Herramientas y proyectos

Fundamentos de programación

A. Garrido

UNIVERSIDAD DE GRANADA

HERRAMIENTAS Y PROYECTOS

Fundamentos de programación

A. GARRIDO

HERRAMIENTAS Y PROYECTOS

Fundamentos de programación

GRANADA
2020

HERRAMIENTAS Y PROYECTOS Fundamentos de programación

Printed in Spain *Impreso en España*.

Índice general

1 Introducción a Code::Blocks 1

1.1 Introducción 1

1.1.1 ¿Por qué Code::Blocks? 1

1.1.2 Instalación en Windows 2

1.2 Primeros pasos con *Code::Blocks* 2

1.2.1 Archivos en disco 6

1.3 Compilación y ejecución 6

1.3.1 Archivos en disco 8

1.4 Errores de compilación 8

1.4.1 Estructura de un programa 8

1.4.2 Compilación 9

1.4.3 Introducción a la E/S 9

1.4.4 Ejemplo de error de compilación 10

1.5 Opciones de configuración 12

1.5.1 Flags de compilación 12

1.5.2 Sangrado y estilo 14

1.5.3 Codificación del programa fuente 14

1.6 Problemas 18

A Tablas 19

A.1 Tabla ASCII 19

A.2 Operadores C++ 20

A.3 Palabras reservadas de C y C++ 21

Bibliografía 23

Referencias electrónicas 23

Índice alfabético 25

1 Introducción a Code::Blocks

Introducción 1
¿Por qué Code::Blocks?
Instalación en Windows
Primeros pasos con *Code::Blocks* 2
Archivos en disco
Compilación y ejecución 6
Archivos en disco
Errores de compilación 8
Estructura de un programa
Compilación
Introducción a la E/S
Ejemplo de error de compilación
Opciones de configuración 12
Flags de compilación
Sangrado y estilo
Codificación del programa fuente
Problemas 18

1.1 Introducción

En este guión se presenta un "Entorno de desarrollo integrado" (en inglés, *"Integrated Development Environment"* o *"IDE"*) para el desarrollo de programas en C++.

Un *IDE* es un programa especialmente diseñado para facilitar el desarrollo de software. Para entenderlo mejor, podemos listar algunos de los módulos que forman parte de él:

- *Un editor de código fuente*. Es un editor de texto que nos permite escribir uno o varios ficheros con el programa que se está desarrollando.
- *Un compilador*. El conjunto de programas necesarios para transformar el programa fuente en un programa ejecutable.
- *Herramientas para la generación automática de los ejecutables*. Programas para facilitar el manejo de múltiples fuentes y recursos para formar el programa final.
- *Depurador*. Permite trazar (ejecutar paso a paso) el programa fuente y estudiar su comportamiento para detectar y corregir errores.

En este documento se presenta *Code::Blocks*, que se usará durante el curso para desarrollar programas en C++. A pesar de ello, tenga en cuenta que aunque éste es un entorno especialmente adecuado para esta asignatura, el objetivo fundamental de la misma es aprender programación haciendo uso del lenguaje C++.

Por tanto, si se desea usar otros entornos, como *"Microsoft Visual Studio"*, *"Dev-C++"*, *"QtCreator"*, etc. la asignatura puede desarrollarse sin ningún problema. Tenga en cuenta que la mayor parte del tiempo y del trabajo se centrará en aprender a programar en C++.

1.1.1 ¿Por qué Code::Blocks?

El entorno de desarrollo *Code::Blocks* es:

- Software libre (Licencia: *GNU General Public License* versión 3).
- MultiPlataforma: *GNU/Linux, Windows, Mac OS X*[1].
- Ligero: No requiere excesivos recursos de la máquina.

Por otro lado, es importante tener en cuenta que no es un compilador. El entorno está desarrollado para trabajar con distintos compiladores; creados por terceros. En el diseño, se ha intentado independizar la funcionalidad del entorno de las características concretas de cada compilador. Así, resulta muy simple adaptarlo a la mayoría de ellos. Por ejemplo, el entorno permite crear programas con el compilador de la GNU (en *GNU/Linux* o *Windows*), *MS Visual C++ Free Toolkit 2003*, *Borland's C++ Compiler 5.5*, etc.

Además, el diseño lo constituye un núcleo altamente configurable mediante complementos (*"plugins"*), es decir, módulos adicionales que se incorporan al entorno para aumentar la funcionalidad. De hecho, una buena parte de las herramientas más avanzadas del entorno vienen dadas por esos complementos. En particular, se ha creado un módulo para poder manejar el lenguaje *Fortran* con este entorno.

[1] El equipo de desarrollo de *Code::Blocks* no cuenta con programadores que mantengan la versión para MacOs, lo que hace poco recomendable su uso en esta plataforma.

Algunas de las características que lo hacen un buen entorno para aprender a desarrollar son:

- Permite configurar el espacio de trabajo, adaptándolo a la tarea que se esté realizando.
- Facilita la gestión de distintos proyectos, con múltiples archivos y dependencias.
- Facilita la edición. Desde el coloreado del texto hasta el auto-completado o la generación de estructuras básicas del lenguaje.
- Facilita la generación del programa ejecutable facilitando la compilación, gestionando las dependencias incluso entre proyectos.
- Facilita la depuración, incluyendo herramientas clásicas que podemos encontrar en cualquier depurador (puntos de ruptura, consulta de valores, etc.).

Por último, y no menos importante, permite a cualquier usuario disponer de una plataforma de desarrollo gratuita para aprender a programar en C++.

1.1.2 Instalación en Windows

Para comenzar con la asignatura, la primera tarea es la instalación del entorno de desarrollo, por ejemplo, en el sistema de *Microsoft Windows*, probablemente el más conocido y usado por la mayoría de los estudiantes. Sin olvidar que este entorno también se puede instalar y usar en linux sin perjuicio para la asignatura.

En primer lugar, deberá descargar el paquete de instalación. En nuestro caso, usaremos el entorno junto con el compilador de la GNU MinGW (Minimalist GNU for Windows), por lo que lo más sencillo será descargar el paquete que integra el entorno junto con ese compilador. Concretamente, el archivo codeblocks-20.03mingw-setup.exe que puede encontrar en http://www.codeblocks.org/.

Una vez descargado, sólo es necesario ejecutarlo para que comience la instalación. Todo es muy simple y automático; basta con avanzar pulsando *Next* y aceptar la configuración por defecto.

Después de la instalación, puede lanzar el entorno. En la ejecución, el programa detecta automáticamente los compiladores que tiene instalados para poder configurar el entorno con un compilador concreto. Si no tiene otro entorno y ha instalado el paquete que incluye el compilador de la *GNU —MinGW—* lo detectará para ser usado en los proyectos que desarrollemos.

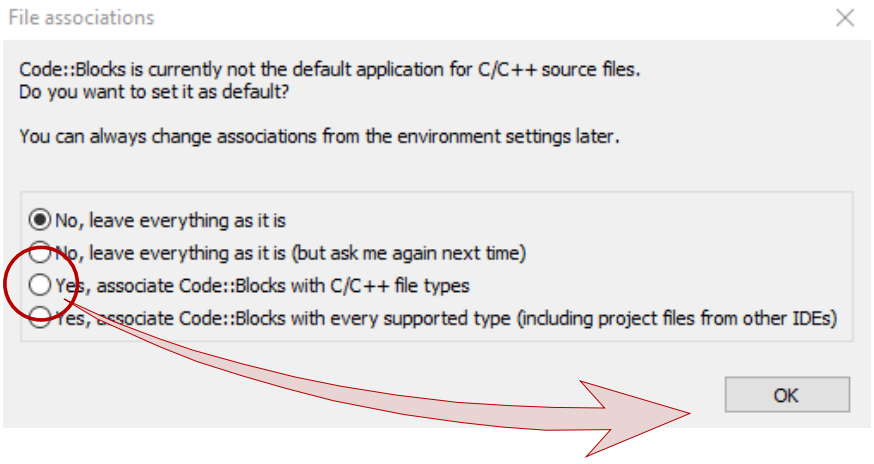

Figura 1.1
Asociación de archivos en *Code::Blocks*.

Además, puede aparecer una ventana como la de la figura 1.1 en la que se pregunta por la asociación de archivos fuente. En esta ventana central se presenta la opción de asignar los archivos C++ con este entorno de forma que la apertura por defecto de un archivo fuente —por ejemplo, un archivo cpp— lanzará el entorno *Code::Blocks*. Lo recomendable es que seleccione la tercera opción para asociar los archivos C++.

1.2 Primeros pasos con *Code::Blocks*

En las siguientes secciones se presentan detalles de cómo usar el entorno de desarrollo. Tenga en cuenta que los ejemplos gráficos donde se muestra este entorno pueden no coincidir exactamente con la última versión que tiene instalada. Comprobará que las diferencias no afectan, pues los aspectos que se discuten no han cambiado.

El primer paso es crear una carpeta para almacenar los proyectos de desarrollo que vayamos añadiendo al curso. En principio puede estar situada en cualquier lugar, aunque se recomienda una carpeta con una ruta donde no haya caracteres especiales. Los desarrolladores del entorno trabajan en inglés y no suelen centrarse en problemas concretos de plataformas o lenguajes, sobre todo, porque muchos de ellos dependen del compilador o configuración del usuario.

El entorno es multiplataforma e independiente del compilador que se usa. Debido a la variedad de configuraciones y de lenguajes que pueden usarse, conviene prevenir problemas de codificación en las llamadas al compilador así como en el procesamiento de los resultados obtenidos. Lo más sencillo es usar rutas donde evitar espacios y caracteres que no estén en

el alfabeto inglés, incluyendo las tildes o la diéresis. Puede crear una carpeta en otra unidad donde almacena sus datos, por ejemplo, en `D:\proyectos`.

Para crear un nuevo programa, podemos optar por dos opciones:

- Crear y editar un archivo fuente cpp con todo el programa. Este archivo se puede abrir con el entorno y compilar/ejecutar para obtener el resultado. Es lo más sencillo, pero sólo permite tener un único archivo y ejecutarlo.
- Crear un proyecto. Permite un proyecto con un archivo que podría ampliarse con más si se desea. Además, puede configurar el entorno y las opciones especialmente para el programa, incluyendo desde características visuales a opciones de compilación y ejecución.

La segunda opción es lo más recomendable, no sólo porque nos ofrece más posibilidades, sino porque su estudio facilita entender mejor en qué consiste un entorno de desarrollo así como las herramientas que usa, especialmente las que se refieren a compilar, enlazar y depurar un ejecutable.

Para comenzar, vamos a crear un proyecto simple de C++ con un único archivo que, lógicamente, contendrá la función **`main`**. La forma más directa es pulsar en el enlace *crear un nuevo proyecto* que aparece en la ventana principal, como muestra la figura 1.2.

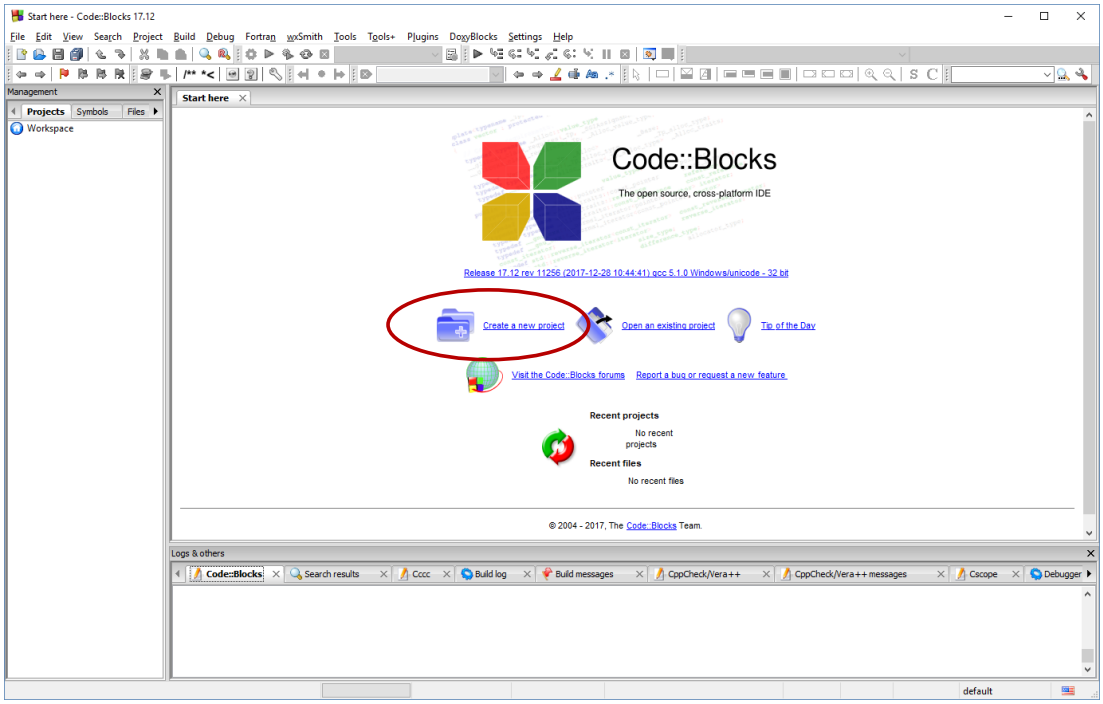

Figura 1.2
Crear un nuevo proyecto con *Code::Blocks*.

Tras pulsar en nuevo proyecto, el entorno nos ofrece un menú con distintas plantillas. La variedad es muy amplia, pues nos permite crear distintos tipos de proyectos, desde módulos para ser reutilizados a programas que incluyen una compleja interfaz gráfica. En nuestro curso de introducción, comenzamos creando pequeños programas con la interfaz más simple y estándar: la consola. Nuestros programas se ejecutarán en una consola, sin interfaz gráfica, con datos que introducimos desde el teclado[2] para obtener el resultado en la pantalla.

Por tanto, la plantilla que seleccionamos para nuestro proyecto es *aplicación para consola*, como muestra la figura 1.3. Después de pulsar, se presenta una pantalla de bienvenida a la configuración de esta plantilla. Pulse *Next* para comenzar.

Los parámetros que se solicitan son muy simples: si es una aplicación C o C++ en primer lugar, y el nombre y localización del proyecto en segundo. En la figura 1.4 se muestran los dos pasos.

Observe que hemos marcado los dos primeros parámetros, pues son los más relevantes. EL primero es el nombre del proyecto; ponga uno significativo, pues luego acumulará distintos proyectos y será necesario reconocer rápidamente cada uno de ellos.'El segundo es la carpeta donde guardarlo; seleccione una ruta simple evitando espacios y caracteres fuera del alfabeto inglés.

Los otros dos parámetros se generan automáticamente; en principio, no es necesario cambiarlos (en la sección 1.2.1 entraremos en más detalle). Déjelos con este valor automático.

Finalmente, una vez aceptados los parámetros, tenemos que seleccionar qué compilador vamos a usar y con qué configuraciones queremos trabajar. Esta selección es la que se muestra en la ventana de la izquierda de la figura 1.5.

Note que en la parte superior se selecciona el compilador. Seguramente en su sistema ya tiene este valor correctamente fijado. Por otro lado nos ofrece la posibilidad de activar dos configuraciones del proyecto. Son configuraciones habituales:

[2] Más adelante veremos que también trabajaremos con datos que se leen o escriben en archivos.

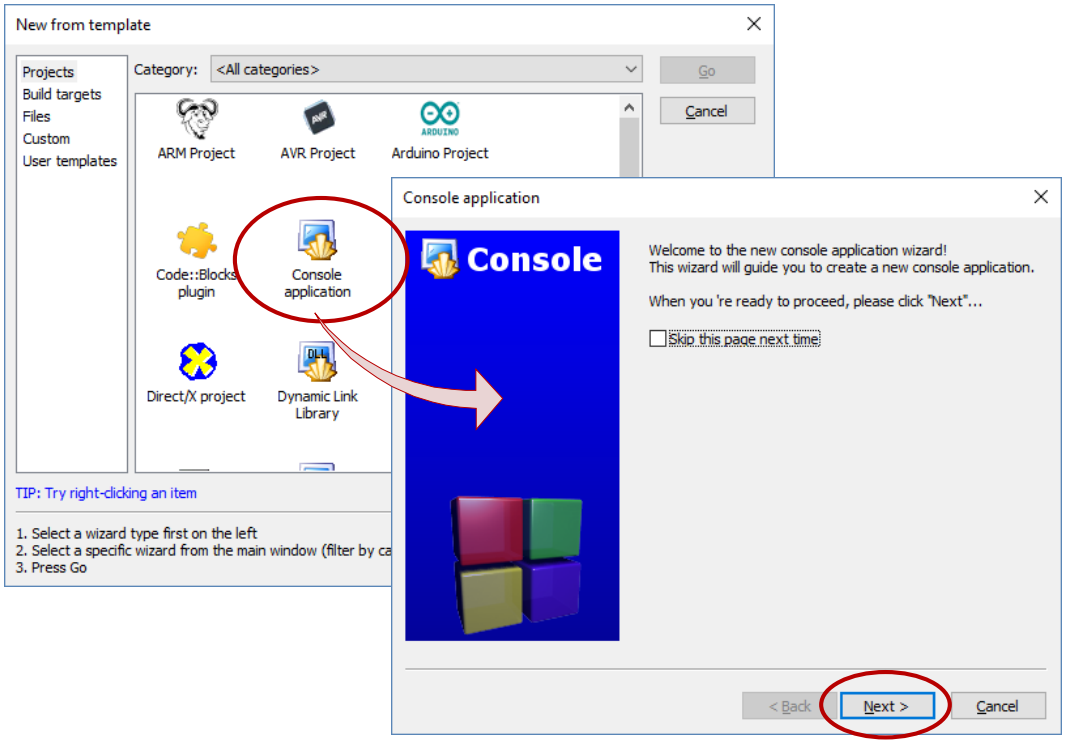

Figura 1.3
Crear un nuevo proyecto con *Code::Blocks*.

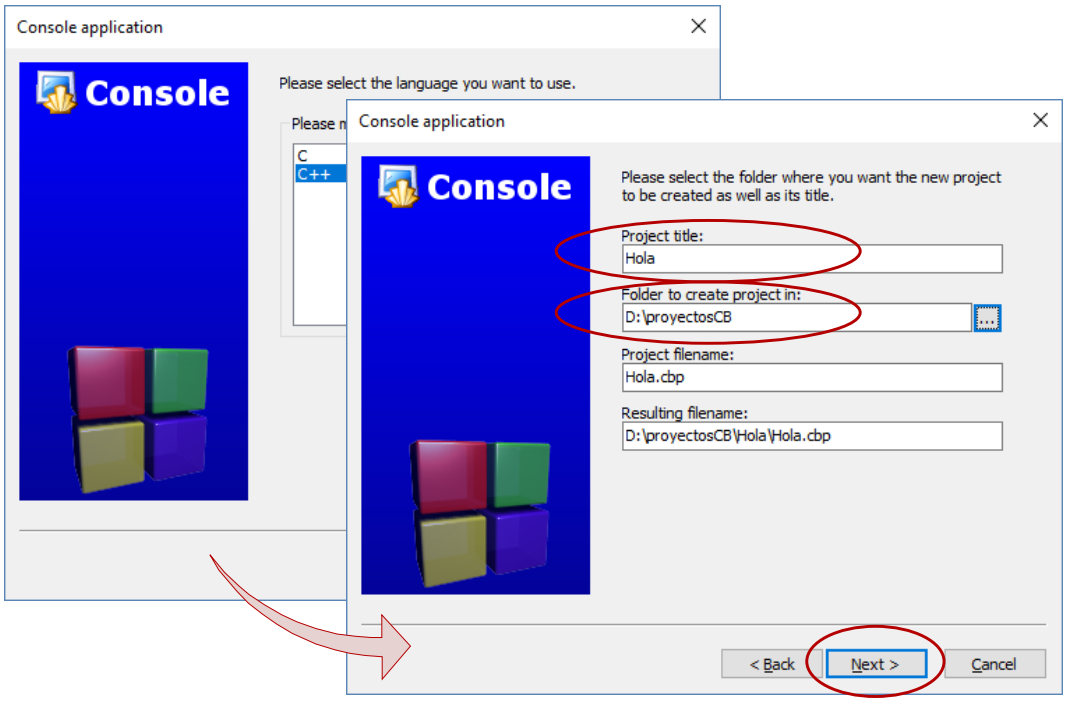

Figura 1.4
Localización y nombre del proyecto *Code::Blocks*.

- La primera —*Debug*— para trabajar en el desarrollo compilando, analizando el comportamiento, arreglando errores, etc.
- La segunda —*Release*— para generar un ejecutable definitivo.

En principio lo puede dejar sin tocar, aunque realmente siempre trabajaremos con la primera, por lo que puede desactivar la segunda; no es relevante, aunque más adelante volveremos a visitar este aspecto del entorno. Para terminar, pulse *Finish*.

Como resultado, obtenemos en la ventana principal —en la pestaña de la izquierda— el proyecto creado. Observe que tiene el nombre que hemos seleccionado y una carpeta que incluirá los fuentes —*Sources* en inglés—.

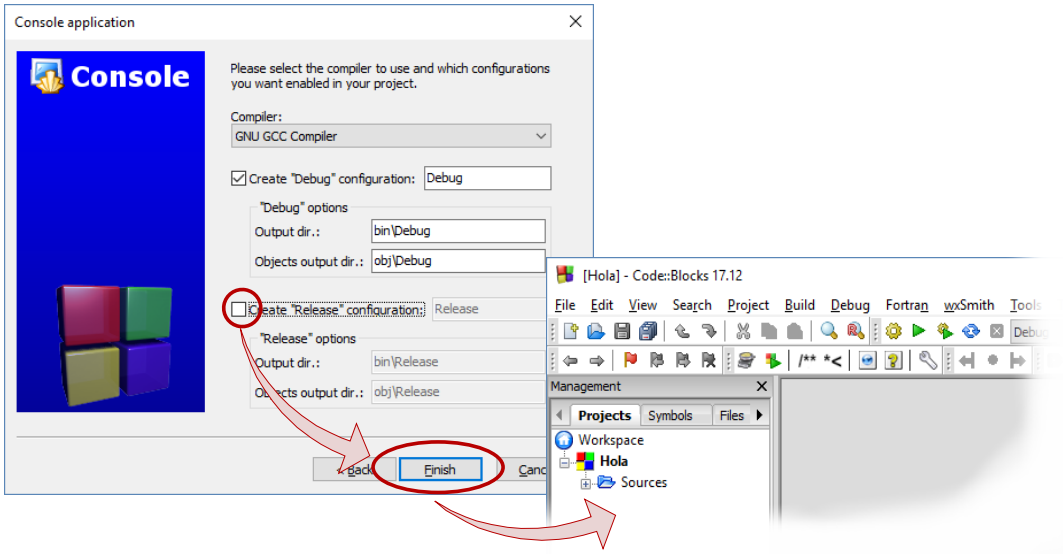

Figura 1.5
Configuración para depuración del proyecto *Code::Blocks*.

Si pulsa sobre el signo + verá que contiene un archivo con nombre main.cpp. Es un archivo que el entorno ha generado por defecto, con ese nombre porque contiene la función **main**. Para reconocer mejor el contenido de ese archivo, renómbrelo: pulse con el botón derecho sobre el nombre y escoja renombrarlo; ponga, por ejemplo, el nombre hola.cpp.

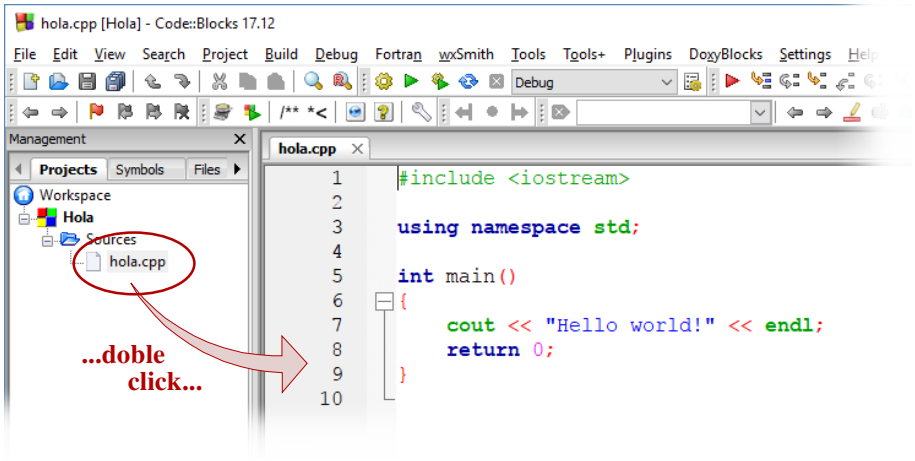

Figura 1.6
Proyecto *Hola* creado.

Ahora puede hacer *"doble click"* en el archivo hola.cpp para que el entorno abra un editor con su contenido. Obtendrá el programa:

```
#include <iostream>

using namespace std;

int main()
{
  cout << "Hello world!" << endl;
  return 0;
}
```

con el programa clásico *"Hola mundo"* que seguramente conoce. Este programa no es más que una plantilla para poder desarrollar nuestro proyecto.

Observe que dado que es un programa para la consola, ya ha incluido el archivo **iostream** y ha usado **cout** y **endl** para escribir en ésta.

Ejercicio 1.1 Pruebe ahora a cambiar el nombre del archivo hola.cpp. ¿Qué ocurre?

1.2.1 Archivos en disco

El entorno nos ayudará a gestionar un proyecto de una forma muy simple, presentando una interfaz intuitiva y accediendo a las utilidades con unos pocos clicks. Sin embargo, es especialmente útil ser consciente de qué ocurre internamente, sobre todo si surge algún problema con la gestión del proyecto y no sabemos exactamente cómo arreglarlo. Además, le ayudará a entender cómo funciona el entorno y el compilador.

Una vez creado el proyecto y obtenido el resultado mostrado en la figura 1.6, navegue por los archivos de su sistema y localice el directorio que escogió (recuerde la figura 1.4). Por ejemplo, si el directorio ha sido proyecto y el nombre Hola, se generan los archivos que puede ver en la figura 1.7.

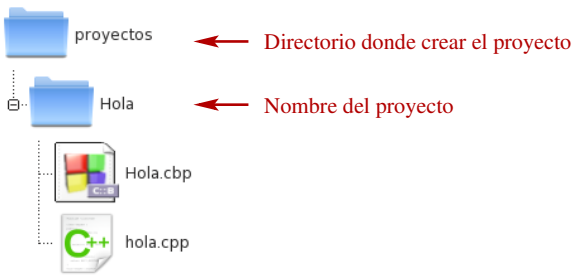

Figura 1.7
Ficheros generados al crear proyecto *Hola*.

Observe cómo se ha creado la carpeta y se han incluido los dos archivos que forman el proyecto con los parámetros que introdujimos en el paso de la figura 1.4. Seguramente, lo más novedoso se refiere al archivo Hola.cbp. La extensión cbp hace referencia a las iniciales de *"Code::Blocks project"*. Es el archivo para gestionar el proyecto. Lo genera y modifica el entorno de desarrollo automáticamente conforme a las opciones y configuraciones que establecemos.[3].

Ejercicio 1.2 Use un editor simple —por ejemplo, notepad— para consultar el contenido del archivo cbp.

1.3 Compilación y ejecución

Este programa generado está terminado y listo para traducir y ejecutar. Corresponde al programa *"Hola mundo"* típico como primer ejercicio en el estudio de un lenguaje de programación, y que además nos puede servir para confirmar el correcto funcionamiento de nuestro entorno de programación.

El efecto de ejecutar este programa es, simplemente, el de imprimir el mensaje *Hola mundo*. Dado que el programa está terminado, es el momento de generar el ejecutable a partir de los fuentes. Para ello, podemos usar la barra de herramientas para construcción —*build*— y ejecución —*run*— de programas.

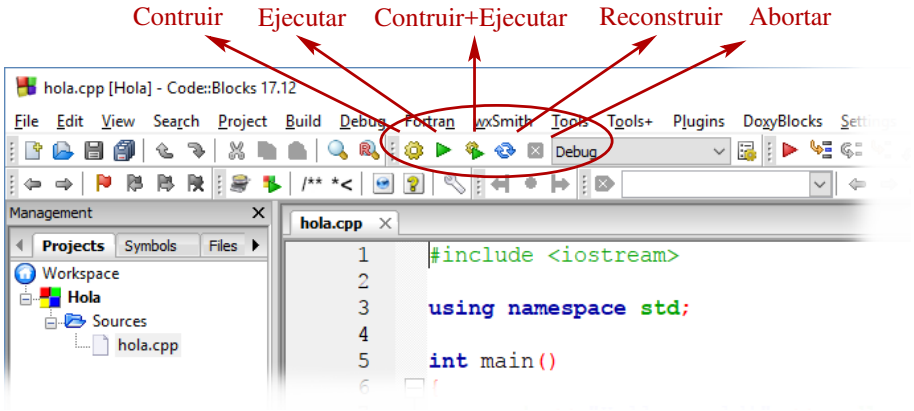

Figura 1.8
Barra para compilar y ejecutar.

En la figura 1.8 se muestra la localización y opciones de esta barra. Tenemos distintas posibilidades:

- *Build*. Permite generar el ejecutable. Se comprueban los cambios realizados y, si es necesario, crea de nuevo el ejecutable.
- *Run*. Permite lanzar el ejecutable.
- *Build and run*. Las dos anteriores en un solo paso.

[3]No tiene que modificarlo, aunque es editable con un simple editor de texto. Un programador con experiencia podría optar por revisarlo y modificarlo manualmente.

- *Rebuild*. Generar de nuevo todo el proyecto. Independientemente de si se han realizado cambios o no, vuelve a obtener el ejecutable desde los fuentes.
- *Abort*. Detiene la ejecución de un programa.

Para probarlo, vamos a generar el ejecutable pulsando el botón *Build* —la rueda dentada—. El resultado es que el entorno lanza las órdenes para compilar y enlazar el archivo hola.cpp, obteniendo un resultado como el que se muestra en la figura 1.9.

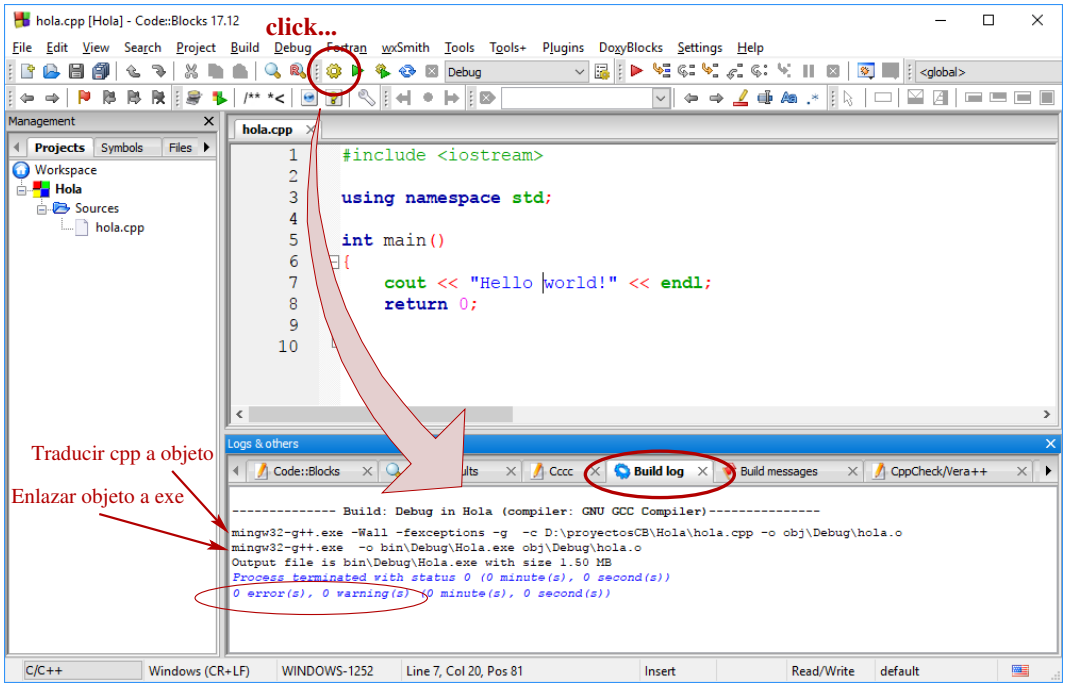

Figura 1.9
Resultado de compilar el proyecto *Hola*.

En esta figura hemos resaltado cómo en la pestaña *"Build log"* aparecen los mensajes que informan sobre el resultado de la construcción del ejecutable. Normalmente no nos vamos a fijar en estos detalles, pero en esta primera ejecución resulta interesante observar que:

- Una ejecución de mingw32-g++.exe se encarga de traducir el archivo hola.cpp al archivo hola.o.
- Una ejecución de mingw32-g++.exe se encarga de enlazar el archivo hola.o para obtener el archivo Hola.exe.

Aunque se usa la misma orden, no hacen lo mismo; el proceso de traducción y enlazado son muy distintos. Aunque están muy relacionados, realmente se realizan con distintos programas; aquí podemos ver que internamente el entorno usa la misma orden para gestionar las dos etapas. Cuando la orden ve que queremos traducir a objeto, llama al compilador y cuando ve que queremos obtener el ejecutable llama al enlazador.

Finalmente nos informa sobre el tamaño del resultado así como el resumen. Cuando un proceso devuelve un cero como resultado (*"Process terminated with status 0"* indica que no ha habido errores. Además, podemos observar que ha habido 0 errores (*0 errors*) y 0 avisos (*0 warnings*).

Si desea probar el programa, puede pulsar el triángulo verde, lo que hace que el sistema lance el ejecutable en una consola. En ella podrá observar que se imprime el mensaje *Hello world!*. En la figura 1.10 puede verlo.

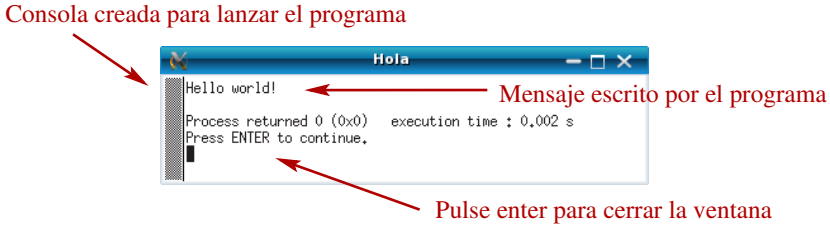

Figura 1.10
Ejecución del programa *hola*.

Al finalizar el programa, el entorno añade un mensaje indicando el resultado de su ejecución (*"Process returned 0"*) y el tiempo que ha tardado. El resultado es un valor entero, que si vale cero, indica que la ejecución ha terminado con éxito. Cualquier otro valor indica un error de ejecución. Recuerde la instrucción **`return`** que incorpora el entorno automáticamente. Esta instrucción es la que indica el valor a devolver. En caso de no ponerla, el compilador la añade al final de **`main`** automáticamente. Más adelante volveremos sobre ella.

1.3.1 Archivos en disco

Si volvemos a navegar sobre el sistema de archivos, y comprobamos los directorios y ficheros que se han generado, observamos una situación similar a la que muestra la siguiente figura 1.11.

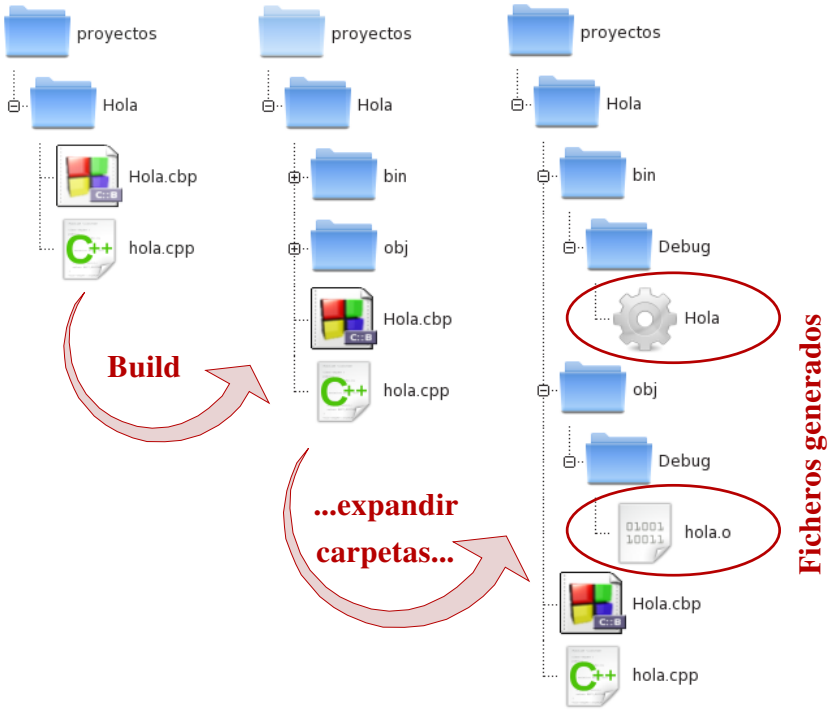

Figura 1.11
Archivos generados al compilar el proyecto *Hola*.

Como puede comprobar se han generado dos nuevas carpetas:

- Carpeta bin. En este directorio se encontrará el ejecutable correspondiente a nuestro programa. En *Microsoft Windows*, este fichero tendrá una extensión exe. En *GNU/Linux* será un archivo sin extensión. Este fichero es el resultado final de la traducción.
- Carpeta obj. En este directorio se encuentran los archivos *objeto*. Recuerde[4] que cuando se traduce un programa fuente C++, primero se traducen los archivos cpp a objeto (compilación) y finalmente se obtienen los ejecutables uniendo estos archivos objeto con los módulos necesarios para obtener un programa final (enlazado).

Por otro lado, se han generado los archivos en una carpeta Debug ya que, al crear el proyecto, hemos indicado que vamos a trabajar sobre esta configuración. Más adelante estudiaremos más detenidamente la creación de diferentes configuraciones y sus efectos.

Ejercicio 1.3 Una vez generado el ejecutable, ejecútelo tanto desde el explorador de archivos (doble click) como desde la consola (símbolo del sistema). ¿Qué diferencia observa?

1.4 Errores de compilación

En esta sección vamos a estudiar la estructura de un programa simple, algunos detalles sobre su contenido, y cómo el compilador los traduce detectando los posibles errores.

1.4.1 Estructura de un programa

En el programa de ejemplo que acabamos de presentar, con el que mostramos un mensaje, podemos observar tres partes diferenciadas:

1. Directiva `#include` que indica al sistema que se incluyan los recursos que el lenguaje C++ ofrece en el fichero **`iostream`**. En este caso, este fichero es necesario para poder usar **`cout`** y **`endl`**. A estos archivos (**`iostream`** y otros) los denominaremos *"de cabecera"*.
2. Instrucción **`using`**, indicando al sistema que usaremos **`std`**. Así podemos usar directamente los identificadores que ofrece el estándar de C++. La incluiremos en todos nuestros programas para facilitar la escritura.
3. Módulo **`main`**, donde aparecen las distintas instrucciones —separadas con `';'`— que componen el programa que desarrollamos.

[4] Cuando estudie la compilación separada, verá más claramente cómo se separan estas dos fases de la traducción.

Este bloque **main** es especialmente importante, ya que la ejecución de nuestro programa consiste en la ejecución secuencial de cada una de las líneas que componen este módulo. En los primeros programas, de tamaño relativamente pequeño, esta función cobra aún más importancia.

1.4.2 Compilación

Realizamos algunas modificaciones sobre el programa hola.cpp que hemos mostrado anteriormente para motivar algunos detalles más sobre el funcionamiento del compilador. En primer lugar, el siguiente programa es equivalente al anterior, ya que el resultado de su ejecución es idéntico:

```
// Programa hola.cpp
// Escribe un saludo
// ————————— Comienzo sección includes
#include <iostream>    // Permite usar cout y endl

// Instrucción using para usar el estándar
using namespace std;

int main()  {
    // <-- Aquí comienzan las instrucciones del programa

    cout << "Hello world!" // El mensaje
         << endl;          // Salto de línea

    /* La última instrucción, que es opcional */  return 0;
}
```

Observe que en este listado:

- Hemos introducido líneas que empiezan con `//`, lo que indica que son comentarios que el compilador debe ignorar hasta el final de la línea. Se introducen para documentar el código fuente, aclarando detalles a los posibles interesados en entender el código (ya sea el propio autor u otros programadores).
- La línea que comienza por **cout** la hemos dividido en dos partes. El programa se compone de una secuencia de *"tokens"* que pueden estar separados por espacios, tabuladores y saltos de línea. Para el compilador, la separación de esos *tokens* es igualmente válida independientemente del número de separadores y su disposición.
- La última instrucción **return** es opcional. Hemos incluido otro comentario, que como no queremos que alcance el final de línea, lo hemos insertado con el par `/* */`.

El proceso de traducción que realiza el compilador se puede esquematizar como:

1. Se eliminan los comentarios. Realmente no se traducen, ya que el proceso de traducción propiamente dicho recibe un programa sin comentarios.
2. Se recorre el programa, desde la primera línea a la última, extrayendo todos y cada uno de los *"tokens"* que lo componen[5]. Podríamos decir que el compilador divide todo el código en una secuencia de *"palabras"*, cada una de ellas independiente. Las *"palabras"* se pueden separar con espacios, tabuladores, saltos de línea, etc. Al compilador le es indiferente el que dos *"palabras"* consecutivas estén diferenciadas por un único separador, por varios, o incluso distintos tipos de separadores. Por ello, el programa anterior es equivalente al original.
3. Se analiza la sintaxis de la secuencia de *"tokens"*. Al igual que en castellano no sería sintácticamente correcto que un artículo apareciera dos veces seguidas en una frase, el compilador analiza[6] si la secuencia de *"tokens"* responde a la sintaxis del lenguaje. Por ejemplo, después de "main" el compilador espera que siga el token `'('`.
4. Finalmente, es necesario convertir esa secuencia en el código objeto, es decir, código traducido. Para ello, deberá determinar el significado de las sentencias que estamos analizando y sintetizar el resultado de la traducción[7].

Los detalles de esta traducción son bastante más complicados, aunque no vamos a entrar en detalle para el nivel en el que estamos; sin embargo, es importante que el alumno entienda que nuestro programa no es más que una secuencia de *"tokens"* que se deben analizar para comprobar que son correctos, y así poder obtener el resultado de la traducción.

1.4.3 Introducción a la E/S

Un programa recibe un conjunto de datos de entrada y obtiene unos datos de salida. Para ilustrar esta idea, así como para mostrar cómo se realizan estas operaciones mediante la consola, vamos a estudiar un ejemplo en el que se incluyen entradas y salidas. Por ejemplo, podemos escribir un programa para calcular la suma de dos valores introducidos por teclado:

```
1  #include <iostream>  // cout, cin, endl
2  using namespace std;
3  int main()
4  {
5     double dato1;
6     double dato2;
7
8     cout << "Introduzca el dato 1: ";
9     cin >> dato1;
10    cout << "Introduzca el dato 2: ";
11    cin >> dato2;
```

15

[5] Técnicamente, la extracción de la secuencia de *"tokens"* se denomina análisis léxico.
[6] Técnicamente hablamos de análisis sintáctico.
[7] Técnicamente hablamos de análisis semántico y síntesis.

```
12
13      double suma;
14      suma= dato1+dato2;
15
16
17      cout << "Resultado de la suma de "
18           << dato1 << " con " << dato2 << ": " << suma << endl;
19  }
```

Si ejecuta este programa en su entorno, comprobará que el entorno lanza una consola de ejecución, solicita que se introduzcan dos datos numéricos, y escribe el resultado de la suma.

Ejercicio 1.4 Cree un nuevo proyecto (por ejemplo, con nombre Suma) y compruebe el correcto funcionamiento de este programa. Para crearlo, seleccione la opción desde el menú *File* como muestra la figura 1.12 y sin cerrar previamente el proyecto *Hola*.

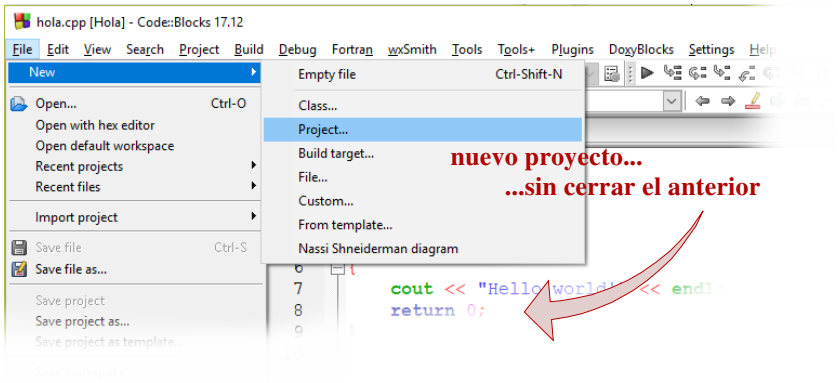

Figura 1.12
Nuevo proyecto *Suma*.

Aunque probablemente ya conoce los detalles de este programa, es importante recordar que:

- Hemos puesto un comentario a la primera línea. Realmente, hemos indicado el motivo por el que se incluye la línea `include`, ya que es necesaria para usar esos tres identificadores en nuestro programa.
- Se usan tres datos. Le hemos dado un nombre a cada uno: `dato1`, `dato2`, y `suma`. Cada uno de ellos aparece en una línea precedido del tipo **`double`**, que indica al programa que serán tres datos numéricos en coma flotante.
- Las líneas con **`cin`** se usan para programar las entradas. Observe que simplemente le tenemos que decir el nombre del dato que estamos leyendo.
- Las líneas con **`cout`** se usan para programar las salidas. Observe que el resultado que se obtiene en la salida corresponde al encadenamiento de cada uno de los datos que se le van indicando con <<.

Ejercicio 1.5 Intente modificar el anterior programa para leer 3 números y escribir su media aritmética.

Finalmente, es importante indicar que a este nivel del curso supondremos que los datos que le introducimos son correctos. Es decir, que si espera que le demos dos números, se van a introducir dos números, evitando errores al encontrarse con datos inesperados. Más adelante se estudiará con detalle por qué se generan y cómo podemos resolverlo.

Ejercicio 1.6 Compruebe el comportamiento del programa de suma cuando se introduce como primer dato una letra.

1.4.4 Ejemplo de error de compilación

Los ejemplos que hemos usado hasta ahora han sido programas sin errores. Para mostrar el comportamiento de Code::Blocks cuando el compilador encuentra errores, vamos a modificar el ejemplo anterior —de suma de dos números— introduciendo un error de compilación. Concretamente, vamos a eliminar el carácter `';'` de la primera línea del programa.

Ejercicio 1.7 Elimine el carácter punto y coma de la primera línea del programa de suma (donde aparece por primera vez `dato1`) y compruebe el resultado de compilar (pulsando *"Build"*).

El efecto de la compilación, después de pulsar el botón de la rueda dentada, se muestra en la figura 1.13. Es interesante que observe que el entorno que le presentamos en esta figura contiene dos proyectos abiertos. El primero el proyecto *Hola*, y el actual, el proyecto *Suma*.

Esta situación con varios proyectos se genera al abrir un proyecto sin cerrar el anterior. Podemos tener tantos proyectos como queramos. Como puede ver, el proyecto activo se presenta en negrita, que es el que se compila cuando pulsamos el botón *Build*.

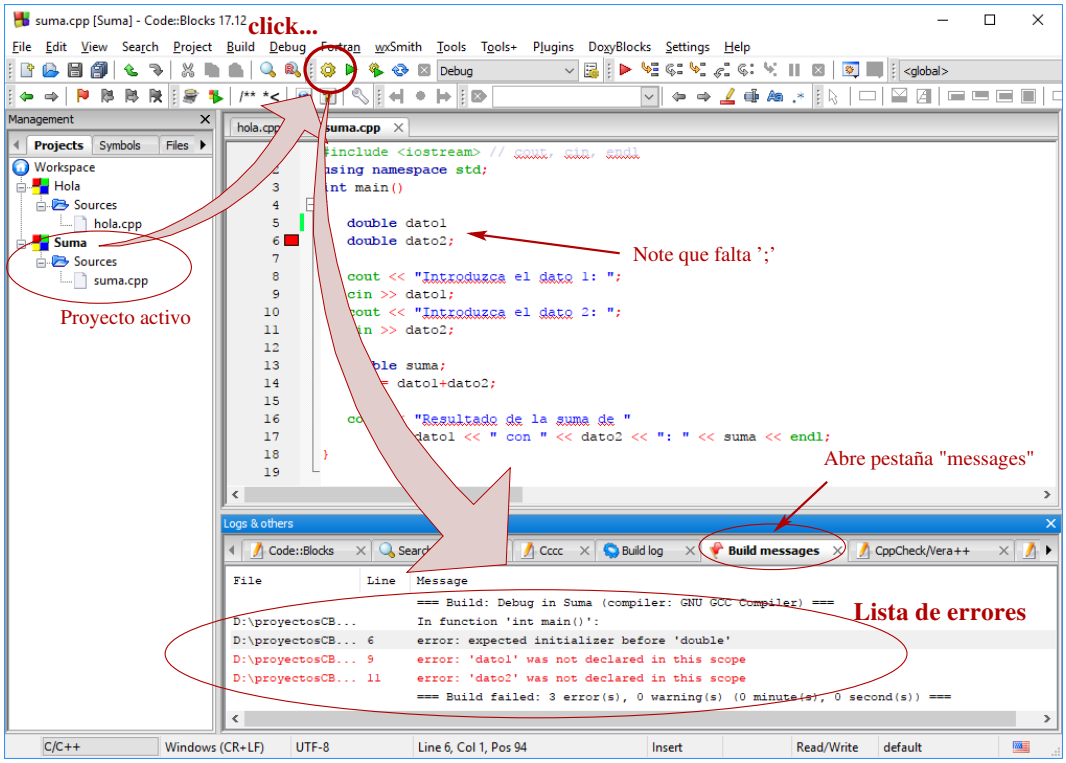

Figura 1.13
Error de compilación en el proyecto *Suma*.

Observe que el error se obtiene en la línea 6, aunque podemos decir que está en la línea 5. Recuerde que el compilador separa el programa en *tokens*, y por tanto, el carácter `';'` podría estar en la línea 7. El error, realmente, se ha detectado en la palabra **`double`** ya que ahí es donde se ha encontrado con un *token* inesperado.

Por otro lado, si lee el mensaje de error, verá que hace referencia a un inicializador; más adelante estudiaremos en detalle en qué consiste esta inicialización. Como puede ver, si añadimos un simple punto y coma al final de la línea 5, todo funciona perfectamente.

Lista de múltiples errores

En el ejemplo anterior el compilador ha mostrado tres errores en este programa, a pesar de haber cometido sólo uno. ¿Qué ha ocurrido?

Un aspecto importante que debemos tener en cuenta cuando compilemos es que, normalmente, cuando el compilador intenta realizar una traducción, no se detiene en el primer error que localiza, sino que intenta saltárselo y avanzar en el resto del programa fuente con la intención de generar un informe lo más completo posible.

El objetivo final de este comportamiento se debe a que si tenemos que realizar una compilación por cada uno de los errores, es posible que necesitemos un número muy alto de compilaciones y, por consiguiente, una pérdida de tiempo importante. Por tanto, el compilador intenta revisar todo el código con el fin de obtener una lista con todos los errores. Una vez que tenemos todos los errores, los podemos revisar y corregir sin necesidad de volver a compilar.

Sin embargo, la situación no es tan simple, ya que el compilador, para poder continuar con la traducción, no tiene más remedio que ignorar los trozos de código con error. Por tanto, es probable que muchos errores sean consecuencia de los anteriores.

Por ejemplo, como el compilador no ha sabido entender las líneas 5 y 6, las ignora; por tanto, genera un error en todas las líneas que hagan referencia a los nombres `dato1` y `dato2`.

En consecuencia, cuando hemos revisado y corregido los primeros errores, es recomendable volver a compilar para generar una nueva lista en lugar de perder tiempo revisando errores que pueden ser consecuencia de los anteriores.

En nuestros ejemplos, al ser tan simples, no cuesta demasiado volver a pulsar el botón para compilar y regenerar la lista de errores muy rápidamente. Cuando tenga proyectos más grandes, con un número de errores más alto, deberá corregir los primeros y decidir en qué momento es necesario volver a compilar.

Por consiguiente, no se sorprenda ni se preocupe especialmente si obtiene una lista muy larga de errores, corrija los primeros y vuelva a compilar.

Avisos (*warnings*)

Una utilidad especialmente útil —pues nos permite evitar muchos errores— son los avisos (*warnings*). Estos avisos no son errores en el sentido de que el código se puede traducir correctamente. Sin embargo, es código con pinta de que podría provocar fallos en la ejecución del programa.

Por ejemplo, supongamos que modificamos el código fuente del programa de cálculo de la suma de dos números. Concretamente eliminamos la instrucción de lectura del valor de `dato1`. Una forma muy simple de hacerlo es anteponer una doble barra de comentario antes de la palabra **cin**.

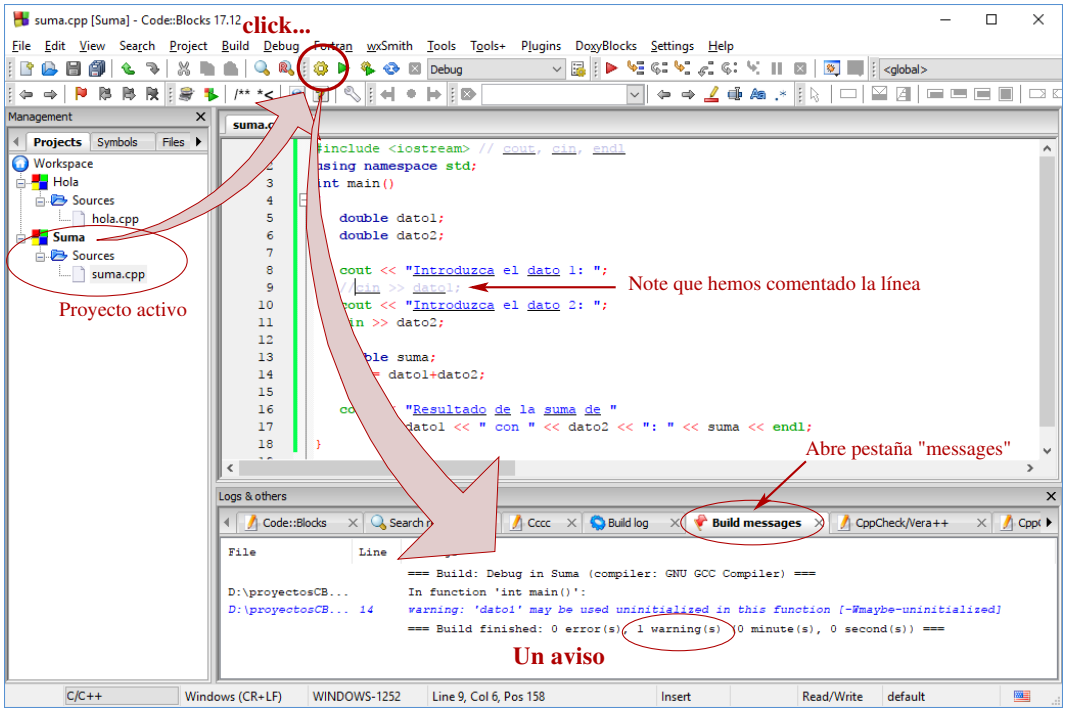

Figura 1.14
Aviso de compilación en el proyecto *Suma*.

Si pulsa el botón para compilar, podrá observar un resultado como el que se presenta en la figura 1.14, indicando que existe un aviso en la línea 14 (tenga en cuenta que el programa resultante es equivalente a no haber escrito la línea 9).

El compilador nos está avisando de que es posible que el programa sea incorrecto, ya que aunque se puede compilar y obtener el archivo ejecutable, hay un trozo de código "sospechoso". Efectivamente, es un error ejecutar el programa, ya que realizamos una suma entre dos valores, uno leído, y el otro indeterminado.

A pesar de todo, el programa es un programa C++ correcto sintácticamente y, por tanto, se puede traducir y ejecutar. Los avisos son simplemente mensajes para que el programador confirme que, efectivamente, es lo que desea. Algunos de ellos se podrán ignorar, porque no conducen a ningún error, otros no.

Lo más aconsejable es que se revisen y se eliminen por completo modificando el código fuente. Una compilación con múltiples líneas de aviso puede confundir y ocultar nuevos avisos que pasan desapercibidos.

Además, los compiladores suelen contener opciones para *activar* la generación de nuevos mensajes de aviso. Resulta recomendable activarlos para evitar algunos errores en tiempo de ejecución (consulte la sección 1.5).

1.5 Opciones de configuración

El entorno de desarrollo tiene muchas herramientas. No es necesario configurar todo en este punto del curso. En la práctica, su uso y la curiosidad por explorar posibilidades le llevará a adaptar muchas opciones de configuración.

En esta sección presentamos las opciones más básicas para empezar a trabajar directamente sin necesidad de que dedique demasiado tiempo a explorar los distintos menús. Más adelante, cuando se estudien otras herramientas, presentaremos nuevas opciones relevantes.

En la figura 1.15 se presenta la localización del menú y las tres opciones que nos van a interesar especialmente para trabajar con el entorno. En este guión vamos a incluir algunos comentarios sobre las dos primeras.

1.5.1 Flags de compilación

En las secciones anteriores se ha lanzado el compilador pulsando sobre la rueda dentada —opción *Build*— de forma que el entorno llama automáticamente al compilador. Además, hemos indicado que conviene que nos avise de los posibles errores. En la figura 1.16 se muestran las opciones que hemos señalado como aconsejables para este curso. Tenga en cuenta que en éste vamos a:

- Usar el estándar C++14. Realmente, la mayoría de contenido se refiere a fundamentos que ya existen en C++98, aunque se incluirán algunos detalles para incorporar tecnología más reciente. Por tanto, se puede activar esta opción en la parte superior.

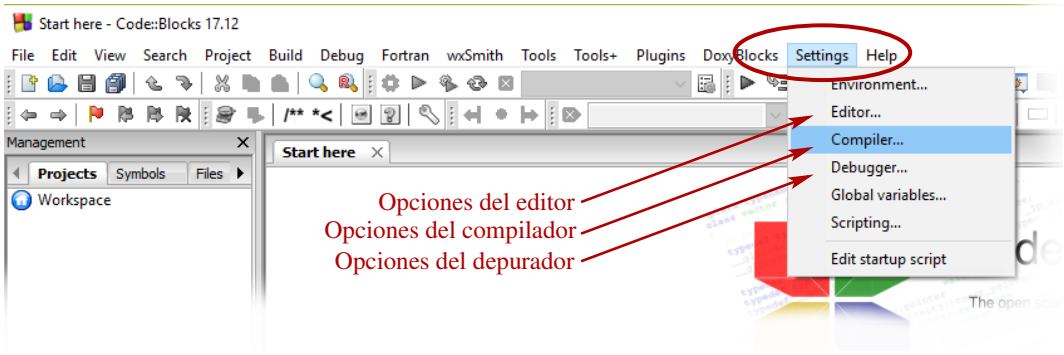

Figura 1.15
Menú *settings* de *Code::Blocks*.

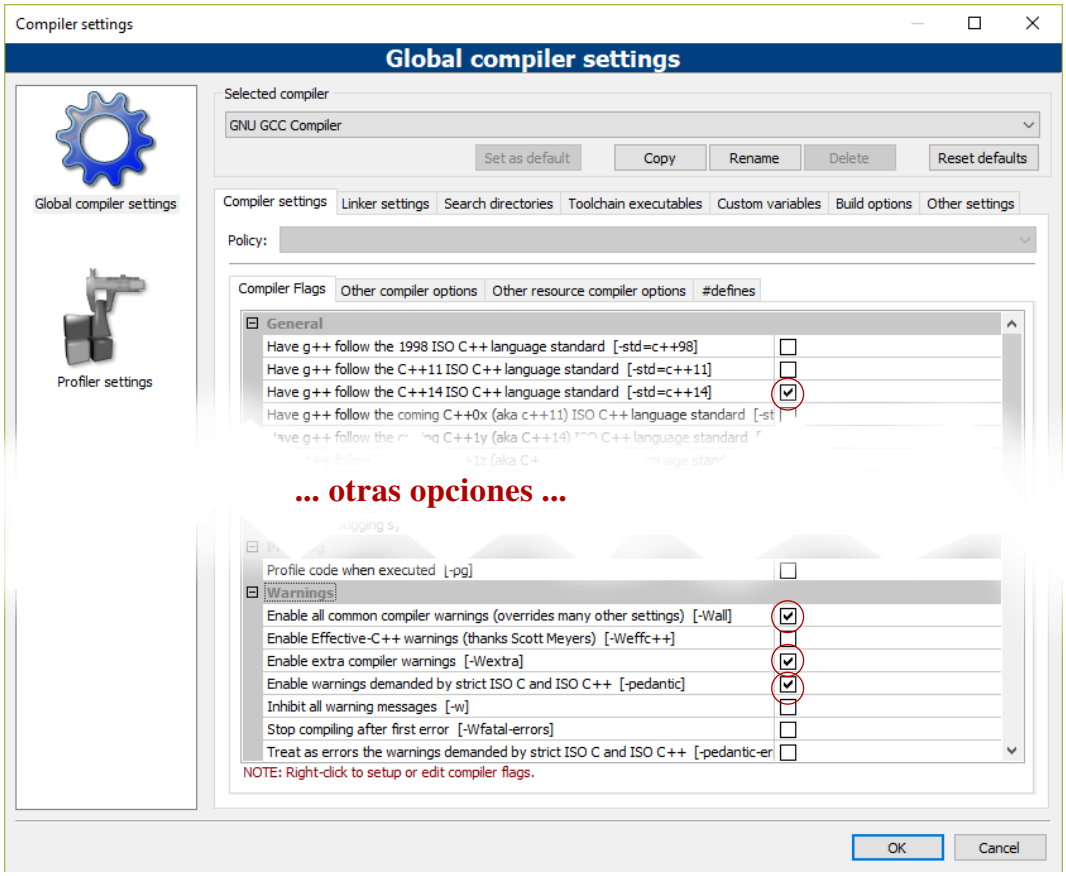

Figura 1.16
Flags del compilador.

En cualquier caso, si está usando la última versión del entorno con su correspondiente *mingw*, la versión por defecto del compilador incluido será ésta o posterior.

- Activar los avisos sobre código "sospechoso" —opción *all*— e incluir algunos avisos más que no están en la opción anterior —opción *extra*—.
- Activar el control sobre el estándar de forma que no se admita código que no siga el estándar. Algunos compiladores admiten código como una extensión, es decir, para ampliar las posibilidades del programador y facilitar algunas operaciones. Como nuestro interés es producir código estándar, activamos esta opción para que el compilador lo garantice.

Ejercicio 1.8 Abra uno de los proyectos que tiene —por ejemplo, *Hola*— y pruebe a compilarlo con estas opciones activadas. Compruebe los detalles que aparecen en la pestaña *Build log* de llamada al compilador.

En la sección 1.5.3 incluimos alguna opción más del compilador. La separamos pues las de esta sección son especialmente importantes y ampliamente usadas. Las opciones de la sección 1.5.3 abordan un aspecto más técnico relacionado con la codificación.

Guión: 1

1.5.2 Sangrado y estilo

La opciones del editor son muy numerosas; en muchos casos pueden ser muy distintas de un programador a otro. Por ejemplo, el aspecto puede ser muy distinto simplemente con seleccionar distintos tipos y tamaños de letra. Lo aconsejable es que con la práctica vaya adaptando las opciones a sus preferencias, sin embargo, conviene indicar algún detalle sobre el sangrado y estilo, pues si comparte código con otra persona puede resultar algo incómodo si no coinciden los estilos.

En concreto, resulta recomendable revisar las opciones de tabulación para el sangrado. Se puede realizar de dos formas: con tabuladores o con espacios. En el primer caso, sangrar un bloque de código un espacio implica insertar un tabulador, en el segundo, varios espacios.

El problema surge cuando se mezclan ambos estilos, especialmente porque el tabulador se suele representar de distintas formas. En muchos casos, se entiende que un tabulador puede equivaler a 8 espacios. Si tiene dos líneas con la misma sangría, una con un tabulador y otra con espacios, la segunda tendrá 8 caracteres para alinearla con la anterior.

El problema aparece cuando se abre ese código con otro editor y el tabulador no equivale a 8 espacios. Tenga en cuenta que sangrar 8 espacios generalmente es demasiado, pues en cuanto tiene código algo complejo con varias estructuras anidadas puede hacer que el programa no pueda visualizarse al salirse por la derecha.

En estos casos, lo más sencillo es indicar al editor que represente los tabuladores con menos espacios. En este punto es cuando el código anterior, en el que la segunda línea tenía 8 espacios, se desalinea. Si desea una solución que muestre siempre el mismo aspecto, use directamente espacios para sangrar; además, los editores suelen incluir la opción de sustituir el tabulador de forma que si se inserta uno, tabula mediante varios espacios.

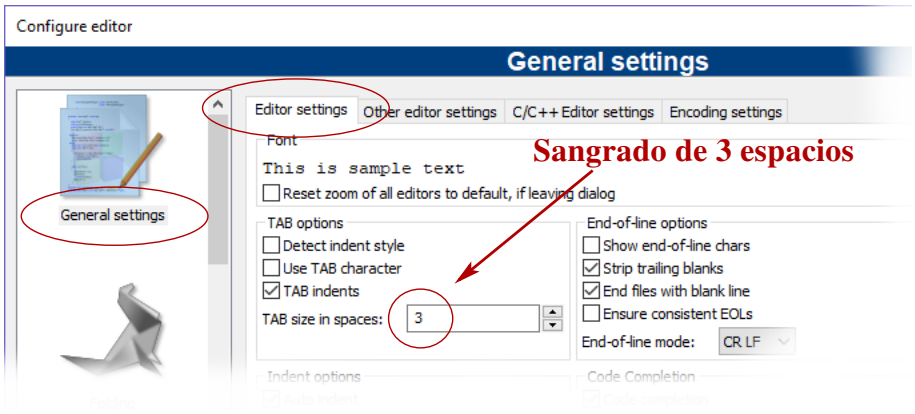

Figura 1.17
Opciones de tabulación.

En la figura 1.17 se muestran las opciones de tabulación del editor. Observe que se ha indicado que se desean 3 espacios[8] y además no se ha marcado la segunda opción que implica el uso de tabuladores en el archivo.

Estilo

El sangrado es sólo un aspecto puntual del estilo de escritura; hay muchas más opciones. Debería decidir un estilo y mantenerlo para facilitar la lectura del programa, especialmente si trabaja con otros programadores (véase, por ejemplo, Garrido[3] con una discusión y complejos relacionados con el estilo de codificación).

Si tiene un trozo de código con un mal estilo o de otra persona que no ha tenido cuidado, el entorno *Code::Blocks* también ofrece una opción para formatearlo. Puede seleccionar un trozo de código y usar el botón derecho para seleccionar *Format use AStyle*. En la figura 1.18 puede ver este menú desplegado y la opción en la parte de abajo.

Además, puede seleccionar distintos estilos. En el menú de configuración del editor puede bajar hasta la opción *Source formatter*. Encontrará distintos estilos y podrá seleccionar el que mejor se ajuste a tu equipo de trabajo. En la figura 1.19 puede ver un ejemplo en el que se ha seleccionado el formato `K&R` (de *Kernighan and Ritchie*). Note que hay una pestaña para seleccionar el sangrado del estilo.

En algunos casos, este formateo puede incluso ayudar a encontrar un error; la operación analiza los bloques de código y las llaves que aparecen de forma que el formateo puede delatar algún error. Por ejemplo, si tiene varias líneas sangradas a la derecha en un bloque pero olvidó las llaves, el formateo detecta que la primera línea debe quedar sangrada pero el resto no, pues no están en el mismo bloque.

1.5.3 Codificación del programa fuente

En esta sección vamos a revisar algunas opciones sobre codificación. Además, nos va a servir para visitar brevemente el problema de los idiomas y la codificación de caracteres. Es algo técnico que en muchos cursos de introducción se ignora, pues

[8] Suelen usarse 3 o 4 caracteres. En mi caso, prefiero el valor menor para evitar que el código se salga por la derecha.

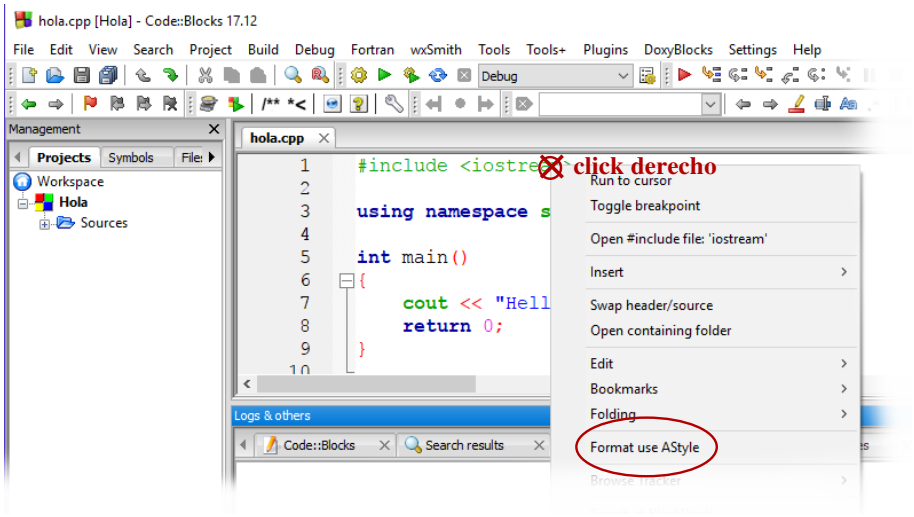

Figura 1.18
Opción de formatear con estilo.

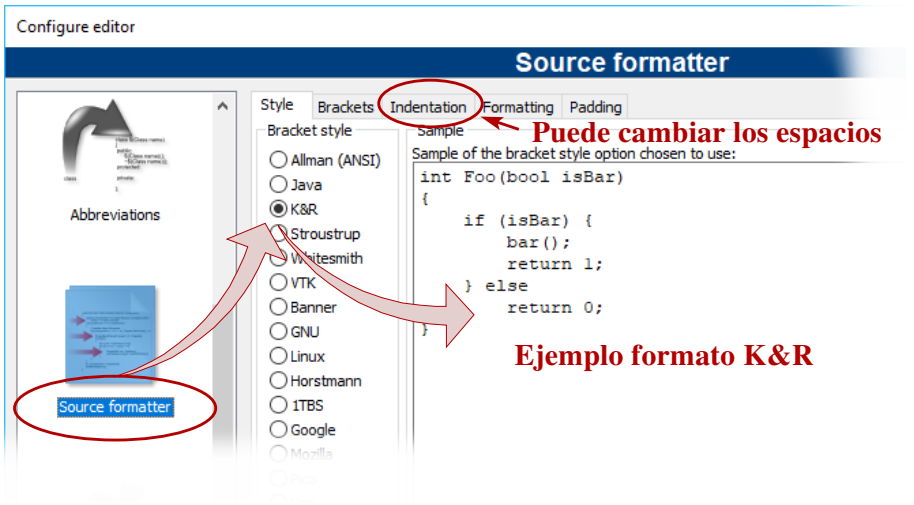

Figura 1.19
Opción de formatear con estilo.

se pueden explicar los conceptos básicos de programación sin preocuparnos sobre el idioma que usamos. De hecho, muchos cursos asumen que usan el inglés y no entran en más detalles[9].

Sin embargo, no debe confundirse. La codificación está en la base de la programación, por lo que la discusión puede resultar especialmente ilustrativa. Los objetos que maneja un programa son la codificación de los datos del problema real. Desde un simple entero que se representa mediante un número binario hasta una letra o un gráfico que maneja mediante una tabla de codificación.

Por tanto, es muy recomendable que intente entender el problema y cómo se resuelve. En Garrido[5] puede encontrar una discusión más extensa y problemas de programación relacionados con la codificación.

Editores y codificación

En primer lugar, es importante destacar que el código fuente de un programa se escribe con un editor de texto. El editor genera un fichero que contiene la secuencia de caracteres que corresponde al programa. Este fichero está codificado según una tabla de codificación.

Hablamos de un editor de *texto simple* o *texto plano* —del inglés *plain text*— pues no codifica más que la secuencia de caracteres, sin detalles sobre tipos de letra, tamaños, estilo de escritura, etc.

¿Qué codificación usa el compilador? En principio, podemos decir que el archivo cpp es un archivo *ASCII*. Esta codificación es una tabla que hace corresponder a cada carácter un número en el rango $[0, 128]$. Sólo necesita 7 bits y es suficiente para el idioma inglés. Si revisa un manual de referencia del lenguaje C++, verá que todos sus caracteres e identificadores están en esta tabla.

[9] Realmente, en los siguientes capítulos trabajaremos de esta forma; escribiremos programas como si los ejecutáramos en inglés, centrándonos en el problema y el algoritmo que lo resuelve de forma que el problema de la codificación no afecte a la solución.

El problema de esta tabla es que no sirve para otros idiomas. No incluye caracteres como las vocales acentuadas, la letra eñe o la diéresis. Para resolverlo, se crea una tabla extendida con un bit más, es decir, de 256 posibilidades; para Europa occidental se puede usar la llamada *Latin-1* o *ISO-8859-1*.

En la tabla A.1 (página 19) puede ver en detalle la *tabla ASCII extendida* que permite codificar estos idiomas; además, es la tabla *15*, diseñada para Europa occidental y que incluye el símbolo del euro (€). La mitad de arriba es la tabla ASCII, la de abajo, la extensión.

Si usa un editor con esta codificación, podrá comprobar que cada letra que escribe aparece como un byte —8 bits— en el archivo resultado. El archivo no es más que una secuencia de números del rango $[0,255]$. Los editores modernos suelen tener menús para seleccionar la codificación deseada. La mayoría de las veces se puede ignorar, porque al abrir un archivo, el mismo editor lo revisa para detectar la codificación y adaptarse a ella.

Tablas de codificación

El problema surge cuando empiezan a aparecer distintas tablas de codificación. Si un programa genera un texto con una codificación y dicho texto se procesa por otro programa, es fundamental que ambos se pongan de acuerdo: o fijan una misma codificación o detectan y se adaptan a la codificación recibida.

En primer lugar, la codificaciones estándar *ISO-8859-15* sirve para Europa occidental. No sirve para otros países que tienen incluso otro alfabeto. Un griego, un ruso o un chino no pueden usar esa codificación.

Por otro lado, las empresas pueden crear y adaptar codificaciones para sus sistemas. Por ejemplo, *Microsoft Windows* creó la codificación *Windows-1252* —también denominada *CP1252* o *code page 1252*— para actualizar la tabla *CP850* que usaba antes. Esta nuesta tabla no es muy distinta a *ISO-8859-1*, aunque sigue siendo una tabla para el alfabeto latino que no sirve para otros idiomas como griego, hebreo, ruso, etc. La intención era crear una tabla lo más amplia posible para codificar con un byte un gran número de idiomas; en concreto, la diferencia con *ISO-8859-1* es que cambia los primeros 32 caracteres de la parte extendida para incluir caracteres imprimibles en lugar de códigos de control.

Los primeros sistemas, desarrollados en un contexto de un idioma concreto, podían resolver los problemas usando una única codificación adaptada a ese idioma. Con el tiempo y la globalización, se ha hecho necesario que se pueda intercambiar información en múltiples idiomas. Por consiguiente, se han tenido que crear otras codificaciones más amplias y complejas.

Para resolverlo se crea *UNICODE* y las distintas codificaciones asociadas. Una codificación moderna muy conocida y que resuelve este problema es *UTF-8*. Es muy conocida porque es la que suele usar el sistema *GNU/Linux* y, por tanto, muchas aplicaciones.

Puede resultarle extraña porque pretende ser compatible con todo lo anterior, añadiendo todos los nuevos caracteres: es una codificación denominada de *longitud variable*. Básicamente, vuelca toda la tabla *ASCII* básica, es decir, los 128 primeros caracteres. Por tanto, si un inglés escribe un texto en *ASCII*, el archivo es también compatible con *UTF-8*.

La diferencia, por tanto, está en las entradas que no son *ASCII*. Por ejemplo, una letra `'a'` está en la parte extendida, en *UTF-8* se va a codificar como dos bytes (dos números del rango $[0,255]$ consecutivos). En la práctica, incluso puede haber entradas más largas, de tres o cuatro bytes.

Otra muy conocida es *UTF-16* porque la usa *Windows* para poder manejar la amplia mayoría de los lenguajes en el mundo. Es de *longitud fija*. Cada carácter ocupa dos bytes.

Como ve, hay multitud de tablas de códigos. Normalmente no nos vamos a preocupar de esto, pero es posible que alguna vez se encuentre con caracteres extraños y comportamientos que tienen su raíz en problemas de codificación.

Codificación en *Code::Blocks*

El *IDE* incluye un editor para el código. Este editor admite también la configuración de la codificación. Cuando lee un archivo, intenta adivinar la codificación para trabajar con ella. Si no es posible detectarla —por ejemplo, si abre un archivo nuevo— opta por la codificación por defecto.

Por ejemplo, el proyecto *Hola* que hemos abierto más arriba se ha creado con la codificación por defecto del sistema: *windows 1252* (*CP1252*). Es una buena opción para trabajar con *Microsoft Windows*, pues es simple, con un byte por carácter y compatible con bastantes idiomas en Europa occidental. En la figura 1.20 se muestra el pie de la ventana donde se incluyen detalles de edición, en concreto, la codificación que se está usando.

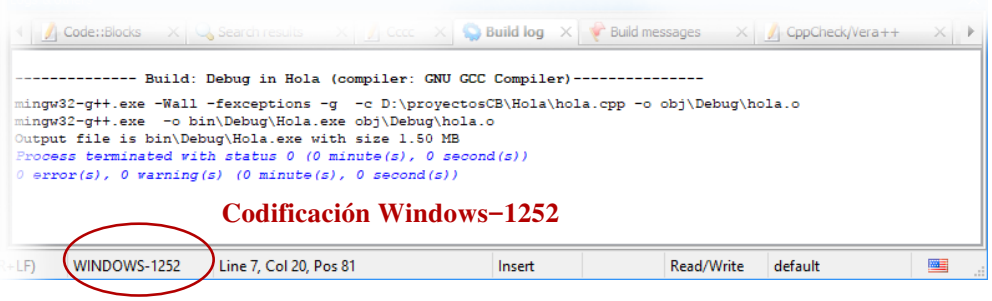

Figura 1.20
Página de códigos que está usando el editor.

Tenga en cuenta que aunque se haya abierto con una codificación, puede seleccionar otra si accede al menú *edit→file enconding*. Si desea fijar una codificación para cualquier archivo que abra, puede hacerlo en los menús de configuración del editor. En la figura 1.21 se muestra cómo se puede seleccionar una codificación *UTF8*.

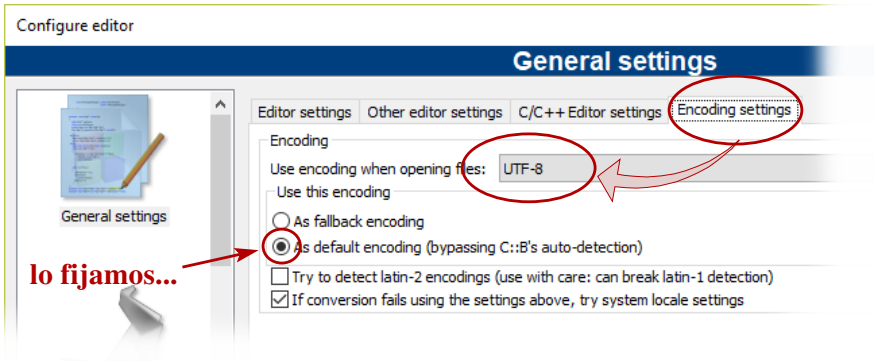

Figura 1.21
Fijando la codificación *UTF-8*.

Por otro lado, los programas que estamos ejecutando mandan la salida a una consola de ejecución. Esta consola es un programa que presenta los caracteres que recibe. Por tanto, cuando la consola tiene que presentar un carácter, tendrá que traducir el código al dibujo o gráfico correspondiente.

La orden chcp permite conocer (y modificar) la página de códigos activa en una consola. Si escribe la orden sin parámetros le informa de la página y si selecciona un número, la cambia. Por ejemplo, la 857 sirve para el turco o la 869 sirve para el griego.

En mi sistema, donde se ha abierto el proyecto con la edición fijada en *CP1252*, la página de código de la consola es la 850 que es válida para múltiples lenguajes (*latin-1*). Esta página se usaba intensivamente con los primeros sistemas *MS-DOS* de *Microsoft*. No había entorno gráfico y los terminales sólo sabían pintar los caracteres de la tabla de códigos. Busque en la red esta tabla, encontrará que incluía muchos caracteres gráficos tipo barra vertical, esquina superior izquierda, etc. Con éstos, era muy sencillo crear pantallas con recuadros y separaciones. Se cambió más adelante por la *CP1252* para poder incluir nuevos caracteres.

Si nuestro programa genera caracteres *CP1252*, se pintarán adecuadamente si manda estos códigos y la consola los sabe interpretar. La primera opción es pensar en configurar el sistema para que la consola use otra tabla de códigos. Sin embargo, vamos a evitar esta solución, pues si lo hacemos, también afectará a otros programas. En lugar de eso, vamos a informar del compilador de qué codificación usamos en los archivos fuente y qué codificación queremos que se use para el ejecutable.

Por ejemplo, en mi sistema he configurado dos opciones más para el compilador. Las puede ver en la figura 1.22. Le configuro que el archivo de entrada usa *CP1252* y el ejecutable usará *CP850*.

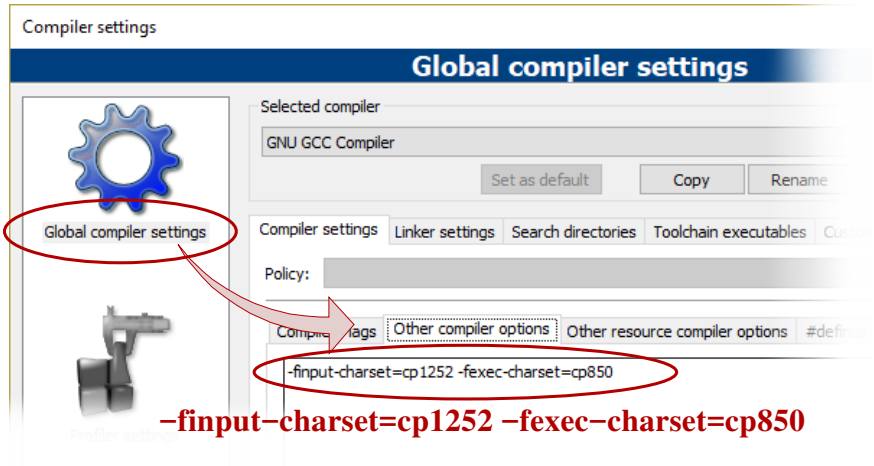

Figura 1.22
Opciones de codificación del compilador.

Ejercicio 1.9 Modifique el proyecto *Hola* para que el mensaje esté escrito en español, sin olvidar el carácter '¡' de apertura de exclamación. Compruebe el resultado sin las opciones de compilación y con ellas.

Finalmente, podemos responder de una forma más precisa a la siguiente pregunta:

- ¿Qué codificación podemos usar como código fuente para el compilador?

La respuesta es: cualquier codificación compatible con la tabla *ASCII*. Las tablas *ISO-8859-15*, *Windows-1252*, *CP850*, o incluso *UTF-8* son compatibles. Eso sí, recuerde que si genera una serie de caracteres con una codificación, el programa que lo recibe deberá saber procesarla.

Observe que hemos dicho compatible con *ASCII*, es decir, los primeros 128 caracteres comunes. Los archivos internos del compilador son compatibles con esta codificación. Si decide, por ejemplo, usar la tabla *UTF16* —que es la codificación que internamente usa *Windows*— para poder internacionalizar todos sus programas a lenguajes tan distantes como el chino, los fuentes del compilador que estamos usando no se corresponden con esta codificación. Así que si sabe chino, no, no lo use.

Entonces, ¿qué hacer para este curso?

En primer lugar hay que decir que no es de especial interés para el curso resolver problemas de codificaciones. Es algo técnico que no aporta demasiado sobre los fundamentos. Podemos hacer los programas como si escribiéramos en inglés o incluso ignorar los caracteres extraños que afectan estéticamente al resultado. Si tiene que hacer un ejercicio para probar algún aspecto e ignorar la codificación, no se preocupe.

Por otro lado, si realmente quiere tener cuidado con los mensajes, los acentos, los caracteres especiales del español, puede considerar añadir la configuración indicada. En concreto:

- Si va a trabajar con *Microsoft Windows*, puede incluir las opciones del compilador para que siempre trabajemos con las codificaciones *CP1252* —para el editor— y *CP850* para la consola (o las páginas de códigos configuradas en su sistema).
- Sin embargo, tal vez le interese intercambiar archivos con otros sistemas, por ejemplo, *GNU/Linux*. En éste se utiliza *UTF-8*. Puede seleccionar que todos sus archivos fuente sean siempre de esta codificación y, por otro lado, añadir solamente la segunda opción[10] en las opciones del compilador (figura 1.22) para obtener *CP850*.

En cualquier caso, no olvide que la consola deberá ser compatible con la codificación con la que trabaja el ejecutable. Si edita con *UTF-8* y usa una consola con *CP850*, no podrá representar todos los caracteres.

Por ejemplo, en *GNU/Linux* tengo la configuración por defecto, tanto editor como consola en *UTF8*. En estas condiciones, puede escribir y ejecutar el programa que se muestra en la figura 1.23.

Figura 1.23
Ejecución del programa *hola* con *UTF-8* en *GNU/Linux*.

Si ejecuto este programa e intento escribir en una consola con la página 850, no es posible representar el chino o el hebreo.

1.6 Problemas

Problema 1.1 Escriba un programa que lea el valor de un ángulo en grados y escriba en la salida estándar el número de radianes equivalente.

Problema 1.2 Escriba un programa que lea los dos valores *a,b* que determinan la ecuación $ax+b=0$ y escriba la solución de dicha ecuación. Pruebe la ejecución con valores que incluyan el cero, e incluso cuando los dos valores valen cero. Cambie las variables al tipo `int` y compruebe de nuevo los mismos ejemplos.

[10] Sin la primera, por defecto, la entrada es *UTF-8*.

A Tablas

Tabla ASCII 19
Operadores C++ 20
Palabras reservadas de C y C++ 21

A.1 Tabla ASCII

En la figura A.1 se presenta la tabla de codificación *ISO-8859-15* del alfabeto latino. Es similar a la ISO-8859-1 aunque difiere en 8 posiciones (*0xA4, 0xA6, 0xA8, 0xB4, 0xB8, 0xBC, 0xBD* y *0xBE*). Esta codificación se distingue especialmente porque incluye el carácter correspondiente al euro.

Aunque es probable que la codificación *ISO-8859-15* no sea la que esté usando en su sistema, la mayoría de los problemas que se resuelven en los cursos de programación asumen que es la codificación para los caracteres o secuencias de caracteres.

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1x	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2x	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL
8x	PAD	HOP	BPH	NBH	IND	NEL	SSA	ESA	HTS	HTJ	VTS	PLD	PLU	RI	SS2	SS3
9x	DCS	PU1	PU2	STS	CCH	MW	SPA	EPA	SOS	SGCI	SCI	CSI	ST	OSC	PM	APC
Ax	NBSP	¡	¢	£	€	¥	Š	§	š	©	ª	«	¬	SHY	®	¯
Bx	°	±	²	³	Ž	µ	¶	·	ž	¹	º	»	Œ	œ	Ÿ	¿
Cx	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
Dx	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
Ex	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
Fx	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Figura A.1
Tabla de codificación *ISO-8859-15*.

Observe que todos los caracteres especiales que existen en distintos idiomas y no en inglés están en la segunda parte de la tabla. Realmente, la tabla *ASCII* propiamente dicha es la primera mitad, mientras que la segunda es una extensión. Por ello, esta tabla no es la tabla *ASCII*, sino la tabla *ASCII extendida ISO-8859-15*.

En la práctica aceptaremos esta codificación para nuestras soluciones, ya que nos interesa especialmente el algoritmo a resolver sin entrar en detalles sobre los problemas de codificación. Note que podría tener problemas en la ejecución de algoritmos en español incluso si su sistema usa esta codificación. Por ejemplo, si quiere comprobar el orden de dos letras para ordenar una cadena, los caracteres con tilde están fuera del rango del alfabeto `'a'-'z'`.

Si su sistema usa otras codificaciones como *ISO-8859-1*, *Windows-1252*, o *UTF-8*, no tendrá problemas en ejecutar y comprobar el funcionamiento de sus algoritmos si evita el uso de la parte extendida, ya que en esas codificaciones los caracteres básicos se representan exactamente igual. Podríamos decir que realizamos soluciones que funcionan sin ningún problema solamente en inglés.

Más adelante es posible que tenga que resolver problemas para distintos lenguajes. Tendrá que consultar el tipo de codificación de su sistema y qué posibilidades tiene para resolverlo. Las soluciones pueden ir desde un tipo `char` con otra codificación hasta usar bibliotecas de funciones que resuelven sus problemas configurando el lenguaje usado.

A.2 Operadores C++

En la siguiente tabla se listan los operadores de C++. Los operadores situados en el mismo recuadro tienen la misma precedencia.

En general, para no tener ningún problema con la asociatividad, intente recordar que los operadores unarios y de asignación son asociativos por la derecha, mientras que los demás lo son por la izquierda.

Si revisa el estándar, podrá encontrar que realmente los unarios postfijo son de izquierda a derecha así como el operador condicional, aunque en la práctica no suelen crear incertidumbre.

Operadores de C++

Operador	*Nombre*	*Uso*
`::`	Global	::*nombre*
`::`	Resolución de ámbito	*espacio_nombres*::*miembro*
`::`	Resolución de ámbito	*nombre_clase*::*miembro*
`->`	selección de miembro	*puntero*`->`*miembro*
`.`	Selección de miembro	*objeto*`.`*miembro*
`[]`	Índice de vector	*nombre_vector*`[`*expr*`]`
`()`	Llamada a función	*nombre_función*`(`*lista_expr*`)`
`()`	Construcción de valor	*tipo(lista_expr)*
`++`	Post-incremento	*valor_i*`++`
`--`	Post-decremento	*valor_i*`--`
`typeid`	Identificador de tipo	typeid(*type*)
`typeid`	... en tiempo de ejecución	typeid(*expr*)
`dynamic_cast`	conversión en ejecución	dynamic_cast<*tipo*>(*expr*)
	con verificación	
`static_cast`	conversión en compilación	static_cast<*tipo*>(*expr*)
	con verificación	
`reinterpret_cast`	conversión sin verificación	reinterpret_cast<*tipo*>(*expr*)
`const_cast`	conversión const	const_cast<*tipo*>(*expr*)
`sizeof`	Tamaño del tipo	sizeof(*tipo*)
`sizeof`	Tamaño del objeto	sizeof *expr*
`++`	Pre-incremento	`++`*valor_i*
`--`	Pre-decremento	`--`*valor_i*
`~`	Complemento	`~`*expr*
`!`	No	`!`*expr*
`+`	Más unario	`+`*expr*
`-`	Menos unario	`-`*expr*
`&`	Dirección de	`&`*valor_i*
`*`	Desreferencia	`*`*expr*
`new`	reserva	`new` *tipo*
`new`	reserva e iniciación	`new` *tipo (lista_expr)*
`new`	reserva (emplazamiento)	`new` *(lista_expr) tipo*
`new`	reserva (con inicialización)	`new` *(lista_expr)tipo(lista_expr)*
`delete`	destrucción (liberación)	`delete` *puntero*
`delete[]`	... de un vector	`delete[]` *puntero*
`()`	Conversión de tipo	`(`*tipo*`)` *expr*

continúa en la página siguiente

continúa de la página anterior

Operador	*Nombre*	*Uso*
`.*`	Selección de miembro	*objeto*`.*`*puntero_a_miembro*
`->*`	Selección de miembro	*puntero*`->*`*puntero_a_miembro*
`*`	Multiplicación	*expr*`*`*expr*
`/`	División	*expr*`/`*expr*
`%`	Módulo	*expr*`%`*expr*
`+`	Suma	*expr*`+`*expr*
`-`	Resta	*expr*`-`*expr*
`<<`	Desplazamiento a izquierda	*expr*`<<`*expr*
`>>`	Desplazamiento a derecha	*expr*`>>`*expr*
`<`	Menor	*expr*`<`*expr*
`<=`	Menor o igual	*expr*`<=`*expr*
`>`	Mayor	*expr*`>`*expr*
`>=`	Mayor o igual	*expr*`>=`*expr*
`==`	Igual	*expr*`==`*expr*
`!=`	No igual	*expr*`!=`*expr*
`&`	Y a nivel de bit	*expr*`&`*expr*
`^`	O exclusivo a nivel de bit	*expr*`^`*expr*
`|`	O a nivel de bit	*expr*`|`*expr*
`&&`	Y lógico	*expr*`&&`*expr*
`||`	O lógico	*expr*`||`*expr*
`?:`	expresión condicional	*expr*`?`*expr*`:`*expr*
`=`	Asignación	*valor_i*`=`*expr*
`*=`	Multiplicación y asignación	*valor_i*`*=`*expr*
`/=`	División y asignación	*valor_i*`/=`*expr*
`%=`	Resto y asignación	*valor_i*`%=`*expr*
`+=`	Suma y asignación	*valor_i*`+=`*expr*
`-=`	Resta y asignación	*valor_i*`-=`*expr*
`<<=`	Desplazar izq. y asignación	*valor_i*`<<=`*expr*
`>>=`	Desplazar der. y asignación	*valor_i*`>>=`*expr*
`&=`	Y y asignación	*valor_i*`&=`*expr*
`^=`	O exclusivo y asignación	*valor_i*`^=`*expr*
`|=`	O y asignación	*valor_i*`|=`*expr*
`throw`	Lanzamiento de excepción	`throw` *expr*
`,`	Coma	*expr* `,`*expr*

Tabla A.1
Operadores de C++

Estos operadores están incluidos en el estándar C++98. En C++11 aparecen tres más:

`sizeof...`, **`noexcept`**, **`alignof`**

A.3 Palabras reservadas de C y C++

Es interesante conocer las partes comunes del lenguaje C y C++. En esta sección presentamos la tabla A.2 con las palabras reservadas de las distintas versiones de ambos lenguajes. La parte más interesante se encuentra en los dos primeros bloques, donde se incluyen las palabras reservadas que estaban presentes en C cuando se creó C++. Dado que éste "heredó" gran parte de los contenidos de C, el segundo bloque de C++ se presenta como una adición a las del primero (comunes a ambos).

A pesar de ello, los últimos tres bloques presentan palabras que se han ido añadiendo en las sucesivas versiones de los lenguajes. Aunque todavía muchos programadores creen que C++ es un lenguaje ampliación de C, lo cierto es que los dos evolucionan de forma independiente.

Tabla A.2
Palabras reservadas de C y C++.

Comunes a C(C89) y C++			
auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
int	long	register	return
short	signed	sizeof	static
struct	switch	typedef	union
unsigned	void	volatile	while
Adicionales de C++			
and	and_eq	asm	bitand
bitor	bool	catch	class
compl	const_cast	delete	dynamic_cast
explicit	export	false	friend
inline	mutable	namespace	new
not	not_eq	operator	or
or_eq	private	protected	public
reinterpret_cast	static_cast	template	this
throw	true	try	typeid
typename	using	virtual	wchar_t
xor	xor_eq		
Añadidas en C99			
_Bool	_Complex	_Imaginary	inline
restrict			
Añadidas en C++11			
alignas	alignof	char16_t	char32_t
constexpr	decltype	noexcept	nullptr
static_assert	thread_local		
Añadidas en C11			
_Alignas	_Alignof	_Atomic	_Generic
_Noreturn	_Static_assert	_Thread_local	
Añadidas en C++20			
char8_t	concept	consteval	constinit
co_await	co_return	co_yield	requires

Bibliografía

Referencia principal

[1] Garrido, A. *Fundamentos de programación con la STL*. Editorial Universidad de Granada, 2016.

[2] A. Garrido, J.Martínez-Baena. *Introducción a la programación C++. Ejercicios.*. Editorial Universidad de Granada, 2016.

[3] A. Garrido. *Prácticas con C++. Metodología de la programación.*. Segunda edición. Editorial Universidad de Granada, 2017.

Referencias secundarias

[4] Garrido, A. *Fundamentos de programación en C++*. Delta publicaciones, 2005.

[5] Garrido, A. *Metodología de la Programación: de bits a objetos*. Editorial Universidad de Granada, 2016.

[6] Garrido, A. *Programación Genérica en C++: la biblioteca estándar*. Editorial Universidad de Granada, 2017.

[7] Deitel y Deitel, *C++11 for programmers* . Segunda Edición. Prentice Hall. 2013.

[8] B. Stroustrup, *El lenguaje de programación C++. Edición Especial*. Addison-Wesley, 2002.

[9] B. Stroustrup, *The C++ Programming Language, 4th Edition*. Addison-Wesley, 2013.

[10] B. Stroustrup, *The design and Evolution of C++*. Addison-Wesley, 1994.

Referencias electrónicas

[11] *C++ Reference*. Página web con una referencia bastante completa de los recursos disponibles en el lenguaje C++. `http://www.cplusplus.com/reference/`

[12] *C++ Reference*. Página web con una referencia bastante completa de los recursos disponibles en el lenguaje C++, incluyendo explicaciones de las aportaciones del último estándar. `http://en.cppreference.com/`

[13] B. Stroustrup, *C++ Style and Technique FAQ*. http://www.research.att.com/~bs/bs_faq2.html.

Índice alfabético

A
análisis léxigo, 9
análisis semántico, 9
análisis sintáctico, 9
ASCII, 15, 19
avisos de compilación, 11

B
bin, directorio, 8

C
C++14
    seleccionar, 12
cbp, fichero de proyecto, 6
Code::Blocks, 1
codificación, 15
compilación, 6
    avisos de, 11
    errores de, 8
    flags de, 12
consola
    proyecto de, 3
CP1252, 16
CP850, 17

D
debug, directorio, 8

E
ejecución, 6
enlace, 6
estilo de codificación, 14

F
flags de compilación, 12

I
IDE, 1
ISO8859, 15, 19

M
MinGw, 2

O
obj, directorio, 8
objeto
    archivos, 8
operadores C++, 20

P
palabras reservadas, 21
proyecto, de Code::Blocks, 3

R
reservadas
    palabras, 21

S
sangrado, 14

T
tokens, 9
traducción, 9

U
UNICODE, 16
UTF-8, 16, 19

W
warnings, compilación, 11
Windows-1252, 16, 19