- Sentencia de inicialización de factorial -- */
10    factorial = 1;
11 }
12
13 for (int i=1; i<num; i++) {
14     /* -- Sentencia de depuración -- */
15     printf("Factorial de %d, Iteracion %i, Acumulado %d \n",num,i,factorial);
16
17     factorial=factorial*i;
18 }
19 printf("El factorial de %d es %d",num,factorial);
20 }
```

Figura 7.6: Programa ‘factorial’ correcto

Con esta sentencia de inicialización, tendremos la salida es la esperada en la figura 7.7. Sólo nos quedaría eliminar el mensaje de depuración (líneas 14-16 de la figura 7.6).

7.3. Operaciones de depuración

Las operaciones de depuración en `Code::Blocks` se encuentran en el menú `Depurar` (ver figura 7.8). Entre estas operaciones, encontramos la posibilidad de comenzar y detener la depuración, y diferentes opciones

```
C:\cod_cmasomenos\factorial.exe
Calcular el factorial de: 6
Factorial de 6 - Iteracion 1 - Acumulado 1
Factorial de 6 - Iteracion 2 - Acumulado 1
Factorial de 6 - Iteracion 3 - Acumulado 2
Factorial de 6 - Iteracion 4 - Acumulado 6
Factorial de 6 - Iteracion 5 - Acumulado 24
El factorial de 6 es 120
El proceso devolvio 0 (0x0) tiempo de ejecucion : 3.439 s
Pulsar cualquier tecla para continuar.
```

Figura 7.7: Resultado de ejecutar el programa ‘factorial’ de la figura 7.6

para controlar la ejecución de un programa: paso a paso, hasta un punto de ruptura, etc.

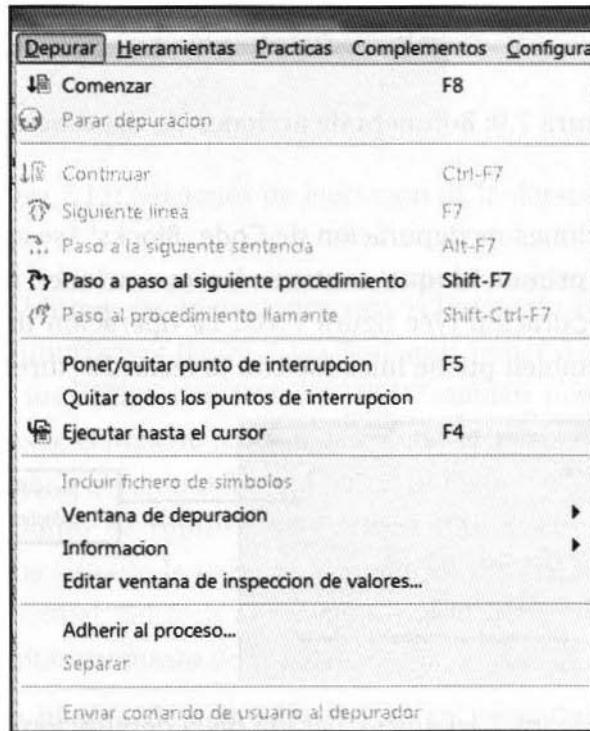


Figura 7.8: Menú *Depurar*

Además, se incluyen los accesos desde este menú a las distintas formas de presentar la información que se puede obtener en la depuración

que tenemos en el entorno. La presentación de esta información se realiza mediante diferentes ventanas de inspección como son: la de puntos de interrupción, la pila de llamadas, los registros de CPU, el desensamblado de la pila, el contenido de la memoria, los threads en ejecución y la ventana de inspección de valores.

Las acciones principales de depuración también son accesibles desde la correspondiente botonera *Debugger* (ver figura 7.9).

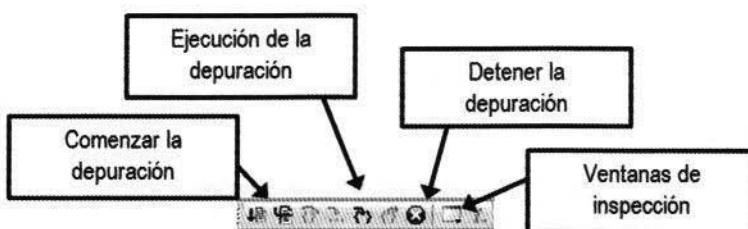


Figura 7.9: Botonera de acciones de depuración

Las operaciones de depuración de Code::Blocks^{C±} se agrupan en cuatro bloques. El **primer bloque** contiene las operaciones de comienzo y parada de la depuración (ver figura 7.10). La operación de comienzo de la depuración también puede iniciarse con la tecla **F8** directamente.

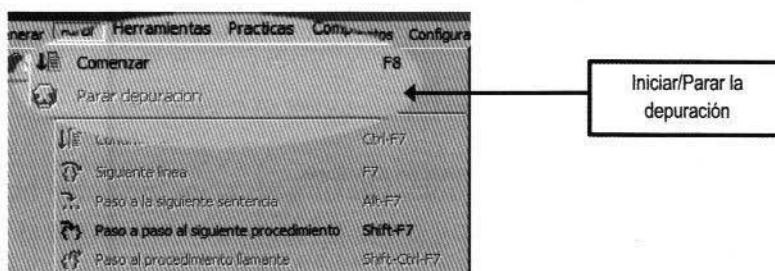


Figura 7.10: Inicio/parada de la depuración

El **segundo bloque** agrupa las operaciones de ejecución de la depuración (ver figura 7.11). Estas opciones permiten controlar la forma de ejecutar la depuración. Se establecen dos modos básicos, uno es ejecu-

tar línea a línea (con la tecla de acceso directo **F7**), y el otro modo es el de continuar la ejecución hasta encontrar un punto de ruptura o el fin del programa. Las otras opciones que disponemos permiten elegir la forma de funcionamiento en el caso de tener que ejecutar subprogramas o volver de un subprograma. Si no existe ningún punto de ruptura, el programa se ejecutará de forma normal, sin hacer ninguna parada.

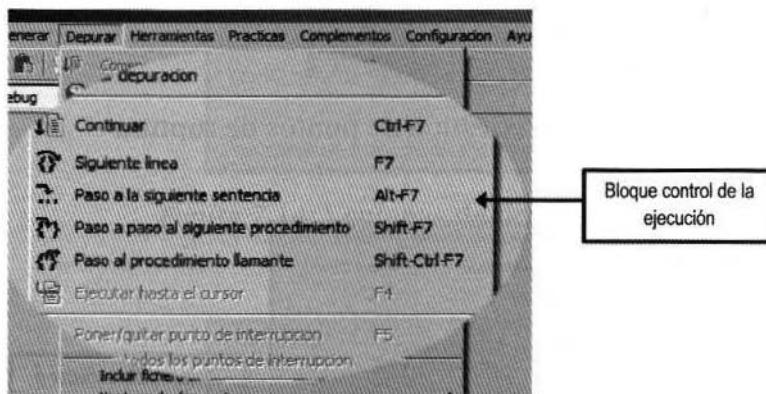


Figura 7.11: Opciones de ejecución de la depuración

El **tercer bloque** de operaciones está relacionado con la gestión de los puntos de ruptura (ver figura 7.12). Podemos poner o quitar los puntos de ruptura de un código con la tecla **F5**. También podemos acceder a esta gestión desde el área de edición, pulsando con el botón izquierdo del ratón en el borde izquierdo del texto (entre el texto y el número de línea). Al activar un punto de ruptura, se marcará con un círculo rojo la línea en la que hemos situado la parada. También es posible usar las opciones del menú contextual del área de edición (botón derecho del ratón) para establecer o quitar un punto de ruptura.

El **cuarto bloque** de operaciones está relacionado con el acceso a las ventanas que nos permiten obtener o seleccionar la información de depuración (ver figura 7.13). Estas opciones se comentan en detalle en el siguiente apartado.

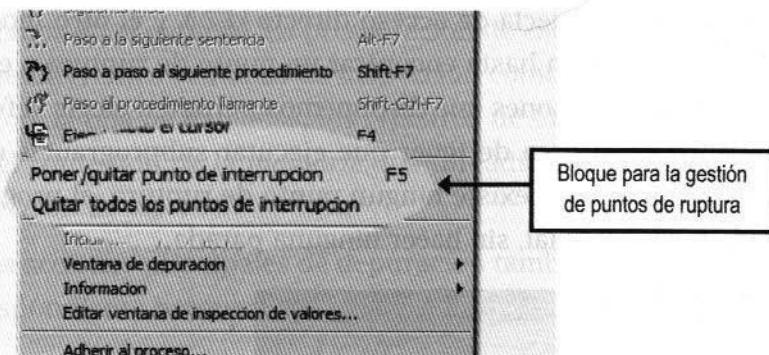


Figura 7.12: Gestión de puntos de ruptura

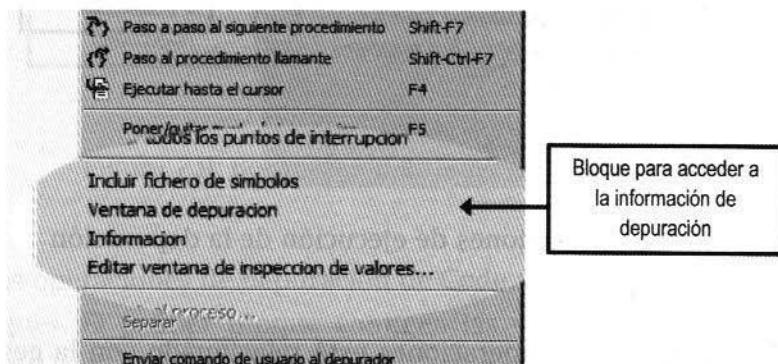


Figura 7.13: Menú de información sobre depuración

El resto de operaciones del menú **Depurar** facilitan la localización de errores de programación concurrente, cuyo estudio está fuera de nuestro curso de fundamentos de programación.

7.4. Ventanas de depuración e información

El menú de depuración de Code::Blocks^{C±} dispone de dos opciones para mostrar información de depuración:

1. **Depurar → Ventana de depuración** (figura 7.14).
2. **Depurar → Información** (figura 7.15).

Las ventanas de depuración tienen como objetivo mostrar la información sobre la ejecución del programa que se está depurando. Mientras que las opciones de información muestran datos sobre recursos del sistema en el que se ejecuta el mismo.

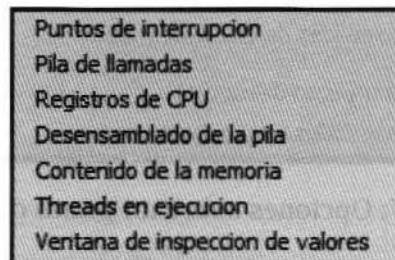


Figura 7.14: Ventanas de depuración

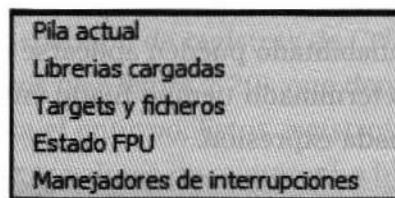


Figura 7.15: Información de depuración

Depurar → Ventana de depuracion → Puntos de interrupcion permite visualizar todos los puntos de ruptura incluidos en un programa (ver figura 7.16).

Puntos de ruptura			
Tipo	Fichero/Direccion	Línea	X
Code	C:/cod_cmasomenos/libro_practicas/ejemplos/mitest/main.cpp	11	
Code	C:/cod_cmasomenos/libro_practicas/ejemplos/mitest/main.cpp	14	
Code	C:/cod_cmasomenos/libro_practicas/ejemplos/mitest/main.cpp	15	

Figura 7.16: Ventana de puntos de ruptura

Además, si seleccionamos una de las filas con el control derecho del ratón podremos acceder a diferentes opciones que nos permite editar los puntos de ruptura activos (ver figura 7.17).

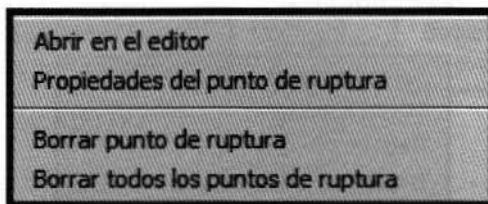


Figura 7.17: Opciones sobre un punto de ruptura

Si elegimos la opción de editar un punto de ruptura, podremos manipular determinadas propiedades específicas del punto (ver figura 7.18). Estas propiedades permiten deshabilitar la aplicación de un punto de ruptura. Las opciones de inhabilitado pueden aplicarse de forma permanente, temporal durante un determinado tiempo fijado en segundos, o mientras sea cierta una determinada expresión.

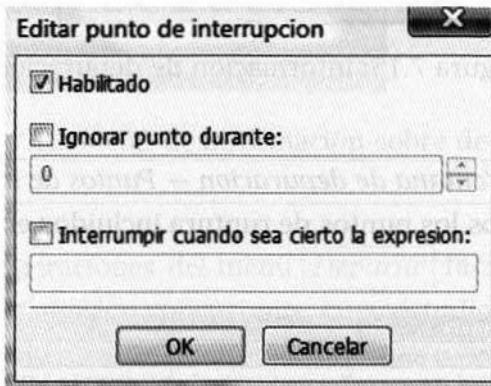


Figura 7.18: Edición de un punto de ruptura

La ventana de inspección de la pila de llamadas permite mostrar cuáles son los subprogramas que se encuentran cargados en memoria en un determinado momento de la ejecución de un programa (ver figura 7.19). En el caso de C±, como mínimo encontraremos siempre la refe-

rencia a la función global `main`. Esta ventana se puede consultar desde ***Depurar → Ventana de depuración → Pila de llamadas***. Desde esta ventana también podemos elegir diferentes operaciones de navegación a los subprogramas a partir del menú que se ofrece utilizando el menú contextual (botón derecho del ratón).

Nº	Dirección	Función	Fichero	Línea
0	00000000	func1()	C:/cod_cmasomenos/libro_...	5
1	0040142B	main()	C:/cod_cmasomenos/libro_...	13

Figura 7.19: Inspección de la pila de llamadas

La ventana de inspección de registros de CPU muestra los valores de los principales registros (ver figura 7.20), que dependen de cada máquina sobre la que se ejecuta el entorno. Estos registros contienen básicamente información para indicar valores de punteros¹ a bloques del programa como son el índice base (EBX), la pila de datos del programa (ESP) o el puntero de programa (EIP). Para mostrar esta ventana de inspección utilizaremos el menú ***Depurar → Ventana de depuración → Registros de CPU***.

La ventana de desensamblado permite conocer la información sobre las operaciones máquina que se están ejecutando (ver figura 7.21). En este tipo de ventanas de inspección podemos ver la correspondencia entre el código fuente y los diferentes direccionamientos internos que se utilizan en código máquina. Con el menú, ***Depurar → Ventana de depuración → Desensamblado de la pila***, podemos acceder a esta ventana de información.

La ventana de inspección del contenido de la memoria nos permitirá visualizar los valores almacenados en memoria (ver figura 7.22). Esta inspección es especialmente útil para comprobar valores de tipos no simples: enumerados, listas, formaciones, cadenas, etc. que son almacenadas

¹El concepto de puntero se explica en el texto básico de referencia [1].

Register	Hex	Integer
eax	0xe	14
ecx	0x759f32c9	1973367497
edx	0x772f5f34	1999593268
ebx	0x7ffd000	2147348480
esp	0x22fef0	2293488
ebp	0x22fef8	2293496
esi	0x0	0
edi	0x0	0
eip	0x4013d4	4199380
eflags	0x206	518
cs	0x1b	27
ss	0x23	35
ds	0x23	35
es	0x23	35
fs	0x3b	59
gs	0x0	0

Figura 7.20: Valores de registros de CPU en ejecución

Desensamblado de la memoria

Función: func1()
Comienzo: 0022FF00

004013CE	push	%ebp
004013CF	mov	%esp,%ebp
004013D1	sub	\$0x8,%esp
004013D4	movl	\$0x444000,(%esp)
004013DB	call	0x4165d0 <printf>
004013E0	leave	
004013E1	ret	

< !!! >

Guardar en un fichero de texto

Figura 7.21: Desensamblado de la memoria del programa en ejecución

en los programas como punteros a la memoria y podemos acceder desde el menú, **Depurar → Ventana de depuración → Contenido de la memoria**

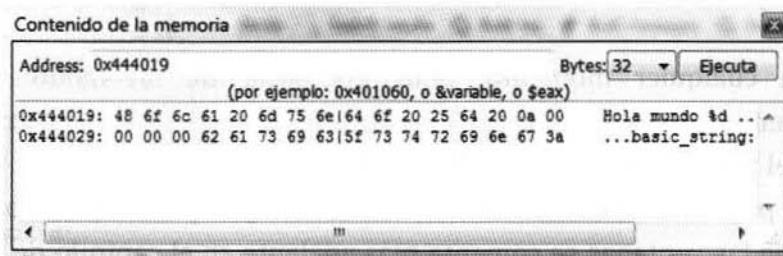


Figura 7.22: Inspección del contenido de la memoria en la posición 0x444019

Como se muestra en la figura anterior, todos los valores referidos en las ventanas se encuentran en formato hexadecimal, por lo que si queremos comprobar los valores que realmente se almacenan deberemos utilizar el correspondiente programa de conversión (de hexadecimal a códigos ASCII, a valores enteros...).

La ventana de inspección de valores es accesible desde **Depurar → Ventana de depuración → Ventana de inspección de valores** y permite consultar el valor de las diferentes variables manejadas en un programa o los argumentos que se utilizan en una determinada llamada a subprograma (ver figura 7.23). En esta ventana, podemos consultar tanto las variables locales como cualquier otra que queramos añadir para la inspección. Desde el menú contextual de esta ventana podremos realizar todas las operaciones de gestión sobre los elementos que se están inspeccionando: incluir nuevos, borrar o incluso cambiar el valor actual de alguno de ellos.

Cada una de las opciones anteriores puede manejarse de dos formas:

1. Como ventanas flotantes independientes, que tienen su propio “control de ventanas”. Es decir, podemos moverlas, ampliarlas o cerrarlas sin afectar al resto del entorno.

2. Como ventanas empotradas en el entorno, de forma que ocupen alguno de los espacios del entorno, ya sea en el área de gestión de proyectos, en el área de edición o en el de los mensajes.

En cualquier momento, podemos pasar de un modo de funcionamiento a otro. Si tenemos la ventana empotrada, podemos arrastrarla fuera del área para que quede flotante y viceversa; si está como ventana flotante podemos moverla hasta un área en la que quede fija. Esta organización de las ventanas se comenta más adelante en el capítulo de conceptos avanzados.

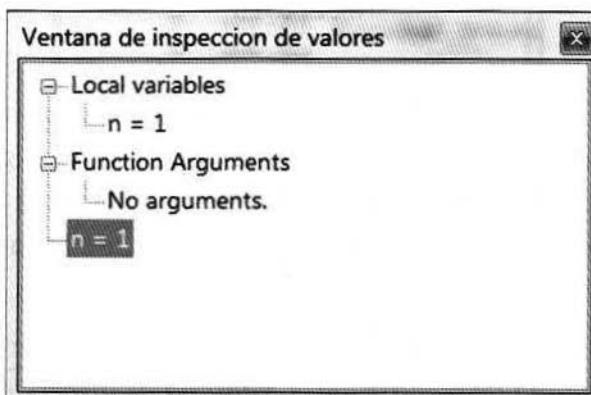


Figura 7.23: Ventana de inspección de valores

7.5. Ejemplo de depuración

En esta sección, utilizaremos el programa de la figura 7.24 para explicar el uso del depurador de *Code::Blocks^{C++}*. En primer lugar, para poder depurar un programa, *Code::Blocks^{C++}* exige que esté incluido dentro de un *proyecto*². Para ello, seleccionaremos *Proyecto → Crear proyecto desde fichero único*.

²En el capítulo 9 se describirá en profundidad el manejo de proyectos en *Code::Blocks^{C++}*, cuyo principal propósito es la gestión de programas compuestos por varios ficheros.

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <ctype.h>
4
5 typedef char* p_char;
6 typedef char cadena[25];
7
8 p_char PasoMAY(cadena szCad) {
9     int i=0;
10    while ( szCad[i] != NULL) {
11        szCad[i] = toupper(szCad[i]);
12        i++;
13    }
14    return szCad;
15}
16
17 int main() {
18     cadena szCadena;
19
20     strcpy(szCadena,"Esto es otra cadena");
21     printf("%s\n", PasoMAY(szCadena));
22     printf("Entre cadenas");
23     printf("%s\n", PasoMAY("Esto es una cadena"));
24
25     return 0;
26 }
```

Figura 7.24: Ejemplo de programa para ilustrar el uso del depurador del entorno

Si ejecutamos el programa anterior, se producirá un error en tiempo de ejecución (ver figura 7.25): no se imprimen las cadenas esperadas y además se devuelve un valor del programa con error.

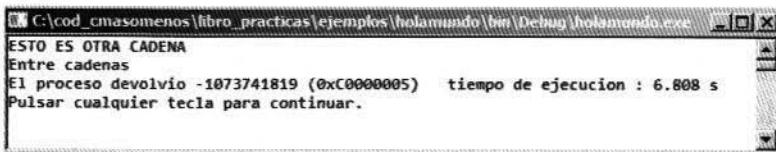


Figura 7.25: Error de ejecución del programa de la figura 7.24

Si lanzamos una sesión del depurador (*Depurar → Comenzar* o la tecla *F8*), veremos que se produce el mensaje de error de la figura 7.26.

Este error nos indica que se ha producido un intento de acceso a una parte de memoria que no se puede modificar. Además, se nos pregunta

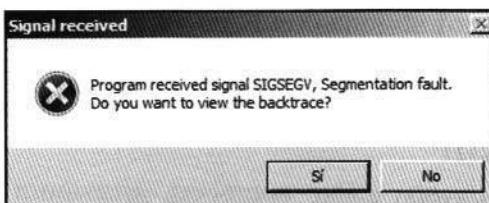


Figura 7.26: Error producido al lanzar el depurador sobre el programa de la figura 7.24

si queremos ver la traza de ejecución. Si contestamos afirmativamente, se nos mostrará la ventana de inspección de la pila de llamadas (ver figura 7.27).

Pila de llamadas			
Nr	Direc...	Funcion	Fichero
			Línea
0	00401302	Paso_MAY(szCad=0x403026 "Esto es una cadena")	C:/cod_cmason...
1	00401384	main()	C:/cod_cmason...

Figura 7.27: Pila de llamadas después del error

Esta ventana nos indica el punto en el que existe el error en ejecución. Concretamente en la línea 11, que contiene la sentencia:

```
szCad[i] = toupper(szCad[i]);
```

Esta sentencia pertenece al subprograma **PasoMAY**. Si se observa el código de la figura 7.24, no queda claro en cuál de las dos llamadas a **PasoMAY** se produce el error (línea 21 ó 23?). Podríamos utilizar la ejecución paso a paso del depurador para conocer qué sentencias se ejecutan realmente en el programa, pero a la vista del código es bastante evidente que el error se produce en la segunda llamada, ya que el argumento que se pasa al subprograma **PasoMAY** es una cadena literal constante que no puede ser modificada en la asignación.

También es interesante comprobar la utilidad del depurador para ver cómo cambian los valores de las variables en el programa. Si eje-

cutamos paso a paso el programa desde la primera sentencia `strcpy` (línea 20 de la figura 7.24), en la ventana de inspección de valores, menú *Depurar → Ventana de depuración → Ventana de inspección de valores*, podemos comprobar cómo cambia el valor de las variables. Por ejemplo, antes de la sentencia `strcpy`, la inspección del valor nos muestra el contenido de la variable `szCadena` sin iniciar (ver figura 7.28). A continuación de la llamada al subprograma `strcpy`³, al ejecutar el programa paso a paso el programa con **F7** o con el correspondiente botón de la botonera *Debugger*, veremos que la variable `szCadena` adquiere el nuevo valor asignado (ver figura 7.29).

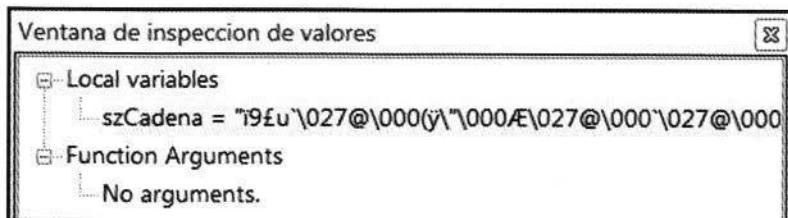


Figura 7.28: szCadena sin inicializar

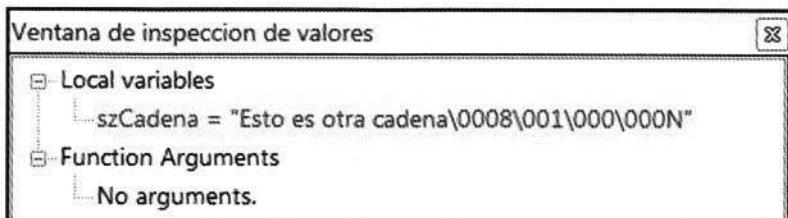


Figura 7.29: szCadena tras la ejecución de strcpy

Después de ejecutar `printf` (línea 21), donde se produce una llamada al subprograma `PasoMAY`, el valor de la cadena habrá pasado a mayúsculas (ver figura 7.30).

³La operación `strcpy` se comenta en el apartado sobre cadenas (`strings`) del texto básico de referencia [1].

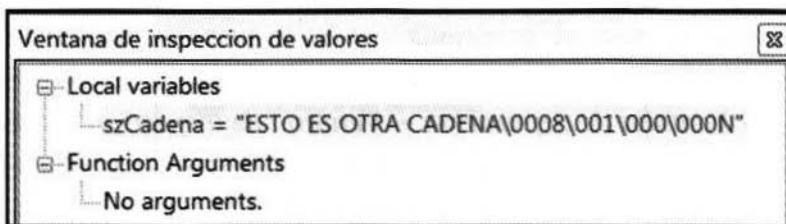


Figura 7.30: szCadena tras la ejecución de PasoMAY

Capítulo 8

Tercera práctica: calendario

ESTE capítulo incluye el enunciado de la tercera práctica, pautas metodológicas sobre cómo resolverla, una descripción de su proceso de corrección automática y sobre su entrega telemática al equipo docente de la UNED. Con el objetivo de que el alumno reflexione sobre la variedad de problemas que contempla la práctica y aprenda los fundamentos de la prueba de programas, el capítulo incluye una breve introducción sobre la técnica de pruebas basada en la partición de clases de equivalencia.

8.1. Enunciado

Se trata de realizar un programa que muestre por pantalla la hoja de calendario de cualquier mes y año comprendido entre los años 1.601 y 3.000. El formato de la hoja de calendario deberá ajustarse exactamente a los que se indican en las figuras 8.1 para el ejemplo del mes de abril de 2.345 y 8.2 para el ejemplo del mes de septiembre de 2010.

Como se puede observar, la hoja debe quedar ajustada a la izquierda de la pantalla y se deben emplear los caracteres '=', '|', '.' y los espacios en blanco tal y como aparecen en el ejemplo. Además, debe haber una línea en blanco antes de escribir la hoja del calendario.

El programa no deberá imprimir nada para los años fuera del rango de 1.601 al 3.000.

```

1 ¿Mes (1..12)?4
2 ¿Año (1601..3000)?2345
3
4 ABRIL           2345
5 =====
6 LU MA MI JU VI | SA DO
7 =====
8 . . . . . | . 1
9 2 3 4 5 6 | 7 8
10 9 10 11 12 13 | 14 15
11 16 17 18 19 20 | 21 22
12 23 24 25 26 27 | 28 29
13 30 . . . . | . .

```

Figura 8.1: Hoja de calendario para abril de 2345

```

1 ¿Mes (1..12)?9
2 ¿Año (1601..3000)?2010
3
4 SEPTIEMBRE      2010
5 =====
6 LU MA MI JU VI | SA DO
7 =====
8 . . 1 2 3 | 4 5
9 6 7 8 9 10 | 11 12
10 13 14 15 16 17 | 18 19
11 20 21 22 23 24 | 25 26
12 27 28 29 30 . | .

```

Figura 8.2: Hoja de calendario para septiembre de 2010

El programa debe pedir el mes y el año sólo una vez para cada ejecución. Nunca debe tener un bucle indefinido pidiendo múltiples veces el mes y el año.

8.2. Metodología de desarrollo del algoritmo

Para elaborar el algoritmo se tendrán en cuenta las siguientes reglas:

1. El día 1 de enero de 1.601 fue lunes.
2. Son años bisiestos los múltiplos de 4 excepto los que son múltiplos de 100 y que no son también múltiplos de 400. Así son bisiestos: 1.604, 1.992, 2.000 ó 2.400, pero no son bisiestos: 1.900, 2.100 ó 2.900.

Desde el punto de vista metodológico, en esta práctica también conviene utilizar la técnica de refinamientos sucesivos, para descomponer el programa en funciones y procedimientos. Concretamente, sugerimos realizar una función para determinar si un año es o no bisiesto y otra para calcular el día de la semana con el que comienza un mes de un año. También es aconsejable realizar un procedimiento para imprimir los días de la hoja del calendario con el formato presentado en el ejemplo. Por último, conviene utilizar tipos enumerados para manejar los días de la semana.

Con las sugerencias anteriores, debemos pensar el algoritmo completo de cada función/procedimiento y finalmente del programa principal.

8.3. Autocorrección de la práctica

Dado que todas las prácticas deben tener la misma cabecera, recomendamos que el alumno edite el código fuente de la tercera práctica a partir del fichero `practica_3.cpp`, generado automáticamente tras la corrección de la práctica 2.

Una vez editado el código de la práctica 3, comprobaremos su corrección seleccionando ***Prácticas → Comprobar la práctica 3*** en nuestro entorno de programación para C \pm .

La figura 8.3 resume esquemáticamente las comprobaciones que realiza el corrector automático.

8.3.1. Proceso de autocorrección

El proceso de autocorrección consta de los siguientes pasos:

1. ¿Fue exitosa la corrección de las prácticas 1 y 2?

Es condición indispensable para corregir la práctica 3 que las prácticas 1 y 2 hayan superado el proceso de corrección automática. En caso contrario se visualizará el mensaje de error de la figura 8.4.

2. ¿El programa 'Práctica 3' es válido según el lenguaje C \pm ?

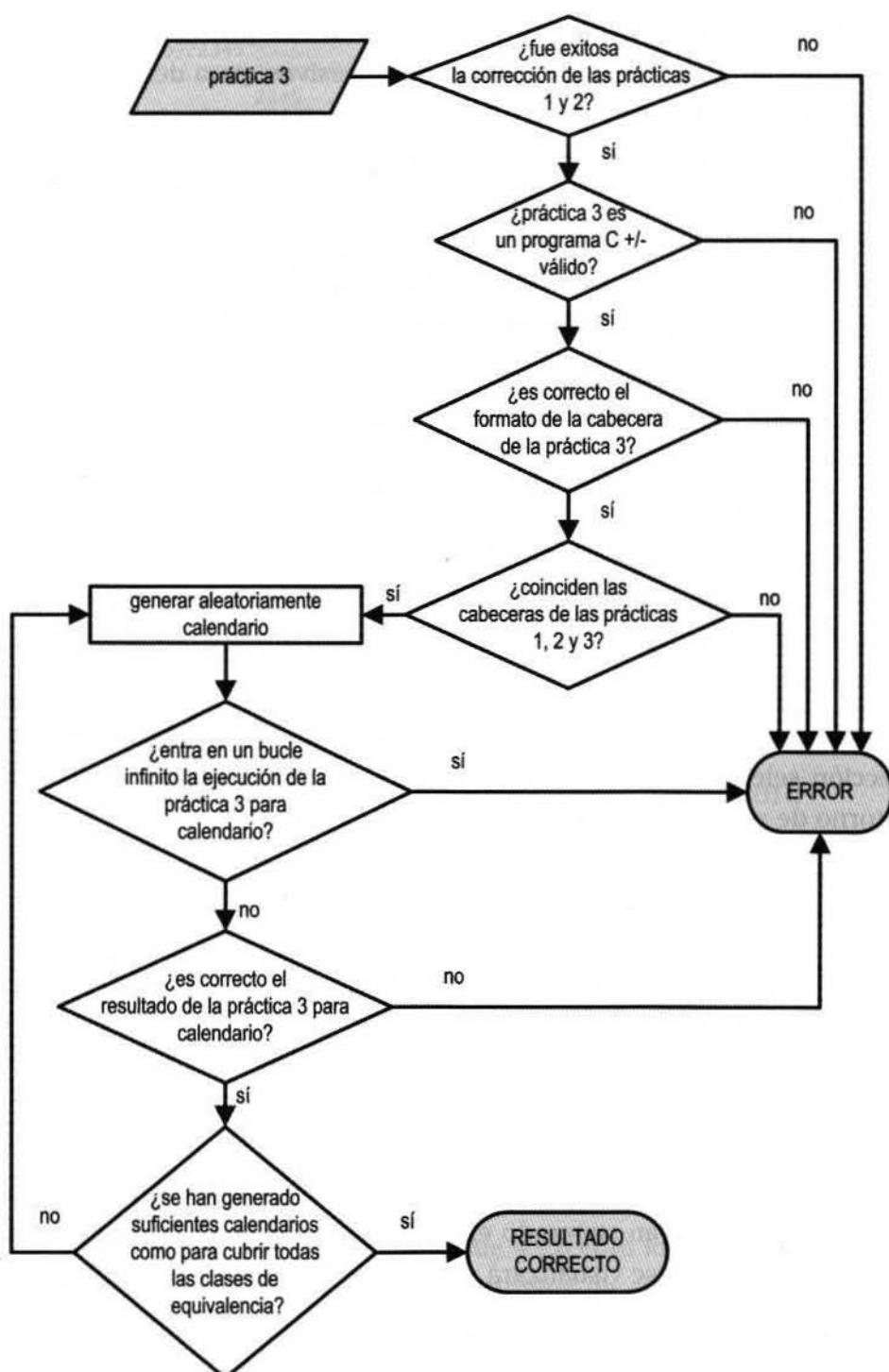


Figura 8.3: Corrección automática de la práctica 3

```
=====
<<ERROR>>: antes de corregir esta practica,
debe pasar la correccion de la practica 2
=====
```

Figura 8.4: Mensaje de error: se ha intentado corregir la práctica 3 sin que la práctica 2 pasara el proceso de corrección automática

Si el código fuente de la práctica 3 no satisface las restricciones sintácticas y semánticas de C \pm aparecerá un mensaje de error. Para solventarlo, conviene consultar los mensajes explicativos que aparecen en la parte inferior de la pantalla (pestañas *Log de PreCompilado C \pm* , *Log de generacion* y *Mensajes de generacion*).

3. ¿Es correcto el formato de la cabecera de la práctica 3?

Si la cabecera de la práctica es incorrecta se generará un mensaje de error. Por ejemplo, la figura 8.5 muestra el mensaje correspondiente a la cabecera errónea incluida en la figura 8.6.

```
=====
<<ERROR>>: Error en la cabecera del fichero
'C:\cmasmenos\practicas\practica_3.cpp',
revise el campo 'PRIMER APELLIDO'
=====
```

Figura 8.5: Mensaje de error provocado por la cabecera de la figura 8.6

```
*****
* NOMBRE: #Ismael#
* SEGUNDO APELLIDO: #Cardiel#
* DNI: #99999999#
* EMAIL: #iabad@issi.uned.es#
*****/
```

Figura 8.6: Ejemplo de cabecera incorrecta (falta el campo PRIMER APELLIDO)

4. ¿Coinciden las cabeceras de las prácticas 2 y 3?

Por ejemplo, si la figura 8.7 es la cabecera de la segunda práctica y la figura 8.8 la cabecera de la tercera práctica, se visualizará el mensaje de error de la figura 8.9 porque el correo electrónico de la segunda práctica (iabad@issi.uned.es) no coincide con el de la tercera (ISMAELABAD@issi.uned.es).

```
*****  
* NOMBRE: #Ismael#  
* PRIMER APELLIDO: #Abad#  
* SEGUNDO APELLIDO: #Cardiel#  
* DNI: #99999999#  
* EMAIL: #iabad@issi.uned.es#  
*****/
```

Figura 8.7: Ejemplo de cabecera válida para las prácticas 1, 2 y 3

```
*****  
* NOMBRE: #Ismael#  
* PRIMER APELLIDO: #Abad#  
* SEGUNDO APELLIDO: #Cardiel#  
* DNI: #99999999#  
* EMAIL: #ISMAELABAD@issi.uned.es#  
*****/
```

Figura 8.8: Cabecera de la práctica 3

```
=====  
=<>ERROR>: en la cabecera de esta práctica  
el campo 'EMAIL' es distinto del de las prácticas anteriores  
=====
```

Figura 8.9: Mensaje de error: los EMAILs de las cabeceras no coinciden
(ver figuras 8.7 y 8.8)

5. ¿Entra en un bucle infinito la ejecución de la práctica 3?

El corrector automático comprueba que la ejecución de la práctica no entra en un bucle infinito, es decir, no se queda colgada. Por otro lado, el programa debe pedir los datos de entrada (Mes y Año) una sola vez. Si los datos introducidos son erróneos, la ejecución del programa finalizará. Es decir, nunca debe haber un bucle indefinido que

solicite los datos de entrada repetidamente hasta que estos sean correctos.

6. ¿Es correcto el calendario para los casos de prueba?

El entorno verifica si se satisface el enunciado de la práctica. Para ello, se ejecuta el código para distintos valores de entrada (ver sección 8.3.2). Algunos valores de entrada se saldrán de los rangos de los meses (1-12) y los años (1600-1300). En estos casos, el programa no debe imprimir nada.

En caso de que el resultado de ejecutar el programa sea erróneo, se mostrará el correspondiente mensaje de error. Por ejemplo, si no se respeta el formato del calendario y, en lugar de usar el símbolo '=' para la cabecera de los días de la semana, se utiliza el símbolo '*', se producirá el mensaje de la figura 8.10.

8.3.2. Casos de prueba

Para verificar si el programa satisface el enunciado de la práctica, Code::Blocks^{C++} emplea un juego de casos de prueba destinado a someter al programa a un máximo número de situaciones diferentes. En general, probar completamente un programa es inabordable y además no resulta rentable ni práctico. En nuestro caso, el número de casos de prueba sería infinito:

$$\left\{ \begin{array}{l} (3000 - 1601 \text{ años}) \times 12 \text{ meses} \Rightarrow 16788 \text{ posibilidades válidas} \\ (-\infty, 1600] \text{ y } [3001, +\infty) \Rightarrow \infty \text{ posibilidades inválidas} \end{array} \right.$$

Como se muestra en la figura 8.11, con las pruebas sólo suele explorarse una parte de todas las posibilidades del programa. Se trata de alcanzar un compromiso para que con el menor esfuerzo posible se puedan detectar el máximo número de defectos y, sobre todo, aquellos que puedan provocar las consecuencias más graves. Para probar la tercera práctica se utiliza la técnica de partición en clases de equivalencia, dividiendo el espacio de ejecución del programa en varios subespacios. Cada subespacio o clase equivalente agrupa a todos aquellos datos de entrada al programa que producen resultados equivalentes.

=====

Calendario correcto (para 1-1601):

ENERO 1601					
LU	MA	MI	JU	VI	SA DO
1	2	3	4	5	6 7
8	9	10	11	12	13 14
15	16	17	18	19	20 21
22	23	24	25	26	27 28
29	30	31	.	.	.

=====

Calendario obtenido:

ENERO 1601					
LU	MA	MI	JU	VI	SA DO
1	2	3	4	5	6 7
8	9	10	11	12	13 14
15	16	17	18	19	20 21
22	23	24	25	26	27 28
29	30	31	.	.	.

=====

<< ERROR >>

=====

Figura 8.10: Formato de calendario erróneo

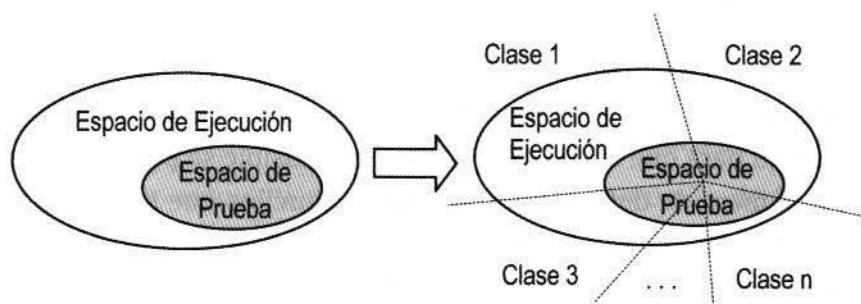


Figura 8.11: Partición del espacio de ejecución en clases de equivalencia

La corrección automática del entorno produce casos de prueba aleatorios dentro de las siguientes clases de equivalencia:

año	< 1600
	1601
	no bisiesto en [1602, 2099]
	bisiesto en [1602, 2099]
	3000
	> 3000
mes	< 1
	con 28 días
	con 30 días
	con 31 días
	> 12

Muchos programas se construyen codificando primero un tratamiento general y retocando luego el código para cubrir casos especiales. Por esta y otras razones es bastante normal que los errores tengan cierta tendencia a aparecer precisamente al operar en las fronteras o valores límite de los datos normales. Como puede observarse, los juegos de prueba que utiliza el entorno hacen especial hincapié en las zonas del espacio de ejecución que están en el borde de las clases de equivalencia (por ejemplo, los años 1.601 y 3.000).

8.4. Entrega telemática de las prácticas

Si el resultado de ejecutar `Practicas → Comprobar la practica 3` es correcto, el entorno se conectará automáticamente al servidor de la UNED y enviará encriptados los datos del alumno y de las prácticas al equipo docente.

Capítulo 9

Manejo de proyectos con el entorno

SEGÚN crece el tamaño de los programas, tener todo el código en un sólo fichero deja de ser práctico. Si dividimos nuestro programa en varios ficheros, las ventajas son múltiples: facilitamos que varios programadores puedan trabajar en paralelo sobre distintas partes del programa, es más fácil localizar y corregir los errores que podamos cometer, favorecemos la reutilización de cada fichero en otros programas...

Code::Blocks^{C±}, al igual que la mayoría de los entornos de desarrollo (Visual Studio, Eclipse...), utiliza el concepto de *proyecto* para organizar un programa modular en varios ficheros.

Utilizando un pequeño ejemplo, este capítulo describe cómo gestionar y procesar proyectos para obtener código ejecutable.

9.1. Un simulador del juego “Piedra, Papel o Tijera”

“Piedra, papel o tijera” es un juego infantil donde dos jugadores cuentan juntos 1... 2... 3... ¡Piedra, papel o tijera! y, justo al acabar, muestran al mismo tiempo una de sus manos, de modo que pueda verse el arma que cada uno ha elegido: Piedra (un puño cerrado), Papel (todos los dedos extendidos, con la palma de la mano mirando hacia abajo) o Tijera (dedos índice y corazón extendidos y separados formando una ‘V’). El objetivo del

juego es vencer al oponente, seleccionando el arma que gana a la que ha elegido él. La figura 9.1 resume la precedencia entre las armas:

- La piedra aplasta o rompe a la tijera (gana la piedra)
- La tijera corta al papel (gana la tijera)
- El papel envuelve a la piedra (gana el papel)
- Si los jugadores eligen la misma arma, hay empate

Vamos a realizar un pequeño programa que simule aleatoriamente 5 partidas del juego, produciendo una salida del estilo de la figura 9.2.

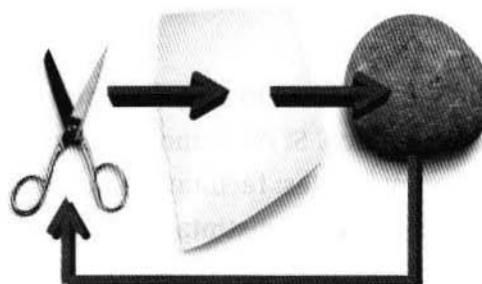


Figura 9.1: Tijera gana a Papel, Papel gana a Piedra, Piedra gana a Tijera

9.2. Solución del ejemplo

La figura 9.3 representa el diseño de nuestro simulador, que contiene un módulo principal (*Simulador*) y tres módulos auxiliares (*Jugada*, *Jugador* y *Arbitro*). Las flechas indican relaciones de dependencia entre módulos. Así, $X \rightarrow Y$ significa que el módulo X usa algún elemento del módulo Y . A nivel de código, la dependencia se refleja escribiendo en X la sentencia de importación `#include "Y.h"`.

Partida 1:

Jugador 1:

Tijera

Jugador 2:

Tijera

Resultado: Empate

Partida 2:

Jugador 1:

Papel

Jugador 2:

Papel

Resultado: Empate

Partida 3:

Jugador 1:

Tijera

Jugador 2:

Papel

Resultado: Gana Jugador 1

Partida 4:

Jugador 1:

Piedra

Jugador 2:

Piedra

Resultado: Empate

Partida 5:

Jugador 1:

Piedra

Jugador 2:

Papel

Resultado: Gana Jugador 2

Figura 9.2: Resultado de ejecutar el simulador PiedraPapelTijera

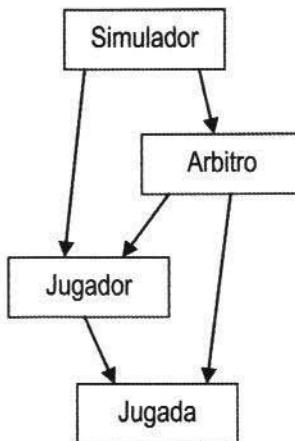


Figura 9.3: Diseño del programa

Todos los módulos están distribuidos en ficheros separados. Tal y como se indica en el tema 15 del libro básico de la asignatura [1], por cada módulo auxiliar hay dos ficheros¹:

1. Un fichero de interfaz `NombreModulo.h`, que comienza con la línea `#pragma once` y describe la especificación del módulo. Es decir, todo lo que se necesita saber para poder usar los elementos definidos en el módulo (*¿qué hace módulo?*).
2. Un fichero de implementación `NombreModulo.cpp`, que comienza con la línea `#include "NombreModulo.h"` y describe la realización del módulo (*¿cómo lo hace?*).

Gracias a esta separación `.h/.cpp` se consigue la *ocultación* de la realización de los módulos. Es decir, el programa que usa un elemento de un módulo sólo tiene visible la información de la interfaz, pero no la de la realización. Como se explica en el texto básico de la asignatura, gracias a la ocultación se consigue que los programas sean más fáciles de mantener.

¹Sin embargo, el módulo principal sólo consta de un fichero `.cpp` (esto no es ninguna novedad: hasta el presente capítulo, todo los programas que se han escrito estaban compuestos por un único módulo principal).

A continuación, se resume el código de los ficheros que componen el programa:

1. Módulo Jugada (figuras 9.4 y 9.5). Implementa con un tipo enumerado las armas que puede escoger un jugador (ver línea 3 de la figura 9.4). Incluye la función **GenerarJugada**, que devuelve un valor aleatorio del tipo enumerado.
2. Módulo Jugador (figuras 9.6 y 9.7). Simula el comportamiento de un jugador. Puede elegir un arma (mediante el procedimiento **Jugar**, que a su vez utiliza a **GenerarJugada**), devolver el arma elegida (**ConsultarJugada**) e imprimirla por pantalla (**ImprimirJugada**).
3. Módulo Arbitro (figuras 9.8 y 9.9). Decide, de entre dos jugadores, cual de ellos ha ganado la partida (o si hay empate).
4. Módulo Simulador (figura 9.10). Contiene la función principal (**main**), que controla la ejecución de los dos jugadores y el árbitro.

```
1 #pragma once
2
3 typedef enum TipoJugada { Piedra, Papel, Tijera } ;
4
5 TipoJugada GenerarJugada();
```

Figura 9.4: Jugada.h

```
1 #include "Jugada.h"
2 #include <stdlib.h>
3
4 TipoJugada GenerarJugada() {
5     return TipoJugada(rand() %3);
6 }
```

Figura 9.5: Jugada.cpp

```

1 #pragma once
2 #include "Jugada.h"
3
4 typedef struct Jugador {
5     void Jugar();
6     TipoJugada ConsultarJugada();
7     void ImprimirJugada();
8 private:
9     TipoJugada jugada;
10 };

```

Figura 9.6: Jugador.h

```

1 #include "Jugador.h"
2 #include <stdio.h>
3
4 void Jugador::Jugar() {
5     jugada = GenerarJugada();
6 }
7
8 TipoJugada Jugador::ConsultarJugada() {
9     return jugada;
10 }
11
12 void Jugador::ImprimirJugada() {
13     switch (jugada) {
14         case Piedra:
15             printf("\tPiedra\n");
16             break;
17         case Papel:
18             printf("\tPapel\n");
19             break;
20         default:
21             printf("\tTijera\n");
22     }
23 }

```

Figura 9.7: Jugador.cpp

```

1 #pragma once
2 #include "Jugador.h"
3
4 typedef struct Arbitro {
5     void DecidirResultadoPartida(Jugador j1, Jugador j2);
6 };

```

Figura 9.8: Arbitro.h

```

1 #include <stdio.h>
2 #include "Arbitro.h"
3 #include "Jugada.h"
4
5 void Arbitro::DecidirResultadoPartida(Jugador j1, Jugador j2) {
6     printf("Resultado: ");
7     if (j1.ConsultarJugada() == j2.ConsultarJugada()) {
8         printf("Empate\n\n");
9     } else if (j1.ConsultarJugada() == Piedra) {
10        if (j2.ConsultarJugada() == Tijera) {
11            printf("Gana Jugador 1\n\n");
12        } else {
13            printf("Gana Jugador 2\n\n");
14        }
15    } else if (j1.ConsultarJugada() == Papel) {
16        if (j2.ConsultarJugada() == Piedra) {
17            printf("Gana Jugador 1\n\n");
18        } else {
19            printf("Gana Jugador 2\n\n");
20        }
21    } else { /* j1.ConsultarJugada() == Tijera */
22        if (j2.ConsultarJugada() == Papel) {
23            printf("Gana Jugador 1\n\n");
24        } else {
25            printf("Gana Jugador 2\n\n");
26        }
27    }
28 }
```

Figura 9.9: Arbitro.cpp

```

1 #include <stdio.h>
2 #include "Jugador.h"
3 #include "Arbitro.h"
4
5 int main () {
6     Jugador j1;
7     Jugador j2;
8     Arbitro a;
9     for (int partida=0; partida<5; partida++) {
10        printf("-----\n");
11        printf("Partida %d:\n", partida+1);
12        printf("-----\n");
13        printf("Jugador 1:\n");
14        j1.Jugar();
15        j1.ImprimirJugada();
16        printf("Jugador 2:\n");
17        j2.Jugar();
18        j2.ImprimirJugada();
19        a.DecidirResultadoPartida(j1, j2);
20    }
21 }
```

Figura 9.10: Simulador.cpp

9.3. Escritura de la solución con Code::Blocks^{C±}

En vista de que la mayor parte de los programas del “mundo real” siguen un diseño modular y están compuestos por varios ficheros, prácticamente todos los entornos de programación modernos ofrecen algún mecanismo para facilitar la organización de un programa en distintos ficheros. El mecanismo que ofrece Code::Blocks^{C±} se denomina *proyecto*.

Si organizamos nuestro programa PiedraPapelTijera con un proyecto, tendremos disponible una representación gráfica de los ficheros que lo componen como la figura 9.11, que indica que el programa está compuesto por 4 ficheros de realización Arbitro.cpp, Jugada.cpp, Jugador.cpp, Simulador.cpp (*Fuentes*) y 3 de interfaz Arbitro.h, Jugada.h, Jugador.h (*Cabeceras*). Además, esta representación facilitará la edición del código de cualquier fichero (bastará con hacer doble clic sobre el nombre del fichero, y su código aparecerá sobre el área de edición).

Al utilizar proyectos, Code::Blocks^{C±} tratará a todos los ficheros de un programa como una unidad. Así, la apertura y cierre de un proyecto abrirá y cerrará todos los ficheros del programa (no tendremos que ir abriendo y cerrando cada uno de los ficheros). Mas importante aún, para generar código ejecutable, en lugar de tener que ir compilando cada fichero de uno en uno, Code::Blocks^{C±} compilará y montará todos los ficheros con una sola orden: ¡ bastará con que pulsemos **F6** !

Para crear un proyecto, seleccionaremos la opción **Archivo → Nuevo → Proyecto...**. Aparecerá la figura 9.12, donde elegiremos **Proyecto vacío** y pulsaremos el botón **Elegir**.

Escribiremos el nombre del proyecto y la carpeta donde deseamos almacenarlo (ver figura 9.13). Bastará con que rellenemos los campos *Título del proyecto* y *Directorio para crear el proyecto*, ya que los otros dos campos se derivan automáticamente de ellos. Pulsaremos **Next**.

Aparecerá la figura 9.14, donde pulsaremos **Finish**.

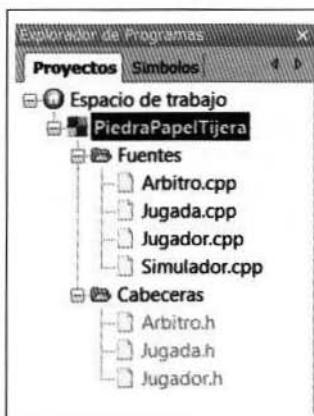


Figura 9.11: Representación gráfica de los ficheros que componen el programa PiedraPapelTijera

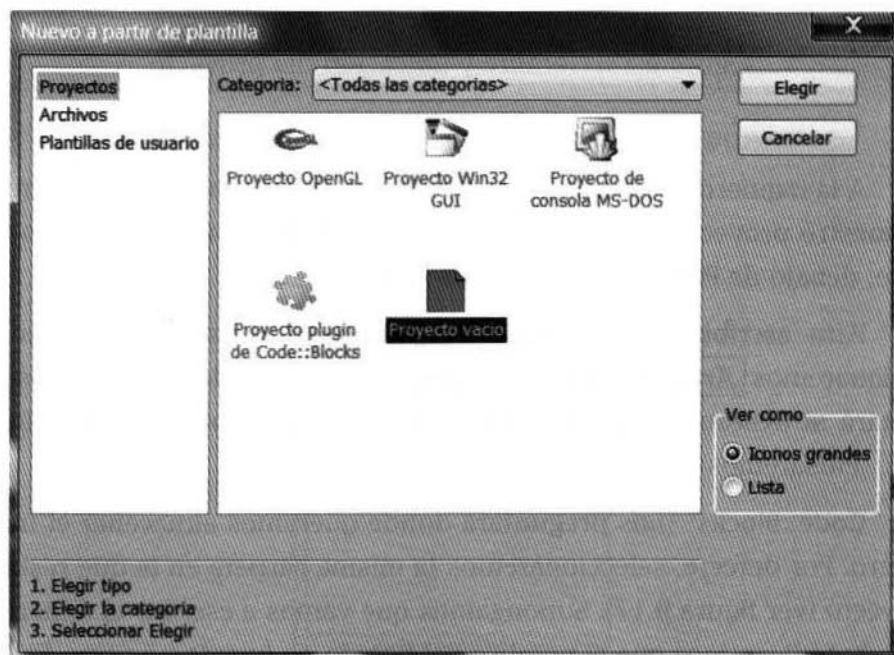


Figura 9.12: Creación de un proyecto vacío

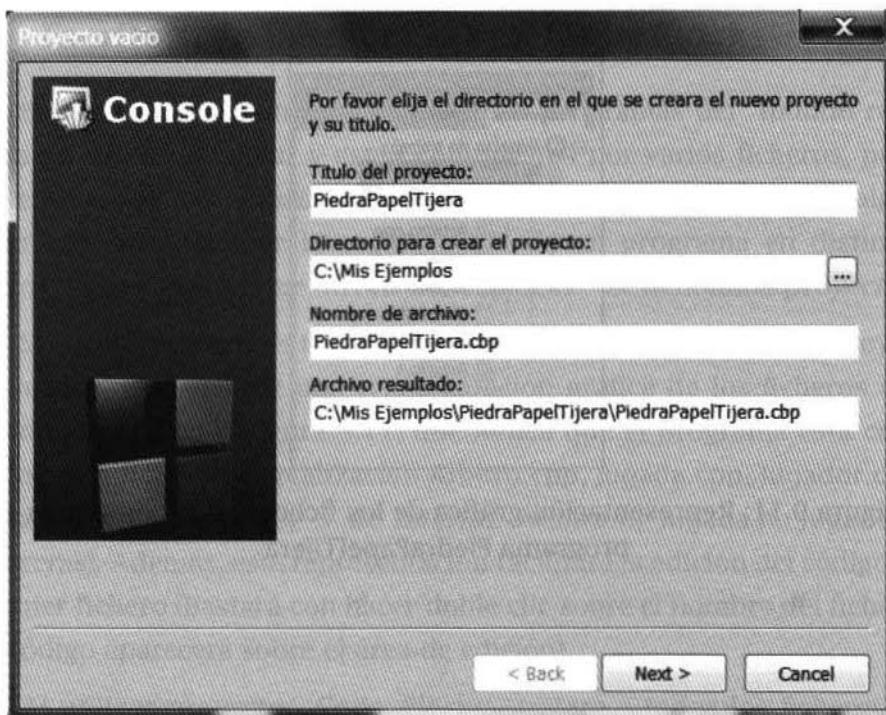


Figura 9.13: Nombre y ubicación del proyecto

A la izquierda de $\text{Code}::\text{Blocks}^{C\pm}$, aparecerá la representación gráfica de nuestro proyecto (ver figura 9.15). Como de momento el proyecto está vacío, debajo de PiedraPapelTijera “no cuelga” nada.

Para escribir los ficheros que componen nuestro programa, seleccionaremos **Archivo → Nuevo → Fichero vacío**. $\text{Code}::\text{Blocks}^{C\pm}$ nos preguntará si deseamos incluir el nuevo fichero en el proyecto (ver figura 9.16). Pulsaremos **Sí**.

$\text{Code}::\text{Blocks}^{C\pm}$ nos preguntará dónde queremos almacenar el nuevo fichero. Por defecto, seleccionaremos la misma carpeta en la que reside el proyecto (ver figura 9.17). Supongamos que vamos a escribir el fichero *Jugada.h*. Indicaremos que el nombre del fichero es *Jugada* y su tipo *Fichero de cabecera Code::Blocks*.

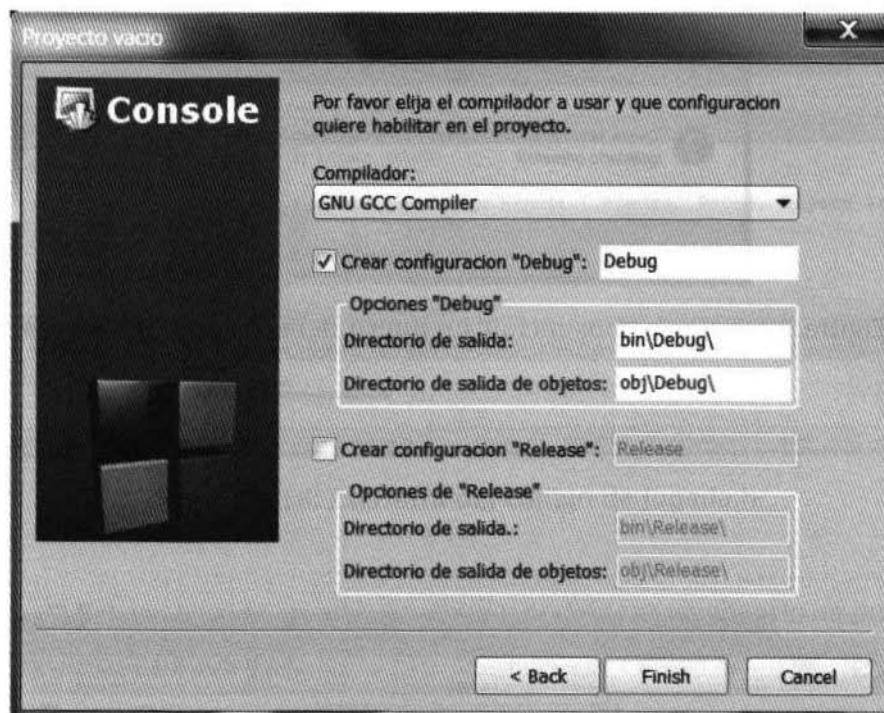


Figura 9.14: Última pantalla de creación de un proyecto

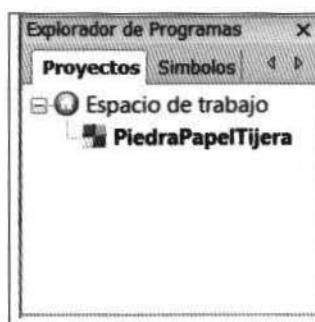


Figura 9.15: Representación gráfica del proyecto vacío

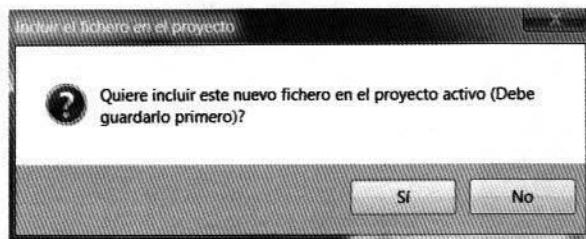


Figura 9.16: ¿Deseamos incluir un nuevo fichero en un proyecto?

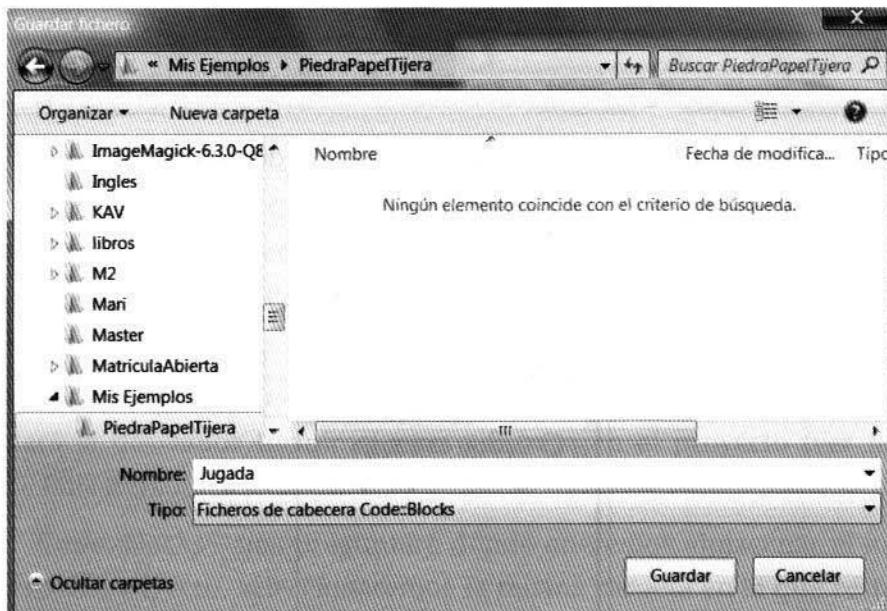


Figura 9.17: Localización del nuevo fichero Jugada.h

En el área de edición, escribiremos el código del fichero (ver figura 9.18).

The screenshot shows the Code::Blocks IDE interface. The title bar reads "C:\Mis Ejemplos\PiedraPapelTijera\CodeBlocks-B17". The menu bar includes Archivo, Edición, Ver, Buscar, Proyecto, Generar, Depurar, Herramientas, Prácticas, Complementos, Configuración, and Ayuda. The "Explorador de Programas" window is open, showing a project named "PiedraPapelTijera" with a file "Jugada.h" selected. The code editor window displays the following C++ code:

```

1 #pragma once
2
3 typedef enum TipoJugada { Piedra, Papel, Tijera };
4
5 TipoJugada GenerarJugada();

```

The status bar at the bottom shows the path "C:\Mis Ejemplos\PiedraPapelTijera\Jugada.h", line 5, column 28, and the message "Insertar Modificado Lectura/Escritura ccuned".

Figura 9.18: Escritura del código de Jugada.h

El fichero ha sido creado con éxito y podemos visualizarlo en nuestro proyecto (ver figura 9.19).

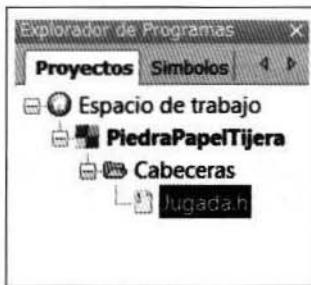


Figura 9.19: Representación de Jugada.h en el proyecto

Tras escribir el fichero de interfaz Jugada.h, escribiremos su correspondiente fichero de realización Jugada.cpp. Para ello, volveremos a seleccionar Archivo → Nuevo → Fichero vacío. Cuando Code::Blocks^{C±} nos pregunte por el nombre del fichero, indicaremos que su tipo es *Ficheros de C+/- Code::Blocks* (ver figura 9.20). Tras editar el código del fichero (ver figura 9.21), podremos visualizar su etiqueta en el proyecto (ver figura 9.22).

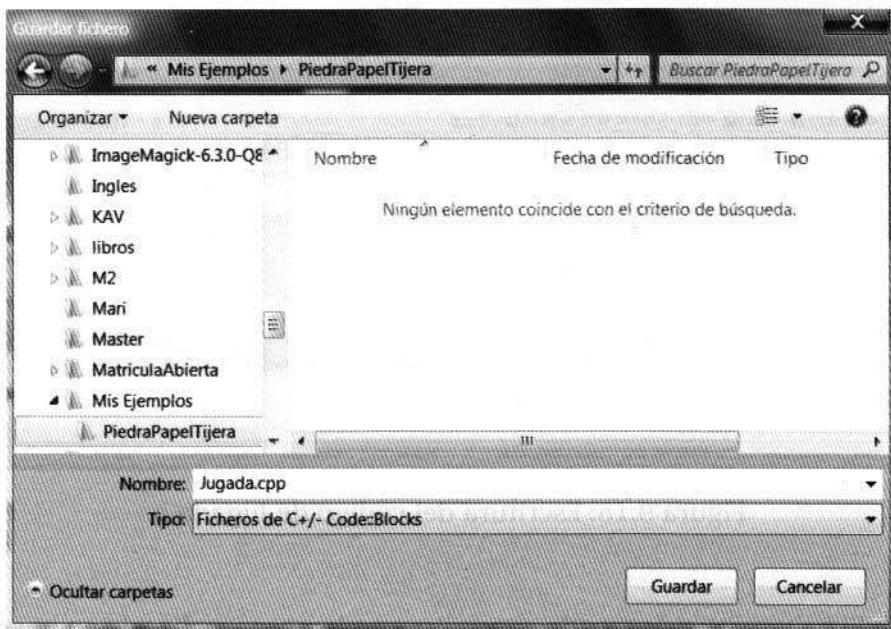


Figura 9.20: Localización del nuevo fichero Jugada.cpp

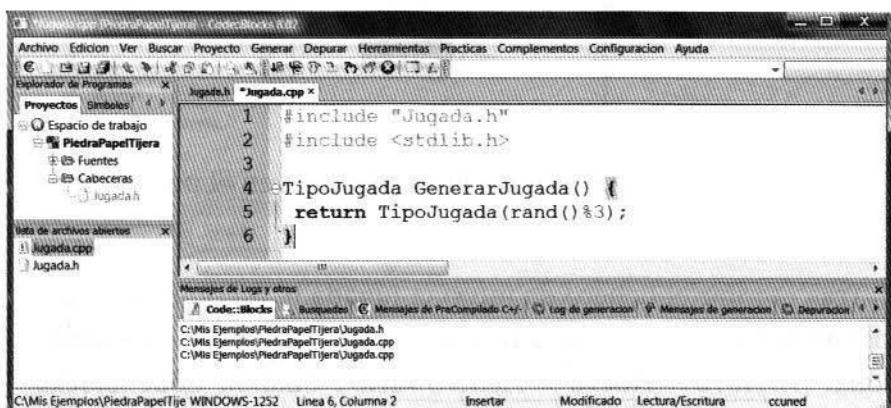


Figura 9.21: Escritura del código de Jugada.h

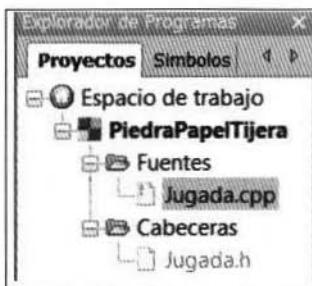


Figura 9.22: Representación de Jugada.cpp en el proyecto

Para incorporar al proyecto el código del resto de los ficheros (Arbitro.h, Arbitro.cpp, Jugador.h, Jugador.cpp, Simulador.cpp) repetiremos los pasos anteriores, obteniendo la representación de la figura 9.11.

Finalmente, generaremos y ejecutaremos el programa pulsando **F6**.

9.4. Organización de un proyecto en MS Windows

A efectos prácticos, un proyecto en Code::Blocks^{C±}no es más que una estructura de carpetas y un fichero de configuración. Si abrimos la carpeta donde guardamos el proyecto (C:\Mis Ejemplos\PiedraPapelTijera según la figura 9.13) con el explorador de Windows, veremos que contiene:

1. Los ficheros .cpp y .h de nuestro programa. Esto se debe a que decidimos guardarlos en la carpeta del proyecto (ver figura 9.17).
2. Un fichero de configuración .cbp (en nuestro caso, PiedraPapelTijera.cbp). Este fichero contiene toda la información del proyecto y nunca debe editarse.
3. Una carpeta obj\Debug que almacena el código compilado de cada fichero del proyecto.
4. Una carpeta bin\Debug que almacena el código ejecutable del programa (en nuestro caso, PiedraPapelTijera.exe).

Consecuentemente:

1. Si deseamos copiar, mover o eliminar un proyecto, basta con copiar, mover o eliminar la carpeta que lo contiene con el explorador de Windows (en nuestro caso, C:\Mis Ejemplos\PiedraPapelTijera).
2. Si deseamos distribuir una versión ejecutable de nuestro programa, basta con copiar el archivo .exe de la carpeta bin\Debug.

9.5. Menús para la gestión de proyectos

Code::Blocks^{C±} ofrece diversas operaciones sobre proyectos mediante el menú **Proyecto** (ver figura 9.23).

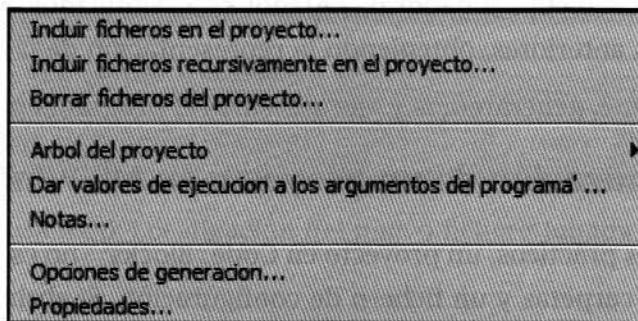


Figura 9.23: Menú **Proyecto**

Además, estas opciones y otras relacionadas con la generación de proyectos también están disponibles en el menú contextual que aparece al pulsar con el botón derecho del ratón el nombre de un proyecto en el área del *Explorador de Programas* (ver figura 9.24).

A menudo, es conveniente reutilizar un fichero entre varios proyectos. Para añadir ficheros ya existentes a un proyecto, utilizaremos la opción del menú **Proyecto→Incluir ficheros en el proyecto**, que abre un diálogo de selección del archivo que queremos incluir. Una vez que el fichero seleccionado ha quedado incluido, se mostrará en el árbol del proyecto del área del *Explorador de Programas*. El archivo incluido se visualizará con la apariencia que esté configurada (podemos

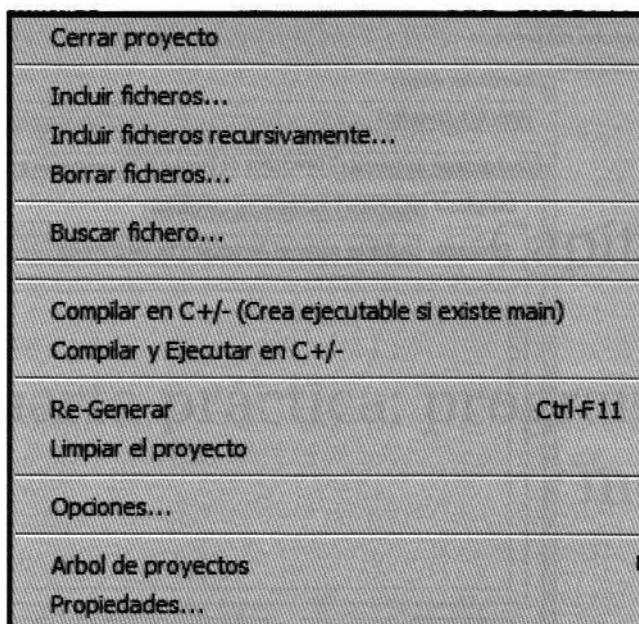


Figura 9.24: Menú contextual para el manejo de un proyecto

seleccionar diferentes opciones de apariencia del árbol en el menú *Proyecto→Arbol del proyecto*).

La opción de inclusión tiene un efecto meramente relacionado con la gestión de archivos en el proyecto, porque aunque se haya incluido como fichero en el proyecto no quedará incluido desde el punto de vista de la generación. Por ejemplo, si incluimos un fichero de cabecera que contiene el código de algunos subprogramas, sólo podremos hacer uso de estas operaciones en un programa fuente del proyecto si añadimos en las opciones de generación del proyecto una indicación sobre donde se encuentra este fichero de cabecera. Para resolver esto, debemos acceder al menú *Proyecto→Opciones de generación*, figura 9.25, y acceder a la pestaña de directorios de búsqueda. En este diálogo, se deberá incluir el directorio en donde se encuentre almacenado el fichero de cabecera que queremos (cuidado con usar espacios en blanco en los nombres de los directorios porque pueden afectar a la búsqueda. Una posible solución es utilizar direccionamiento relativo al directorio del proyecto).

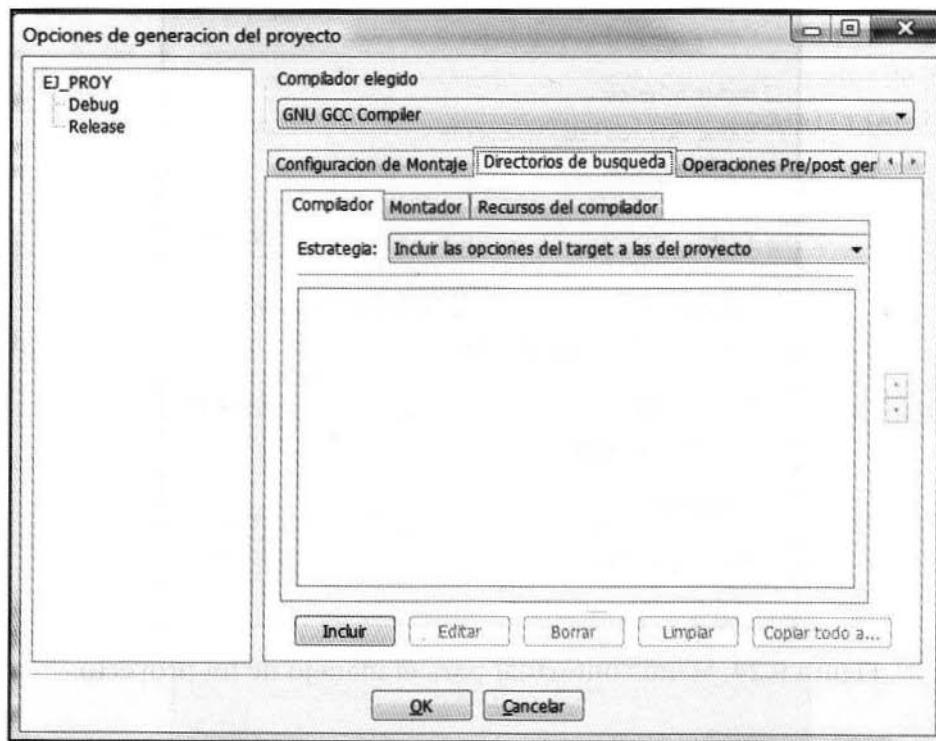


Figura 9.25: Opciones de generación del proyecto

Capítulo 10

Cuarta práctica: programación modular

La cuarta y última práctica del curso tiene como objetivo que el alumno ejercite la descomposición modular de programas como herramienta para resolver problemas complejos. Frente al resto de las prácticas, esta última tiene las siguientes peculiaridades:

1. Es diferente cada curso académico. Los enunciados se publicarán en la web de la asignatura [5]. A modo de ejemplo, este capítulo incluye un enunciado de este tipo de práctica.
2. **No se corrige automáticamente**, sino que **el alumno deberá entregarla al tutor de su centro asociado**.

10.1. Normativa

Las tres primeras prácticas presentadas en este libro se mantienen en todos los cursos académicos de la asignatura *Fundamentos de Programación*. Sin embargo, la cuarta práctica será distinta cada año.

La cuarta práctica se deberá entregar personalmente al tutor presencial del centro asociado donde se matriculó. La recogida de la práctica dependerá completamente de cada tutor y de las condiciones particulares

de su centro. Le aconsejamos que se ponga en contacto con su tutor a principios de curso.

Su tutor será quien evalúe la práctica. Por tanto, si desea una revisión de su calificación deberá dirigirse a él.

El enunciado de la práctica y un listado con los tutores por centro asociado se publicarán en la web de la asignatura [5].

10.2. Ejemplo de enunciado de la cuarta práctica

Con el objetivo de que el alumno se haga una idea del tipo enunciado que puede tener la cuarta práctica, se incluye el siguiente enunciado de ejemplo. Por lo tanto, ésta no es la práctica que debe realizarse. La práctica que debe resolverse cada curso académico será la publicada en la web de la asignatura [5].

ENUNCIADO DE EJEMPLO:

Realizar un programa en C± para la gestión de los eventos de un pabellón polideportivo. El núcleo del programa será un tipo abstracto de datos, llamado *Pabellón*, que gestionará la información de hasta 50 eventos deportivos. La información de cada evento incluirá los siguientes campos:

1. Código del evento: se trata de un número entero que identifica el evento.
2. Equipo 1: nombre del primer equipo que participa en el evento.
3. Equipo 2: nombre del segundo equipo que participa en el evento.
4. Fecha: día, mes y año de celebración del evento.
5. Hora: momento en que se celebra el evento.
6. Tipo de evento: puede ser partido de baloncesto, voleibol, balonmano ó fútbol sala.

La figura 10.1 muestra el menú principal del programa, que incluye las siguientes opciones que implementará el tipo abstracto *Pabellón*:

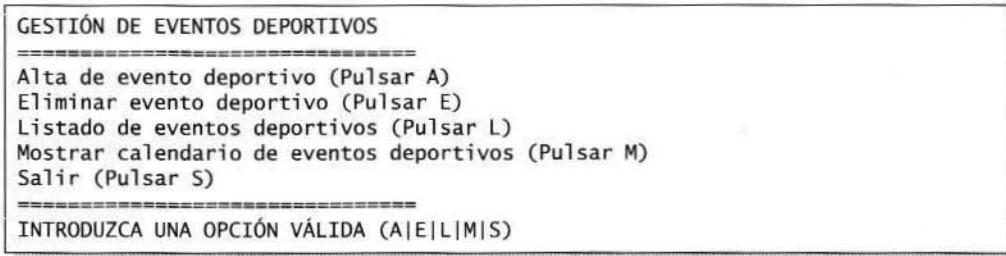


Figura 10.1: Menú principal del programa

1. Alta de evento deportivo: la figura 10.2 muestra un ejemplo de pantalla de alta de evento. Antes de dar de alta un evento deportivo, deberán realizarse las siguientes comprobaciones:
 - a) El código del nuevo evento es distinto del código de los eventos que ya han sido dados de alta. Si no es así, se informará de que ya existe un evento con ese mismo código y no se introducirá el nuevo evento.
 - b) No existe ningún evento deportivo en la fecha indicada. Es decir, en cada fecha se podrá celebrar como máximo un evento. En caso contrario, se informará de que ya existe un evento en esa fecha y no se introducirá el nuevo evento.

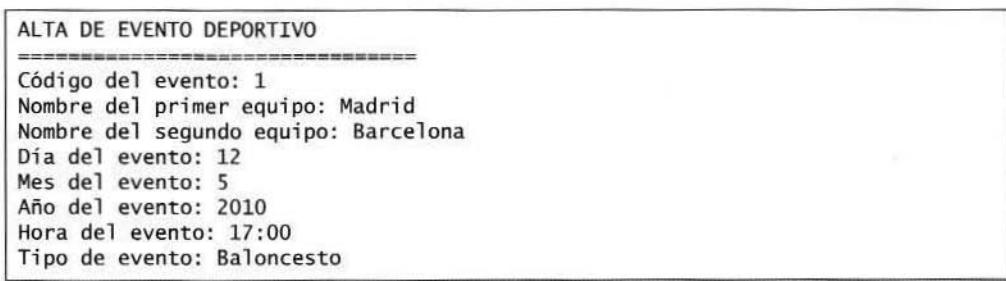


Figura 10.2: Alta de evento

2. Eliminar evento deportivo: la figura 10.3 muestra un ejemplo de pantalla de eliminación de evento. Antes de eliminar un evento, deberá comprobarse que su código ha sido dado de alta en *Pabellón*. En ca-

so contrario, se informará de que no existe un evento con el código introducido y, por tanto, no se eliminará ningún evento deportivo.

Cuando un evento se ha eliminado, su posición dentro de los 50 eventos que puede manejar el programa quedará vacía y podrá ser ocupada por un nuevo evento.

```
ELIMINAR EVENTO DEPORTIVO
=====
Código del evento a eliminar: 1
```

Figura 10.3: Eliminación de evento

3. Listado de eventos deportivos: la figura 10.4 muestra un ejemplo de pantalla donde se ha introducido el tipo de evento del que quiere obtenerse un listado (Baloncesto).

```
LISTADO DE EVENTOS DEPORTIVOS
=====
Tipo de evento deportivo: Baloncesto
```

Figura 10.4: Solicitud de listado de eventos

La figura 10.5 muestra el listado producido. En caso de que no existan eventos del tipo solicitado se mostrará un mensaje de error explicativo.

```
LISTADO DE EVENTOS DEPORTIVOS
=====
Baloncesto:
Madrid vs Barcelona. 12-05-2010. 17:00 horas
Madrid vs Unicaja. 15-05-2010. 18:00 horas
```

Figura 10.5: Listado producido ante la solicitud de la figura 10.4

4. Mostrar calendario de eventos deportivos: la figura 10.6 muestra un ejemplo de pantalla donde se han introducido el mes y el año del calendario que se desea consultar. La figura 10.7 muestra el calendario producido. Para implementar esta funcionalidad, **se aconseja**

reutilizar el código de la tercera práctica con las siguientes adaptaciones:

- a) Dentro del calendario, se indicará con los caracteres XX qué días tienen un evento programado.
- b) Debajo del calendario se incluirá un mensaje que indique el tipo de evento deportivo, los equipos que participan y la hora de comienzo del mismo.

```
MOSTRAR CALENDARIO DE EVENTOS
=====
Mes: diciembre
Año: 2009
```

Figura 10.6: Calendario de eventos

DICIEMBRE 2009						
LU	MA	MI	JU	VI		SA DO
.	1	2	3	4		5 6
7	8	9	10	11		12 13
14	15	XX	17	18		19 20
21	XX	23	24	25		26 27
28	29	30	31	.		.
30

16-11-2009 Madrid vs Estudiantes. Baloncesto. 19:00 horas
22-11-2009 Ciudad Real vs Barcelona. Balonmano. 20:00 horas

Figura 10.7: Calendario producido ante la solicitud de la figura 10.6

Capítulo 11

Esto es sólo el principio

ESTE manual trata de cubrir la prestaciones mínimas de Code::Blocks^{C±} para realizar programas. Sin embargo, los entornos de programación modernos suelen incluir muchas más posibilidades. La descripción exhaustiva de todas las prestaciones de Code::Blocks^{C±} requeriría un nuevo libro y se sale de los objetivos de este manual. Sin embargo, creemos que es interesante que el lector indague por su cuenta y experimente con las opciones disponibles. Este capítulo presenta algunas prestaciones adicionales de Code::Blocks^{C±} y resume los menús donde el lector puede iniciar sus propias investigaciones.

11.1. Complementos de Code::Blocks^{C±}

Code::Blocks^{C±} incluye la posibilidad de usar complementos (*plugins* en inglés) que introducen funcionalidades adicionales en el entorno. La gestión de los complementos instalados en Code::Blocks^{C±} se realiza a través de la opción del menú **Complementos**. Por defecto, este menú contiene únicamente dos opciones (ver figura 11.1):

- **Formateador de código (AStyle)**: formatea automáticamente el texto del fichero activo en el área de edición (por defecto, se aplicará el formato para C± descrito en el texto base de la asignatura [1]).

- **Gestionar complementos...**: inicia un diálogo que nos muestra los diferentes complementos cargados en el entorno y su estado (ver figura 11.2). En la siguiente sección, se describe uno de estos complementos: el plugin de búsqueda *ThreadSearch*.

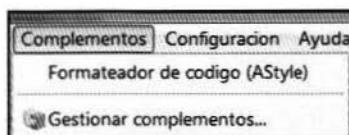


Figura 11.1: Opciones del menú **Complementos**

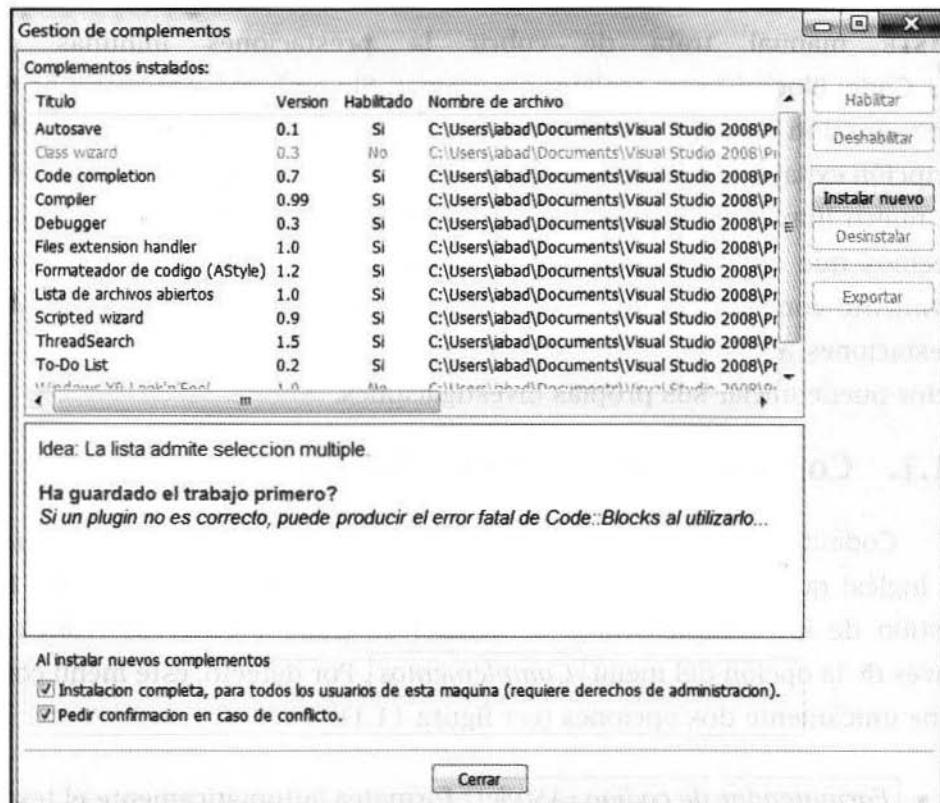


Figura 11.2: Diálogo de gestión de complementos

11.2. Complemento de búsqueda *ThreadSearch*

Code::Blocks^{C++} dispone de dos opciones para localizar texto en nuestros ficheros:

1. Mecanismo de búsqueda nativo: se encuentra en el menú **Buscar** (ver figura 11.3) y genera los resultados de la búsqueda en el panel de mensajes que tiene el título *Busquedas*. Este mecanismo es adecuado para realizar búsquedas rápidas en el fichero activo en el área de edición. Por esta razón, además de opciones de búsqueda, el menú ofrece acciones de reemplazo de texto, así como acciones para navegar por los ítems detectados en una búsqueda sobre un fichero único.

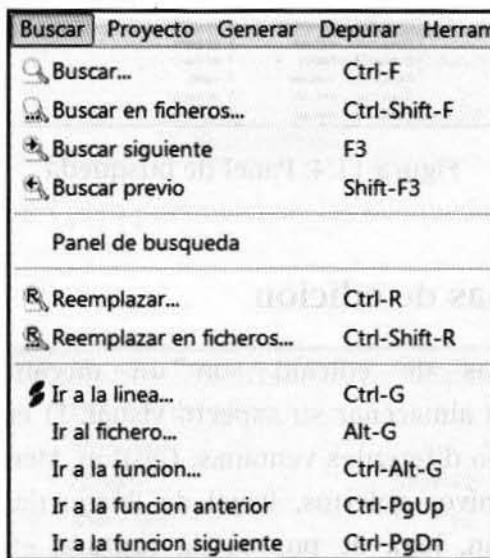


Figura 11.3: Opciones del menú **Buscar**

2. El complemento *ThreadSearch*: facilita la localización de texto en varios ficheros y está accesible en la pestaña del área de mensajes que tiene como título *Panel de búsqueda* (ver figura 11.4). Al utilizar el complemento, deberemos seleccionar el ámbito de aplicación de la búsqueda según las siguientes opciones:

- Todos los ficheros abiertos.
- Todos los ficheros del proyecto activo.
- Los ficheros del entorno de trabajo (todos los ficheros de los proyectos en el área).
- Los ficheros de un directorio.

Estas cuatro opciones pueden combinarse para dar lugar a búsquedas de contenidos más complejas. Además, el funcionamiento de la búsqueda puede parametrizarse a través de la figura 11.5.

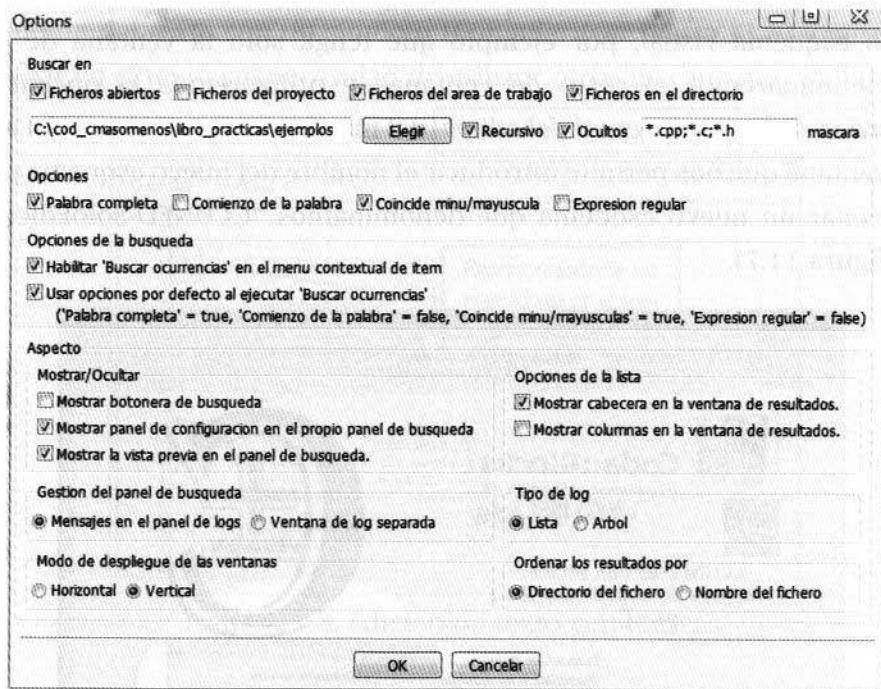
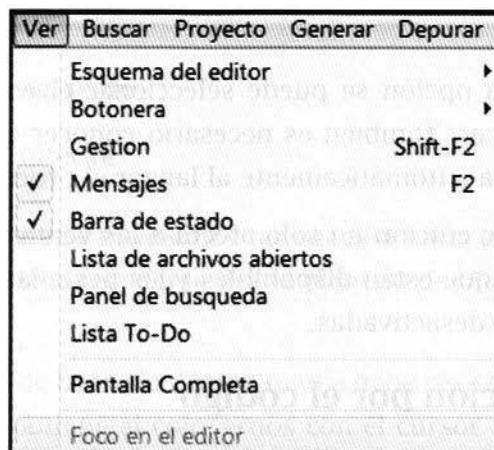


Figura 11.4: Panel de búsqueda

11.3. Esquemas de edición

Los esquemas de edición son un mecanismo que ofrece Code::Blocks^{C±} para almacenar su aspecto visual. El entorno pone a disposición del usuario diferentes ventanas: Gestión, Mensajes, Barra de Estado, Lista de archivos abiertos, Panel de Búsqueda, Lista-To-Do, ventanas de inspección, lista de puntos de ruptura, etc.. Estas ventanas pueden activarse o desactivarse desde el menú **Ver**, o desde el menú **Depurar→Ventana de depuración** con las diferentes opciones de cada caso (ver figura 11.6).

Cada combinación de estas ventanas disponibles y su apariencia visual (tamaño, posición, estado...) es lo que da lugar a un esquema de edición. Por defecto, el entorno proporciona dos esquemas: el esquema por defecto *Code::Blocks CCUNED* y el esquema de depuración *Debugging*. Las operaciones para crear nuevos esquemas o borrar esque-

Figura 11.5: Opciones para parametrizar las búsquedas en *ThreadSearch*Figura 11.6: Operaciones del menú **Ver**

mas se encuentran en el menú **Ver→Esquema del editor**. Para crear un nuevo esquema visual, por ejemplo que tenga sólo la ventana de edición, eliminaremos el resto de ventanas y utilizaremos la opción de **Guardar el esquema actual del editor** del menú anterior. Esta opción abre una ventana que nos permite introducir el nombre del nuevo esquema para almacenar un nuevo esquema que denominamos “CCUNED-SoloEdicion” (ver figura 11.7).



Figura 11.7: Operación para guardar el esquema: CCUNED-SoloEdicion

En esa misma opción se puede seleccionar el esquema de edición que queremos aplicar. También es necesario conocer que el esquema de depuración se activa automáticamente al lanzar el proceso de depuración.

El esquema de edición no sólo afecta a las ventanas, también almacena las botoneras que están disponibles y las pestañas de log que se encuentran activadas/desactivadas.

11.4. Navegación por el código

El entorno incluye una serie de opciones que facilitan al programador las tareas de edición y navegación por el código. Estas opciones se denominan *marcas* y consisten en señales que nos permiten almacenar referen-

cias a líneas de código. Gráficamente, estas marcas se colocan en la parte izquierda del texto con un símbolo triangular (ver figura 11.8) y permiten navegar en el texto sin necesidad de utilizar otros mecanismos de búsqueda.

The screenshot shows a code editor window titled "UNED 2009 - test4.cpp x". The code is written in C++ and defines three methods: `void TipoPapel :: MarcarH()`, `void TipoPapel :: MarcarV()`, and `void TipoPapel :: Imprimir()`. The code uses nested loops to fill two 2D arrays, `marcasH` and `marcasV`, with specific characters based on coordinates `x` and `y`. The arrays are then printed to the console. Several small black square markers with white triangular arrows pointing to the right are placed along the left margin of the code lines, starting from line 30 and continuing down to line 48. A callout box labeled "Representación de los marcadores en el área de edición" points to one of these markers.

```

28
29
30 void TipoPapel :: MarcarH()
31     if (x >= 0 && x < ANCHO)
32         marcasH[x][y] = '^';
33     }
34 }
35
36 void TipoPapel :: MarcarV()
37     if (y >= 0 && y < ALTO)
38         marcasV[x][y] = '|';
39     }
40
41 void TipoPapel :: Imprimir ()
42     for (int y=ALTO-1; y>=0; y--) {
43         for (int x=0; x<ANCHO; x++) {
44             printf( "%c%c", marcasV[x][y], marcasH[x][y]);
45         }
46         printf( "\n" );
47     }
48 }
```

Figura 11.8: Edición de texto con marcas

Las operaciones que podemos hacer con marcas son:

- Incluir una marca
- Borrar una marca
- Ir a la marca siguiente
- Ir a la marca anterior

Las operaciones de inclusión y navegación hacia atrás o hacia delante con las marcas pueden accederse desde el menú de ***Edición→Marcadores*** (ver figura 11.9).

La operación de borrado de una marca debe realizarse desde el menú contextual que se obtiene al colocarnos con el cursor sobre la señal de la propia marca (ver figura 11.10).

A la hora del uso de las marcas es importante tener en cuenta que las marcas no se almacenan. Sólo están disponibles mientras se mantienen

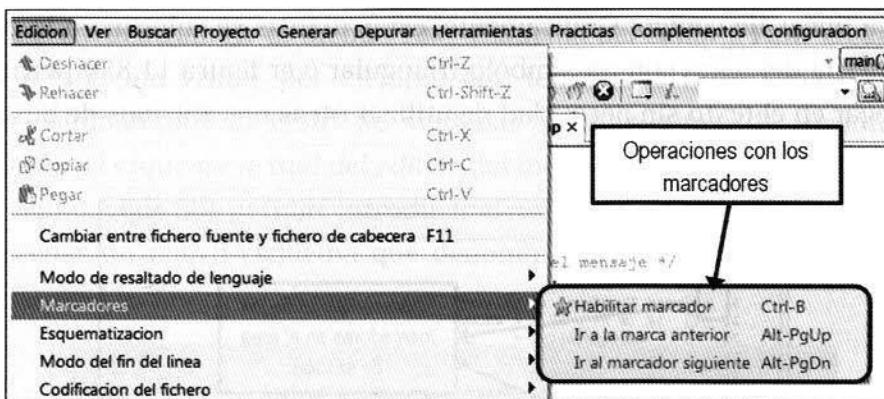


Figura 11.9: Menú de marcadores

```

1  #include <stdio.h>
2
3
4
5  int main() {
6      /* Sentencia de salida del mensaje */
7      printf("Hola mundo!!\n");
8      printf("Hola mundo!!\n");
9      printf("Hola mundo!!\n");
10     if ('a' > 'a')
11         Incluir punto de interrupcion
12
13
14
15 }

```

Figura 11.10: Borrado de una marca

marcados puntos de ficheros abiertos. Al cerrar o reabrir un fichero las marcas existentes se perderán.

11.5. Configuración avanzada del entorno

Las opciones de configuración de Code::Blocks^{C++} se encuentran en el menú de **Configuración** (ver figura 11.11). Estas opciones se organizan del siguiente modo:

1. Configuración general del entorno:

- a) Configuración general (ver figura 11.12).
- b) Opciones de vista.
- c) Aspectos de la edición.
- d) Sistema de paneles.
- e) Gestión de mensajes de confirmación.
- f) Configuración de red.
- g) Seleccionar programas para extensión. Configuración MIME.
- h) Configuración del complemento de búsqueda.
- i) Configuración del complemento de la lista to-do.
- j) Configuración del complemento de autoguardado.

2. Configuración de la edición de texto:

- a) Configuraciones básicas de la edición (ver figura 11.13).
- b) Opciones de esquematización (contracción/expansión) de bloques.
- c) Márgenes y visualización del cursor activo.
- d) Resaltado de sintaxis según el lenguaje de programación.
- e) Configuración del complemento de Auto-completar.
- f) Mensaje por defecto.
- g) Configuración del complemento de proponer código y manejar símbolos.
- h) Configuración del complemento sobre estilo de programación.

3. Configuración de la generación y depuración de programas: ejecutables y de la depuración

- a) Configuraciones globales. Aplicable a la hora de realizar compilaciones de ficheros, fuera de proyectos (ver figura 11.14).
- b) Generaciones batch.
- c) Configuración de inicio del depurador

4. Configuración de las variables globales del entorno.

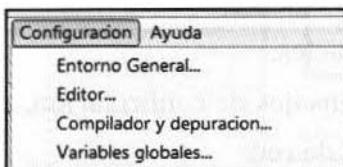


Figura 11.11: Menú Configuración

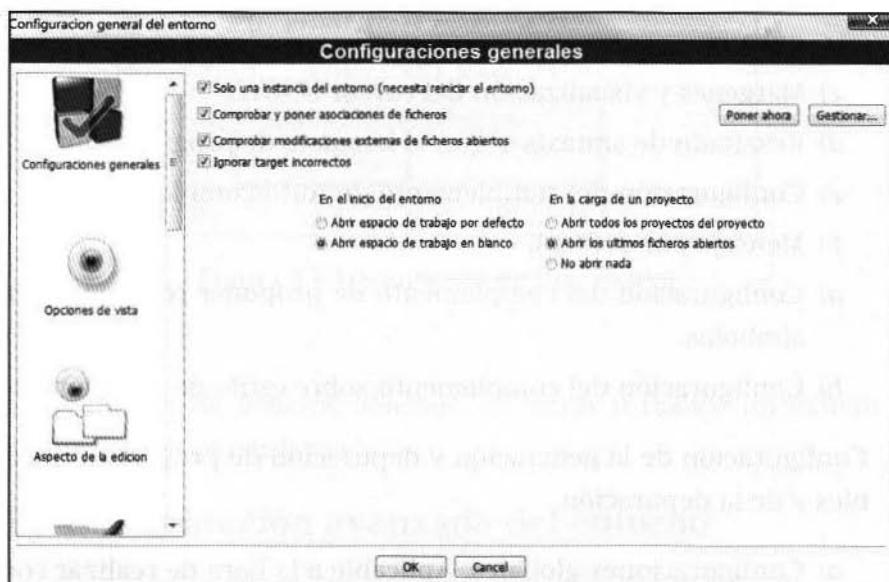


Figura 11.12: Configuraciones generales

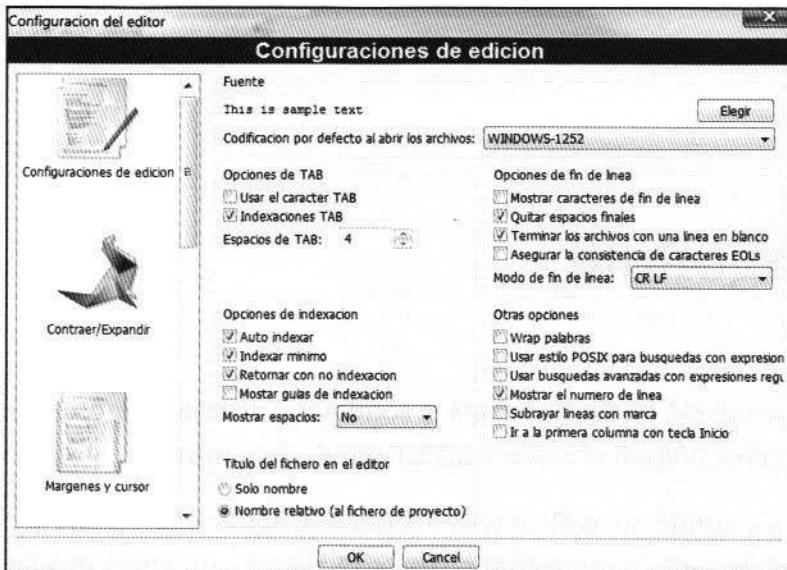


Figura 11.13: Configuración general de la edición



Figura 11.14: Configuración general de la generación y depuración

Bibliografía

- [1] José Antonio Cerrada Somolinos y Manuel Collado Machuca. *Fundamentos de Programación*. Editorial Universitaria Ramón Areces, 2010.
- [2] G. A. Miller. *The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information*. The Psychological Review, 1956, vol. 63, pp. 81-97.
- [3] Julian Smart and Kevin Hock with Stefan Csomor. *Cross-Platform GUI Programming with wxWidgets*. Pearson Education, Inc. 2006.
- [4] TIOBE Software: www.tiobe.com
- [5] Web oficial de la asignatura *Fundamentos de Programación con C±*:
<http://www.issi.uned.es/fp/>
- [6] Web oficial de Code::Blocks: www.codeblocks.org
- [7] Web oficial de wxWidgets: www.wxwidgets.org