Guión 1

Depuración con Code::Blocks
Noviembre, 2012

Fundamentos de Programación

Índice

1. Definición del problema

El objetivo de este guión de prácticas consiste en introducir al alumno en el uso de técnicas de análisis del código del programa, de forma que se comprenda su funcionamiento y puedan corregirse errores lógicos en casos de haberlos (es decir, se asume que el código ha compilado y ejecuta, aunque no produce los resultados deseados).

Hasta este momento, la forma de resolver estos problemas consistía en ir introduciendo trazas (mensajes por pantalla, mediante **cout**, que mostraban el valor de las variables de interés; la información necesaria sobre su funcionamiento). Sin embargo, esta forma de proceder obliga a escribir código que luego habrá que eliminar cuando se disponga de la versión definitiva.

Este proceso de análisis del programa se denomina **depuración**. La mayoría de los entornos integrados de programación ofrecen facilidades para realizar esta tarea. Las herramientas que incluyen para ello se denominan **depuradores**. Mediante los depuradores es posible detener la ejecución del programa en cualquier instante, de forma que se puedan conocer los valores de las variables relevantes, cuál es la secuencia de llamadas a funciones, y continuar la ejecución cuando se desee. De modo informal, se trata de ejecutar el programa bajo lupa, observando detenidamente cómo evoluciona, para comprender así qué posibles problemas hacen que los resultados obtenidos no sean los deseados.

Aunque este guión está centrado en el depurador de **Code::Blocks**, las ideas aquí expuestas son válidas para los depuradores de cualquier otro entorno de programación. El seguimiento de este guión no precisa de más conocimientos que los adquiridos en las clases de teoría y prácticas de la asignatura. Aquí se presenta la forma de depurar un pequeño programa que consta de una función auxiliar y la función **main**, con una llamada desde esta última a la primera.

Tras el estudio de este guión el alumno debería ser capaz de:

1. realizar de forma autónoma la depuración de cualquier programa
2. comprender el funcionamiento del depurador de otras herramientas de programación
3. entender de forma completa el modo en que se ejecutan las sentencias en estructuras de control y en llamadas a funciones

2. Creación de proyectos con información de depuración

Los pasos necesarios para crear un proyecto nuevo (recordad siempre que la necesidad de crear un proyecto es algo propio de la herramienta y no del lenguaje de programación; así Code::Blocks organiza todos los recursos del programa) son:

- Menú **File**, opción **New**, **Project** (ver Fig. 1)

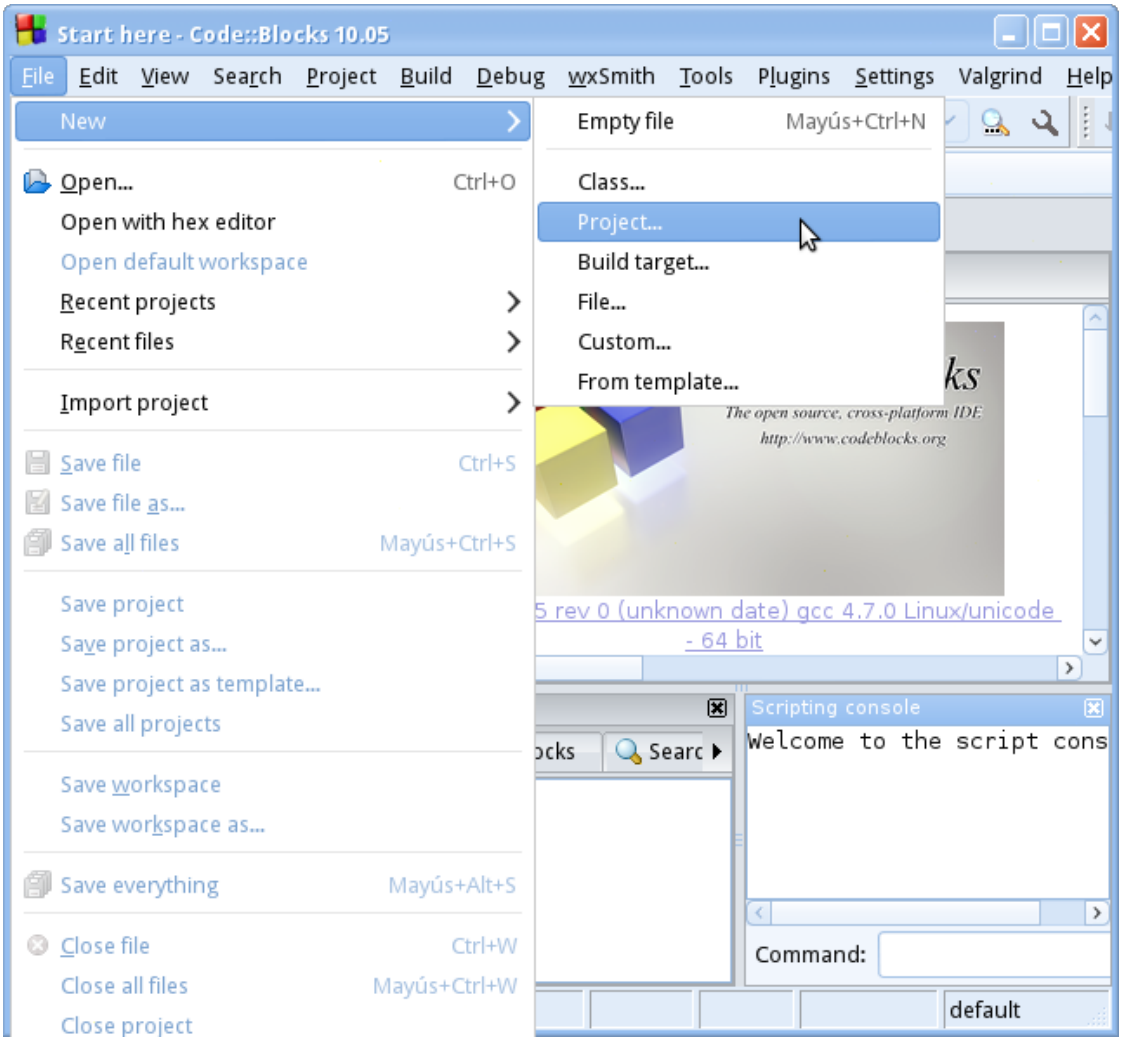

Figura 1: Paso 1

- Esto hace aparecer una nueva ventana en que podemos seleccionar el tipo de proyecto a elegir (nosotros usaremos **Console application**, ver Fig. 2).

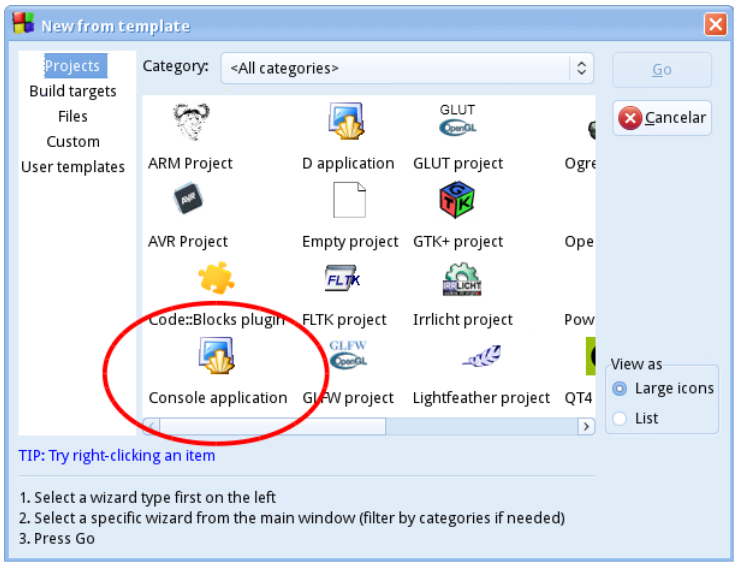

Figura 2: Paso 2: tipos de proyecto

- Seleccionamos **Console application** y pulsamos el botón **Go** (ver Fig. 3).

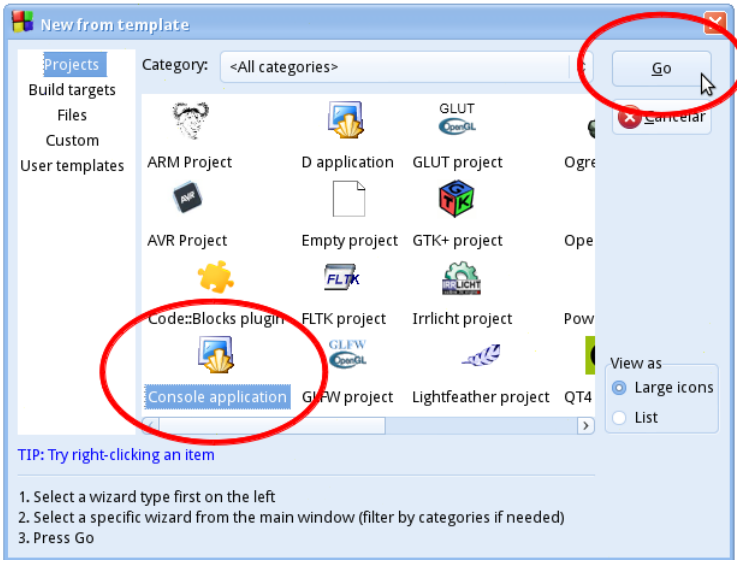

Figura 3: Paso 3: selección de tipo de proyecto

- Tras seleccionar el tipo de proyecto aparece una nueva ventana en la que se indica el lenguaje de programación a usar (C o C++; elegiremos este último), Fig. 4.

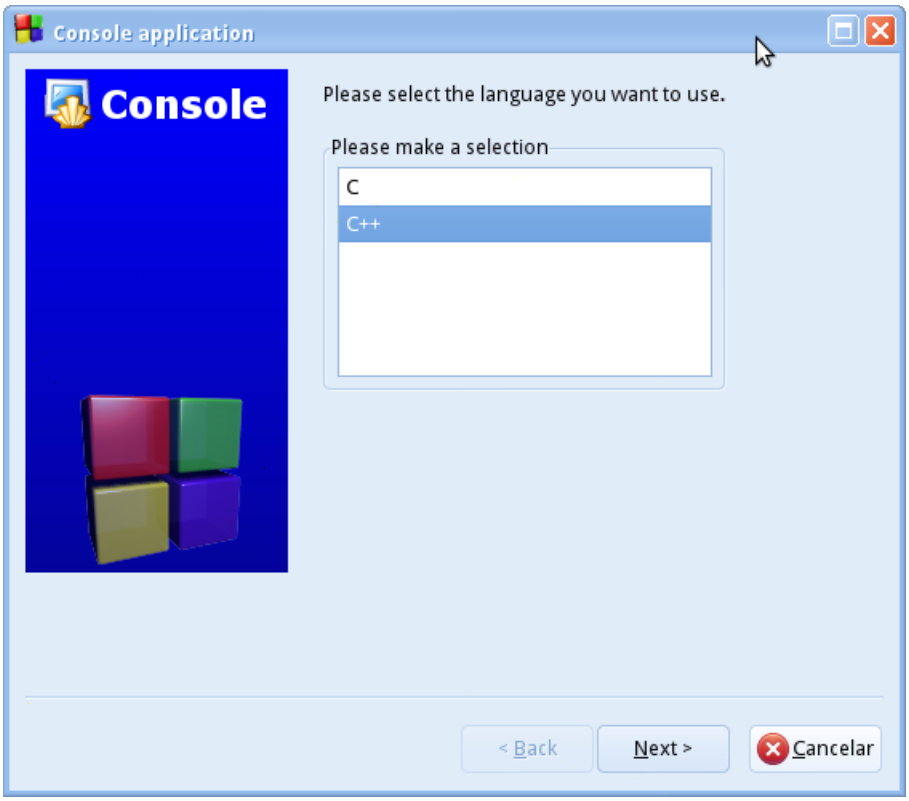

Figura 4: Paso 4: selección de C++

- Al pulsar en **Next** en la ventana del paso anterior, aparecerá otra ventana en que debemos rellenar los datos sobre el proyecto: título, directorio donde se creará el proyecto, etc. (Fig. 5). Se completa esta formulario y se pulsa **Next**.

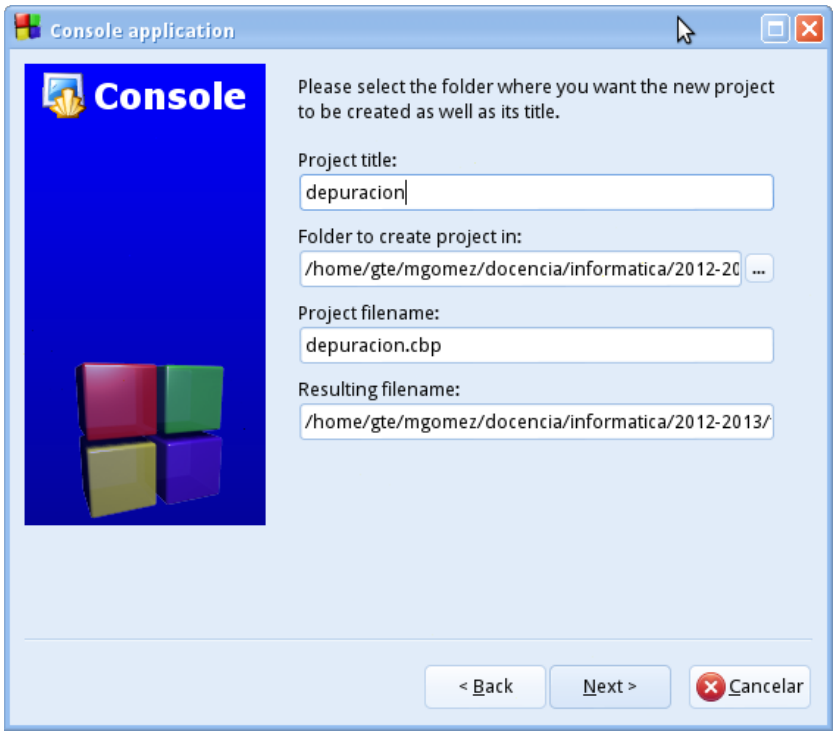

Figura 5: Paso 5: formulario de título y ubicación del proyecto

- El paso final consiste en especificar el compilador a usar y qué configuraciones desean crearse, Fig. 6. Se observa que, por defecto, la ventana muestra activas las opciones de creación de las configuraciones **Debug** y **Release**. La primera de ellas permite realizar el proceso de depuración que deseamos. Para permitir al usuario examinar bajo lupa el programa el compilador necesita introducir información extra en el ejecutable. Esta información adicional no resulta necesaria cuando la aplicación esté lista. Si fuéramos a distribuir la versión final de un programa sólo estaríamos interesados en conseguir la configuración **Release**. Como por defecto se crea la configuración de depuración, dejamos todo tal y como está. Al final se pulsa en **Finish** y **Code::Blocks** crea los elementos necesarios para poder empezar a trabajar con el proyecto.

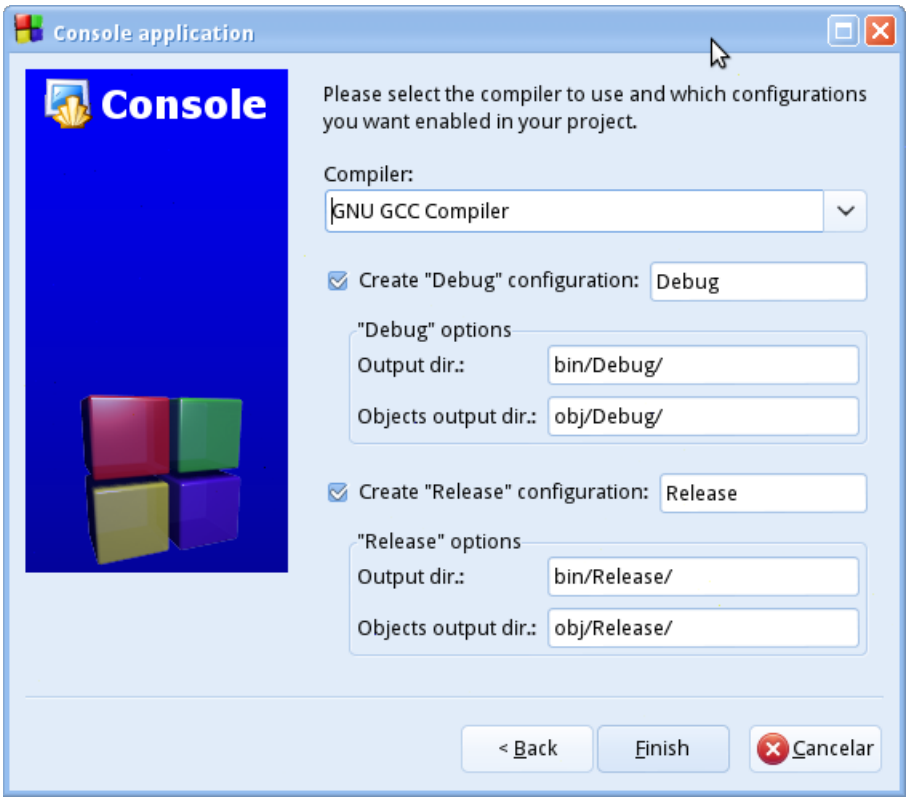

Figura 6: Paso 6: configuraciones disponibles

El código que analizaremos será el del ejercicio 22 de la relación de bucles, agregando una función que se encargará de mostrar los resultados que se vayan obteniendo.

```
#include <iostream>

using namespace std;

/**
  * Metodo para presentar por pantalla los pares de valores
  * que componen la secuencia RLE
  * @param repeticiones
  * @param valor
  * @param fin valor booleano que indica si el programa va a
  *        finalizar
  */
void presentarResultados(int repeticiones, int valor, bool fin){
    cout << valor << " - " << valor << endl;

    // Si es el fin se indica con una linea de asteriscos
    if (fin){
        cout << "****************************************" << endl;
    }
}

int main()
{
  int valorActual, valorAnterior=-1, contadorRepeticiones=0;

  // Bucle de lectura de datos
  do{
    // Se pide al usuario que introduzca el dato
    cout << "Introduzca dato: ";
    cin >> valorActual;

    // Se comprueba si el valor actual es igual que el previo
    if (valorActual != valorAnterior){
      // Hay cambio de secuencia y se trata
      // Se muestra el par contador - valorAnterior
      presentarResultados(contadorRepeticiones,valorAnterior,
                          (valorActual == -1));

      // Se resetea el valorAnterior y el contador
      contadorRepeticiones=1;
      valorAnterior=valorActual;
    }
    else{
      // Se acumula el contador
      contadorRepeticiones++;
    }
  } while(valorActual != -1);
}
```

Para comprobar qué se genera al compilar este código, examinamos la carpeta donde hemos ubicado el proyecto. Observad que en la barra de menú aparece la opción **Build target** que permite seleccionar entre **Debug** y **Release**, las dos configuraciones admitidas en el proyecto. El valor seleccionado por defecto es **Debug** (ver Fig. 7). Se compila de forma normal (para generar la versión de depuración).

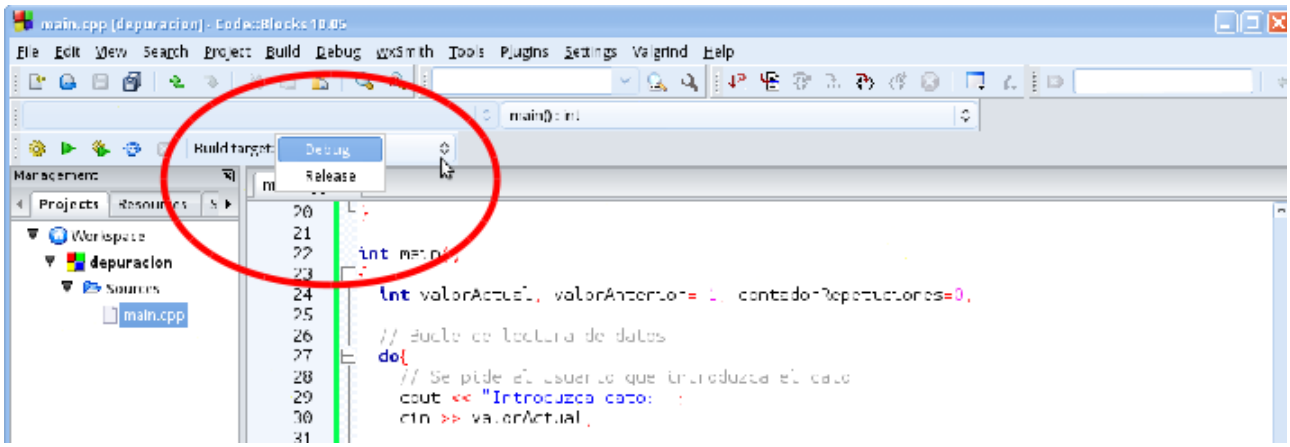

Figura 7: Selección de configuración a crear

El contenido de la carpeta del proyecto se muestra en la Fig. 8.

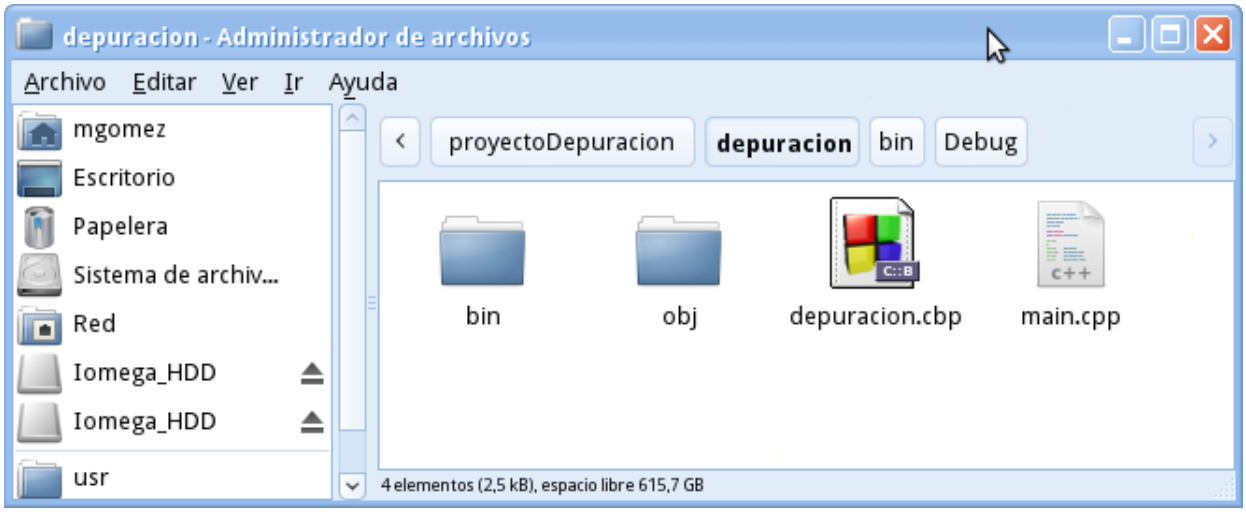

Figura 8: Contenido de la carpeta del proyecto

Los ejecutables se encuentran almacenados en la carpeta **bin** (ver contenido en Fig. 9).

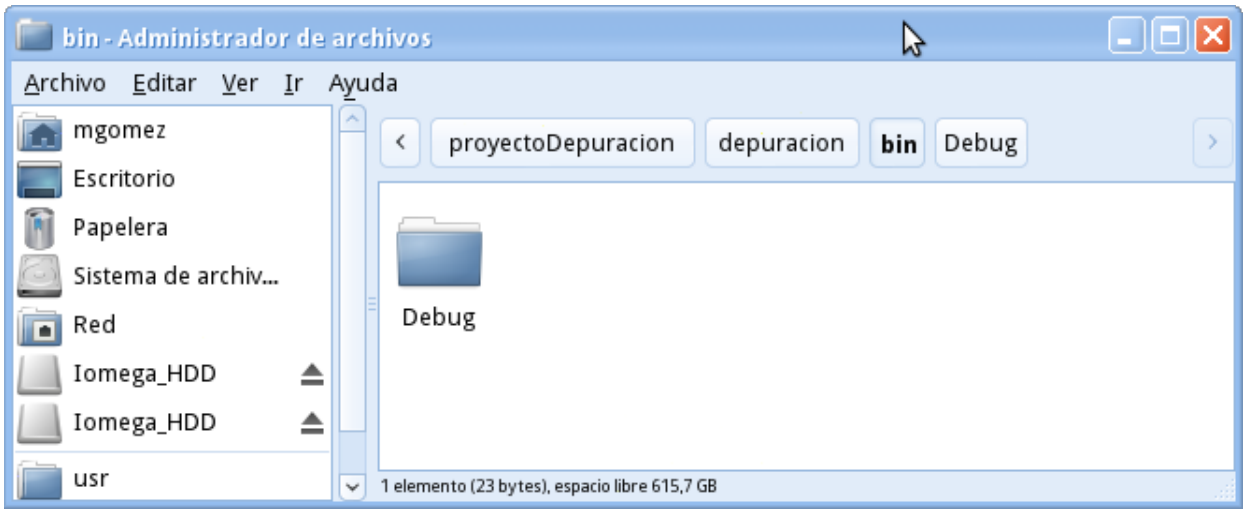

Figura 9: Contenido de la carpeta bin

Se aprecia que el directorio **bin** contiene una carpeta específica para cada configuración generada (de momento sólo **Debug**). Esta carpeta contiene únicamente el programa ejecutable generado, Fig. 10.

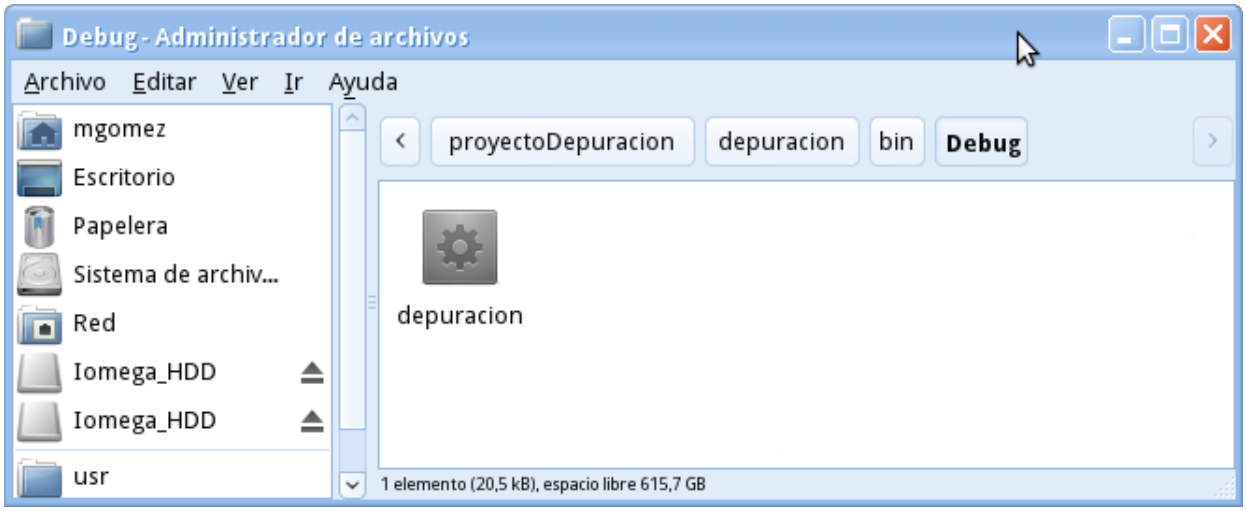

Figura 10: Contenido de la carpeta Debug: ejecutable para depuración

Si selecciono ahora en **Build target** la versión final (**Release**), se creará otra carpeta para el ejecutable final; ahora **bin** contiene dos carpetas: **Debug** y **Release** (Fig. 11).

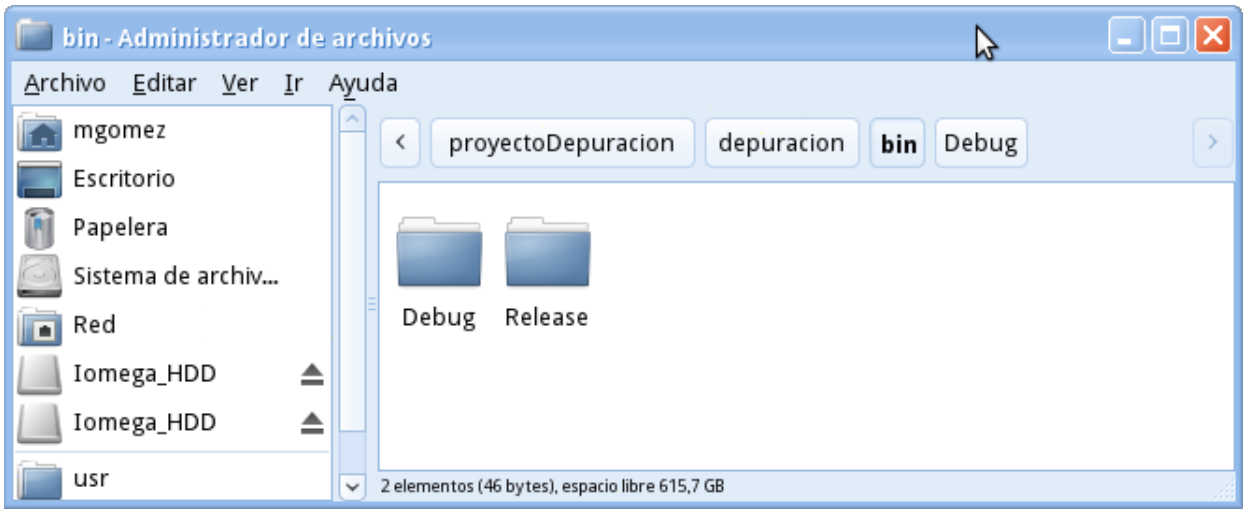

Figura 11: Contenido de la bin con las dos configuraciones

El trabajo del compilador, por tanto, difiere en cada una de las configuraciones anteriores. Su funcionamiento se modifica especificando un conjunto de opciones disponibles. Pueden verse las opciones usadas mostrando la información disponible sobre el proyecto, pinchando sobre el nombre del proyecto con el botón de la derecha y seleccionando **Build options...** (ver Fig. 12).

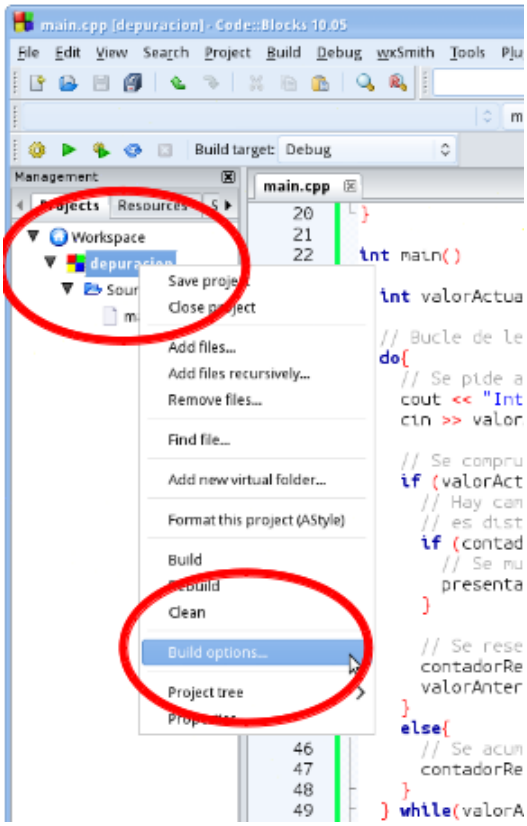

Figura 12: Visualizar las opciones de generación del proyecto

Al seleccionar esta opción aparece una ventana en que pueden consultarse (y configurarse las opciones usadas para ajustar el comportamiento del compilador). En concreto, se aprecia en la Fig. 13 que se encuentra activa la opción **-g**, que indica al compilador que debe generar información de depuración. Esta opción se desactiva. obviamente, en el modo de generación

correspondiente a la configuración **Release**.

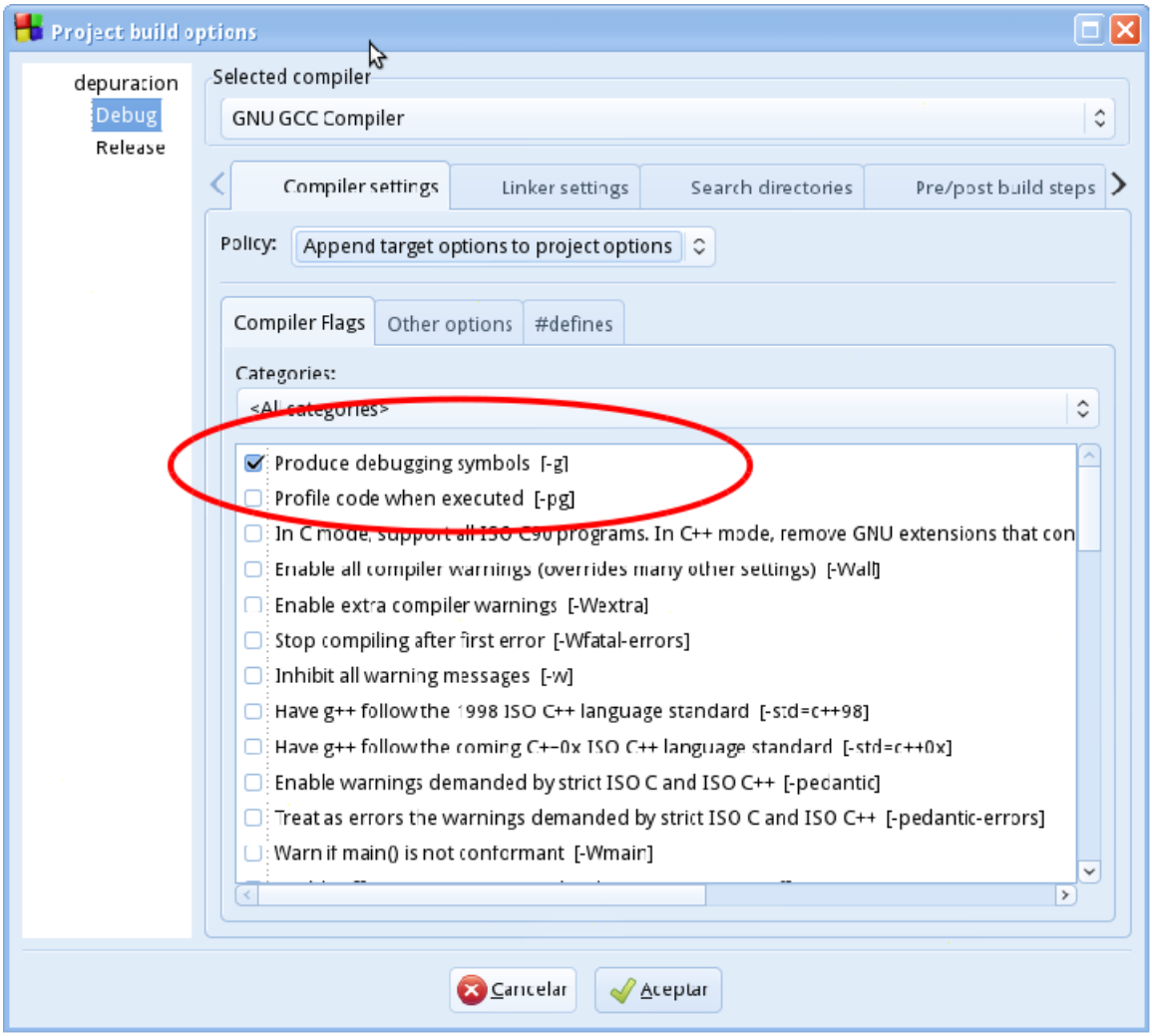

Figura 13: Opciones activadas en modo depuración

3. Perspectivas

Este entorno integrado de desarrollo permite adaptar su apariencia a la forma de trabajo en curso, activando o desactivando ventanas específicas para ciertas tareas. Para facilitar este cambio de configuración del entorno, dispone de varias **perspectivas** (configuraciones especiales de trabajo) que pueden seleccionarse desde la entrada **View** (barra principal de menú), opción **Perspectives**, ver Fig. 14.

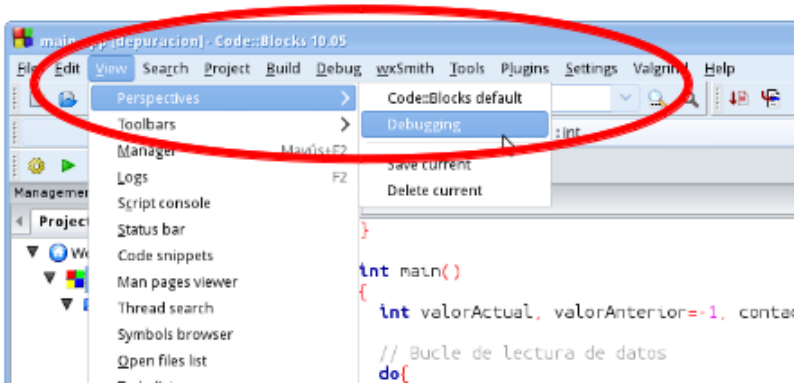

Figura 14: Selección de perspectiva

Si se selecciona la perspectiva de depuración (por defecto está activada la de desarrollo) cambiará la apariencia de la ventana de trabajo, tal y como se aprecia en la Fig. 15.

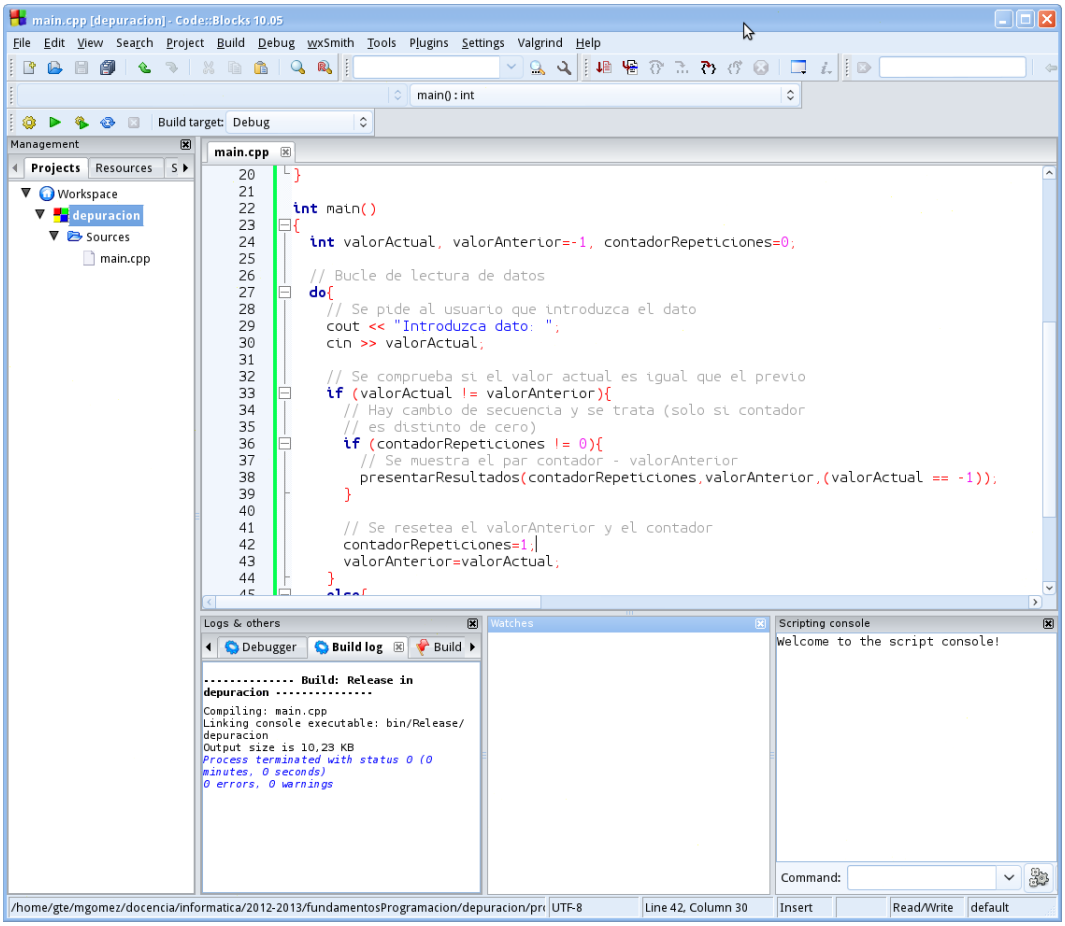

Figura 15: Ventana en modo de depuración

Ahora veremos por qué resulta interesante cambiar la perspectiva para proceder a la depuración del programa.

4. Funcionamiento del programa

Necesitaremos depurar un programa cuando no se comporte de forma adecuada. Recordemos que los errores lógicos son los más complicados de resolver: no tenemos ayuda ninguna del compilador (no hay errores de compilación) y el programa finaliza de forma aparentemente normal aunque no produce los resultados deseados.

En este programa el usuario irá introduciendo una serie de números y hay que detectar las secuencias de repetición de los valores. Cuando la secuencia finaliza hay que escribir por pantalla información sobre: número de repeticiones y valor. El proceso continúa hasta introducir un valor -1 (fin de operación). Por ejemplo, la secuencia $11138882215555-1$ debe producir como resultado (aunque en líneas diferentes, debido a la forma en que se introducen los valores con **cin**) será $3-1\ 1-3\ 3-8\ 2-2\ 1-1\ 4-5$.

Al ejecutar nuestro código observamos un comportamiento no deseado. En cuanto se introduce un valor aparece un mensaje por pantalla que no debería aparecer (ver Fig. 16).

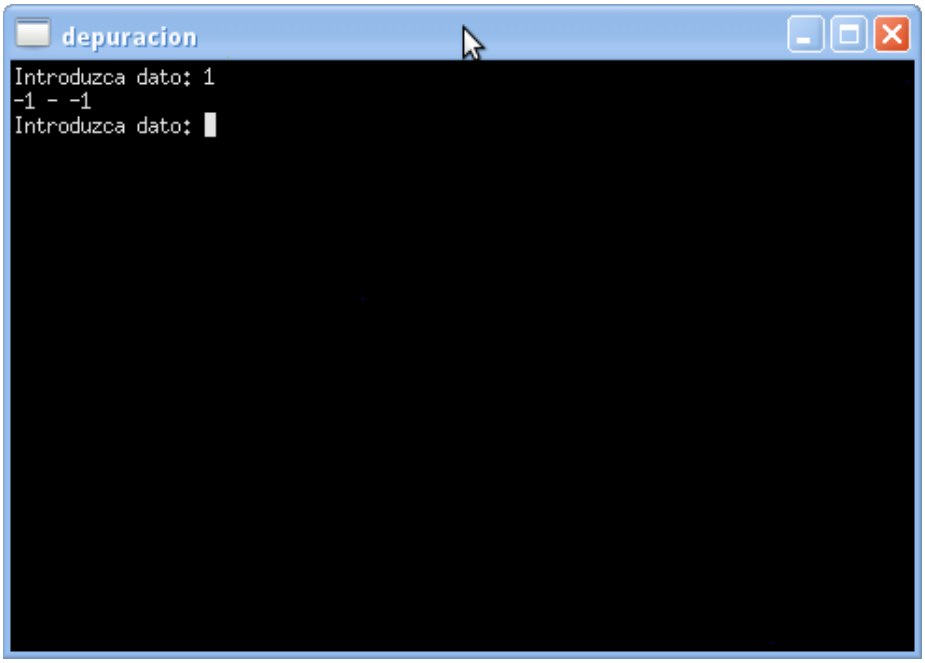

Figura 16: Error lógico del programa

¿Por qué aparece ese mensaje inicial si acabamos de introducir un valor 1? La información de secuencia debe aparecer cuando se cambie el valor tecleado. Si seguimos introduciendo 1's (2, por ejemplo) y luego se teclea 8 aparece debería aparecer **3 - 1** aludiendo a las tres repeticiones del valor 1, ver Fig. 17.

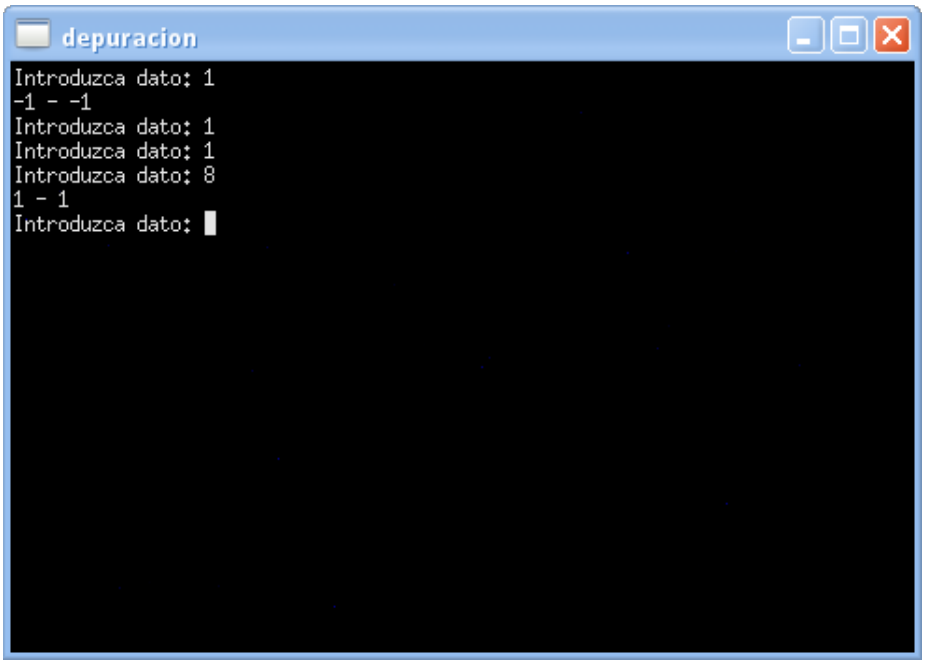

Figura 17: Error lógico del programa

Se aprecia que el programa sigue sin comportarse como deseamos. Siempre tenemos la posibilidad de escribir sentencias **cout** para mostrar por pantalla qué está ocurriendo, pero luego habrá que quitarlas al conseguir que el programa funcione de forma adecuada. Así que analizaremos el comportamiento del programa usando el depurador.

5. Opciones del depurador

El primer paso en el proceso de depuración consiste en marcar un punto inicial donde queremos que se detenga la ejecución del programa, de forma que podamos analizar qué valores tienen las variables. Todas las sentencias previas a este punto se ejecutarán como si el programa se hubiera lanzado en modo normal. En el ejemplo que estamos considerando no vale la pena detenernos en la declaración de variables ni en la lectura del valor (la lectura mediante **cin**, por sí misma, hará que la ejecución se detenga hasta que el usuario introduzca el valor solicitado.

De esta forma, la primera sentencia de interés es:

```
. . . . . .
// Se comprueba si el valor actual es igual que el previo
if (valorActual != valorAnterior){
. . . . . .
```

y haremos que la ejecución se detenga ahí. Así podremos comprobar cuáles son los valores de las variables **valorActual** y **valorAnterior**. Para conseguir esto necesitamos fijar un punto de ruptura (**breakpoint**) en la línea de interés. Hay varias formas de hacer esto:

- pulsando en la línea de interés con el botón de la derecha del ratón. Esto hace que aparezca un menú desplegable donde se selecciona la opción **Toggle breakpoint** (activar punto de ruptura), Fig.18.

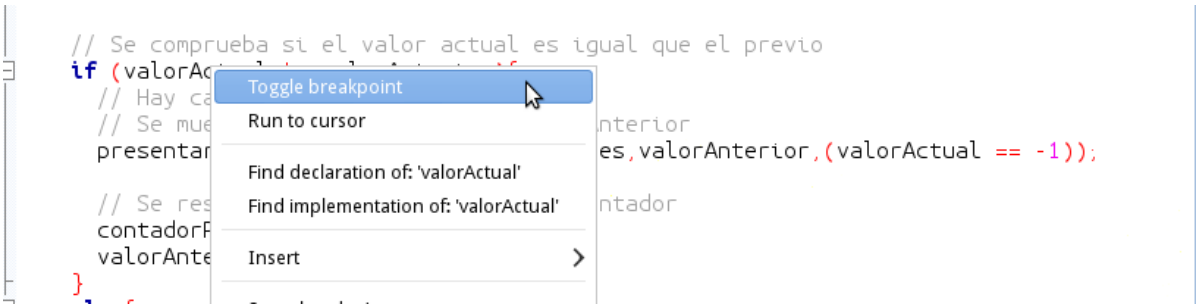

Figura 18: Punto de ruptura: mediante menú

- situándonos en el borde izquierdo de la sentencia, en la zona donde aparecen los números de línea, y pinchando con el botón izquierdo del ratón (ver la ubicación del ratón en la Fig. 19).

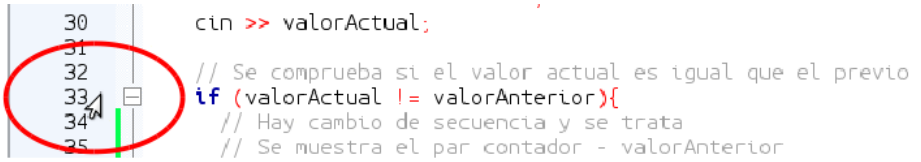

Figura 19: Punto de ruptura: de forma directa

En cualquier caso, en cuanto se ha realizado la operación **Code:.Blocks** deja una marca en la línea para indicar que el punto de ruptura ha sido fijado. Los puntos de ruptura son sentencias en que se detendrá la ejecución del programa (quedando como congelado) a espera de que el usuario del depurador decida qué hacer. cómo seguir. La indicación de punto de ruptura es un punto rojo (en otros entornos se usan otras marcas diferentes, pero siempre suelen aparecer como indicaciones sobre el borde o bien cambiando el color de fondo de la sentencia asociada a la línea), ver Fig. 20.

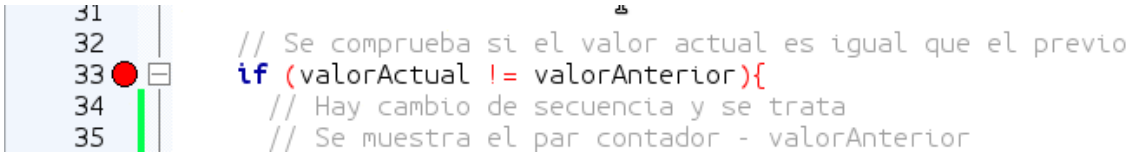

Figura 20: Marca de punto de ruptura

Ahora basta con ejecutar el programa, pero indicando que deseamos la ejecución mediante el depurador. Para ello seleccionamos la opción **Debug** del menú principal y en el menú desplegable escogemos **Start** (observad que en el menú se indica que esta opción también puede activarse con **F8**), ver Fig. 21.

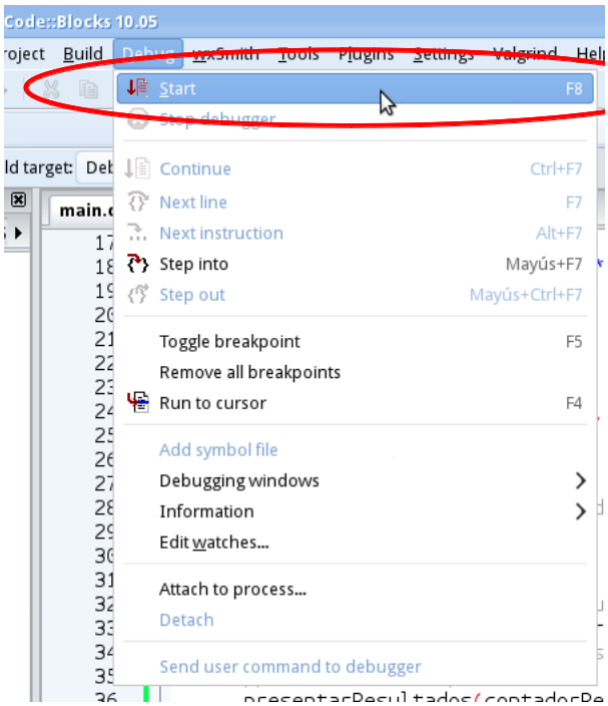

Figura 21: Inicio del proceso de depuración

El inicio del proceso de depuración implica que cambia la perspectiva de **Code::Blocks** a depuración y aparecerá la consola pidiendo al usuario que introduzca dato (recordad que las líneas donde se hace la escritura del mensaje y la correspondiente entrada están antes del punto de ruptura, de forma que se ejecutan de forma normal). En cuanto se introduce el valor da comienzo el proceso de depuración. El punto en que el programa está detenido será el de la línea donde se fijó el punto de ruptura. En todo momento se marca la próxima línea a ejecutar con un pequeño triángulo en color amarillo ubicado en la parte izquierda de la ventana de edición del código (ver Fig. 22).

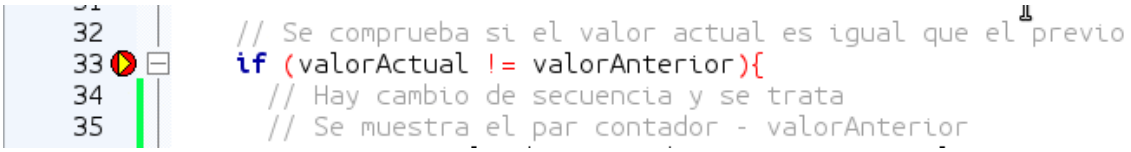

Figura 22: Marcado de la línea a ejecutar

Ahora se puede analizar el contenido de las variables usadas. De forma automática el depurador ofrece la posibilidad de inspeccionar las variables locales a la función en ejecución. Estas pueden verse en la ventana central de la zona inferior, encabezada con el título **Watches**, Fig. 23.

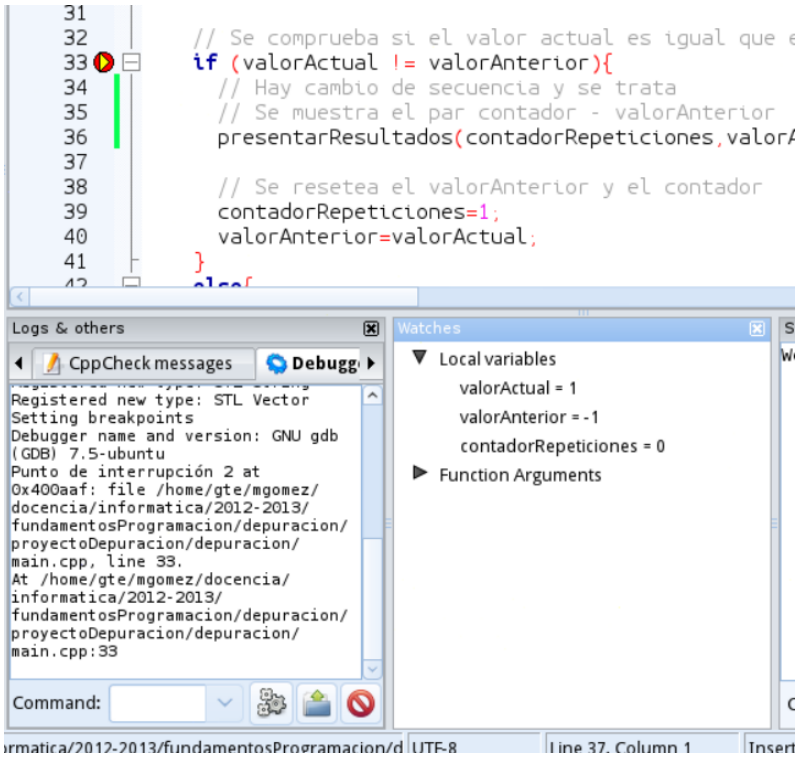

Figura 23: Visualización de variables locales

A partir de este momento las posibles acciones a realizar son las mostradas en un menú situado en la zona superior de la ventana de **Code::Blocks**, mostrado en la Fig. 24.

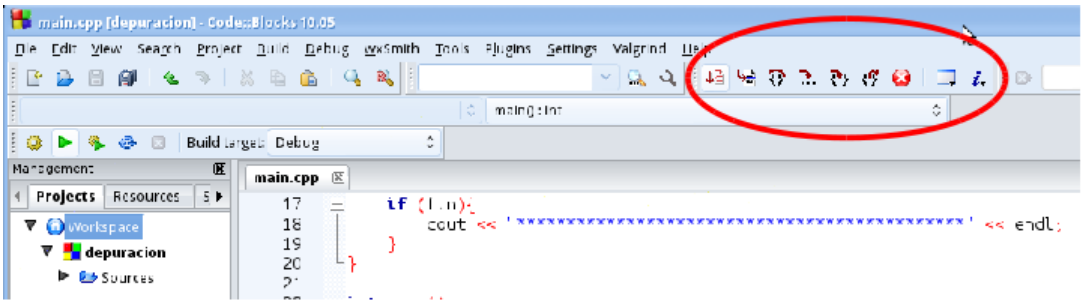

Figura 24: Menú para depuración

Si vamos pasando el ratón sobre las opciones, de izquierda a derecha, aparece el siguiente texto informativo:

- **Debug/continue**: permite continuar con la ejecución, que volvería a detenerse en caso de encontrarse con un punto de ruptura.
- **Run to cursor**: permite ejecutar de una vez todas aquellas sentencias hasta aquella en que esté situado el cursor. Antes de usar esta opción, obviamente, hemos de haber colocado el cursor en la posición donde deseemos se detenga el programa. Al llegar la ejecución a esa línea el programa se detendrá y el triángulo amarillo, marca de la próxima línea a ejecutar, estará a la izquierda de la línea del cursor, como se muestra en la Fig. 25 (asumiendo que habíamos situado el cursor en esta línea).

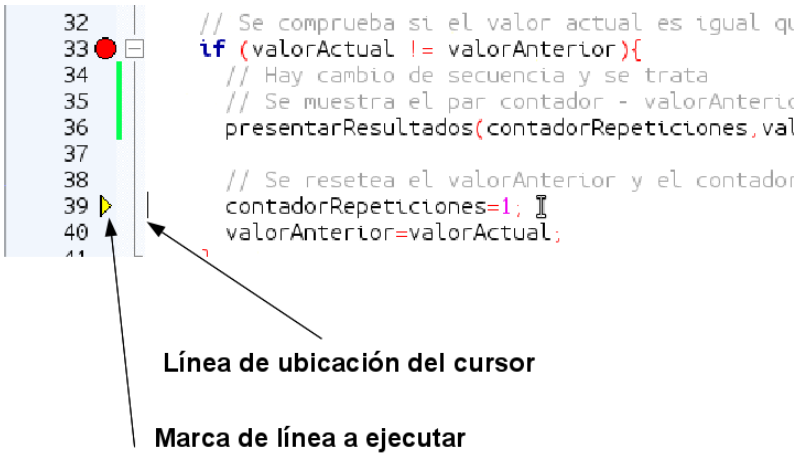

Figura 25: Avance hasta línea del cursor

- **Next line**: prosigue la ejecución avanzando hasta la siguiente línea.
- **Next instruction**: permite seguir la ejecución del código ensamblador asociado a la sentencia en ejecución (en una ventana auxiliar). Nosotros no llegaremos tan lejos.....
- **Step into**: opción (junto a la siguiente) relacionadas con la ejecución de llamadas a funciones. Imaginemos que la línea a ejecutar es la que se corresponde con la llamada a la función **presentarResultados**, tal y como podemos ver en la Fig. 26.

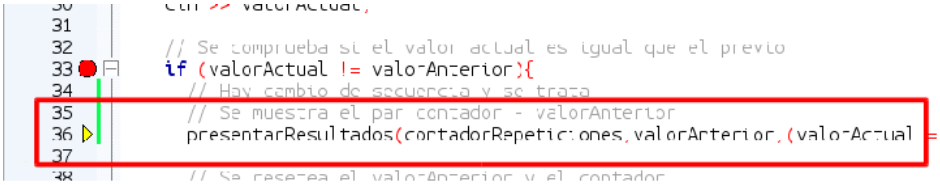

Figura 26: Ejecución de la función

Si deseamos ver paso a paso la ejecución de las sentencias de la función, usaremos esta opción, que indica que iremos paso a paso en el análisis del interior de la función. Eligiendo este modo la marca de ejecución en curso pasa a ser la primera de la función, como se muestra en la Fig. 27.

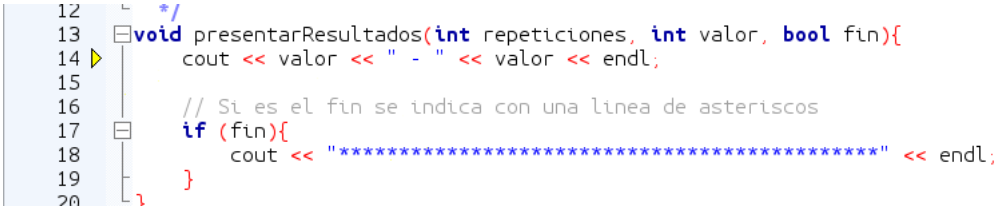

Figura 27: Inspección detallada de la función

Observamos que, una vez dentro de la función, también tenemos accesible, en la ventana **Watches**, los valores de los argumento usados en la llamada, ver Fig. 28.

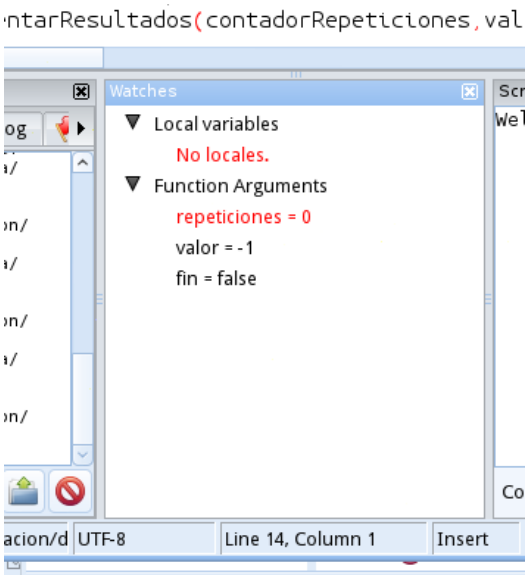

Figura 28: Valores de los argumentos

- **Step out**: si no estamos interesados en analizar el comportamiento interno de la función, ya que, por ejemplo, como ocurre en este caso, sólo muestra información por pantalla, usaremos esta alternativa, que ejecuta la llamada a la función como si se tratase de una única sentencia y se pasa a la línea siguiente. También puede usarse para saltar todas las instrucciones de la función de forma seguida y avanzar hasta la línea siguiente a aquella en que se produjo la llamada.
- **Stop debugger**: detiene la ejecución del programa (para seguir modificando el código o para ejecutar en modo normal).
- **Debugging windows**: permite mostrar diferente información sobre el estado de ejecución del programa. Al pulsar sobre ella aparece una lista desplegable con todas las opciones disponibles. Una muy interesante, **Call stack**, permite ver la pila de llamadas en el momento actual. Al pulsarla, estando la marca de próxima línea a ejecutar en una sentencia de la función, aparece una nueva ventana con la información recogida en la Fig. 29. Se indica que la cima de la pila contiene la información sobre la llamada a la función **presentarResultados**, estando la función **main** justo debajo.

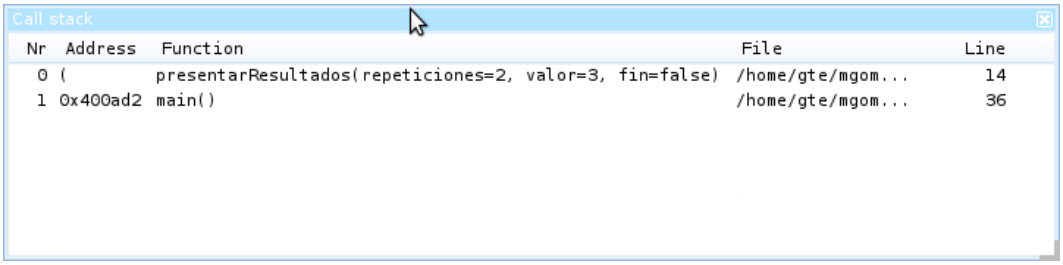

Figura 29: Pila de llamadas

Esta ventana aparece libre, sin vincular al entorno de desarrollo. Si queremos que esté visible de forma continua basta con arrastrar la ventana a la zona donde queramos anclarla (por ejemplo, a la zona de la izquierda). El resultado final se muestra en la Fig. 30.

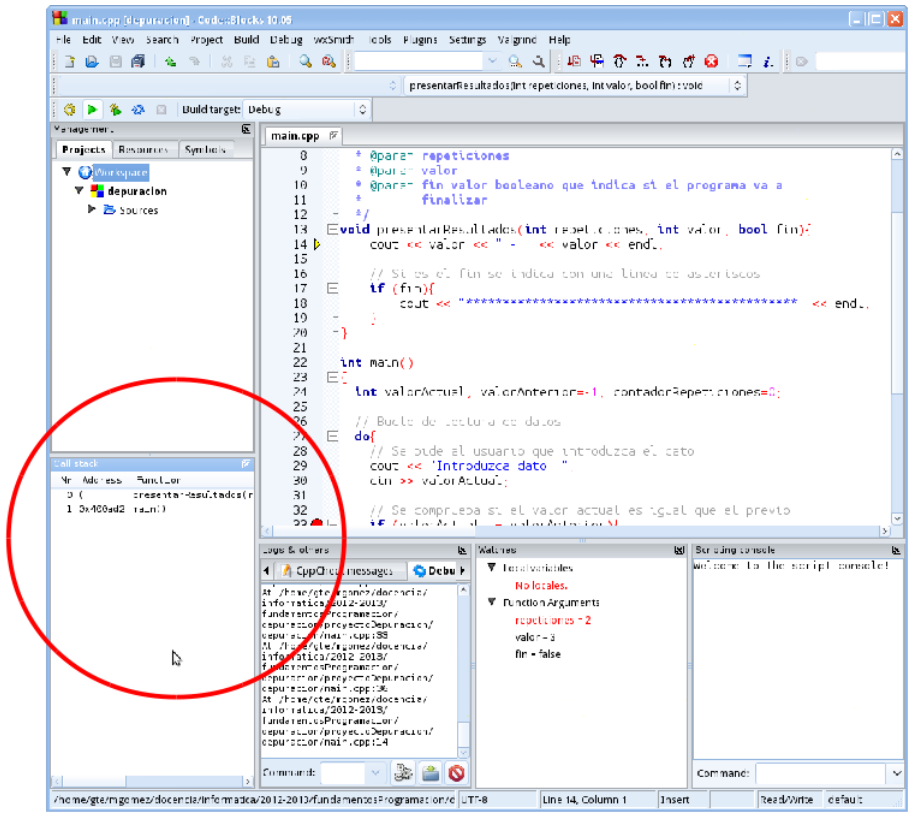

Figura 30: Integración de la ventana de pila de llamadas

- **Various info**: abre nuevas ventanas con información avanzada sobre la ejecución del programa.

6. Sesión de trabajo

Una vez conocido el funcionamiento básico del depurador nos centramos en la detección del error lógico del programa anterior. Para ello volvemos a atrás, al punto en que la próxima instrucción a ejecutar sea la estructura condicional que pregunta sobre la desigualdad entre el valor actual y el previo (tras haber introducido un valor, 1 por ejemplo, por teclado). La ventana **Watches** indica que:

```
valorActual=1
valorPrevio=-1
contadorRepeticiones=0
```

Como son diferentes valor actual y valor previo se cumple la condición y se entra dentro del bloque **if**. Esto se comprueba si hacemos avanzar la ejecución una línea, mediante el botón **Next line** del menú de depuración (ver Fig. 31).

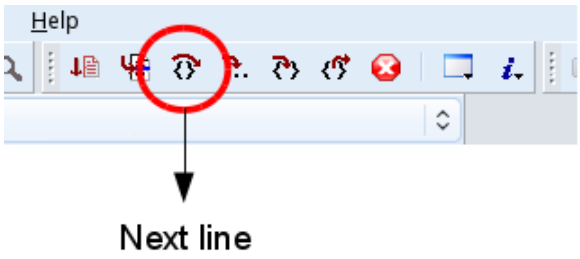

Figura 31: Opción siguiente línea

La siguiente sentencia a ejecutar es ahora la llamada a la función **presentarResultados**. Como el error detectado tiene que ver con la presentación de resultados que no deberían de aparecer, nos preocupamos de forzar al depurador a inspeccionar la ejecución de las sentencias de la función. Para ello se selecciona la opción **Step into**, Fig. 32.

Figura 32: Opción inspeccionar interior de función

Ahora la próxima sentencia a ejecutar es la sentencia **cout** dentro de la función y la ventana **Watches** muestra información sobre los argumentos (además de las variables locales, en caso de haberlas). Los valores observados son:

```
valorActual=1
valor=-1
fin=false
```

Analizando estos datos observamos que se genera información de salida cuando valor anterior es -1. Esto sólo ocurre durante el inicio de la ejecución del programa. Al necesitar valor inicial esta variable (hay que compararla con valor actual) se le ha asignado el valor -1 para asegurarnos que no coincida con ningún valor a tener en cuenta para el incremento del contador. Esto advierte también de la necesidad de tener que comprobar cómo funciona nuestro código en el caso en que el primer valor introducido sea -1 (y arreglarlo, en caso de no ser correcto el funcionamiento).

En cualquier caso, el fallo que estábamos intentando solucionar puede resolverse haciendo que la presentación de resultados se haga sólo en caso en que el argumento valor tenga valor distinto de -1 (es decir, evitando presentar los resultados iniciales). Probamos a introducir una modificación en la función, que ahora quedaría de la forma:

```
void presentarResultados(int repeticiones, int valor, bool fin){
    if (valor != -1){
       cout << valor << "␣-␣" << valor << endl;
    }

    // Si es el fin se indica con una linea de asteriscos
    if (fin){
        cout << "****************************************" << endl;
```

```
    }
}
```

Para poder probar necesitamos detener la ejecución en curso (ver Fig. 33), modificar el código y volver a comenzar el proceso de depuración desde el principio.

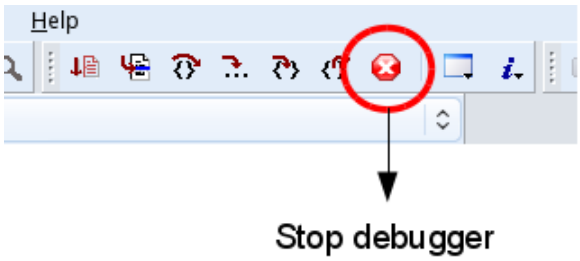

Figura 33: Finalizar depuración en curso

Procedemos de esta forma y observamos ahora qué ocurre cuando la ejecución alcanza la primera sentencia de la función, Fig. 34.

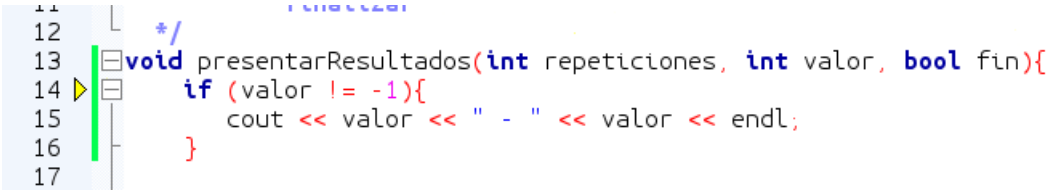

Figura 34: Análisis tras realizar cambio

Estamos interesados en conocer cuál será la siguiente sentencia a ejecutar. Si el cambio era el adecuado no se producirá la salida (observando la ventana **Watches** podemos afirmar que así será), pero lo comprobamos usando la opción siguiente línea del menú de depuración, Fig. 31. Se observa que la siguiente línea a ejecutar es la correspondiente a la siguiente estructura condicional que comprueba si se trata del final del programa.

Como ejercicio práctico, comprobad qué ocurre si el primer valor introducido es -1.

7. Ejercicio adicional

En esta sección se propone el código de resolución de uno de los ejercicios de las relaciones de ejercicios, que contiene errores lógicos, y que debéis depurar para coger soltura en el manejo de esta herramienta.

Se trata del ejercicio 21 de la relación de estructuras de control. Se pide que se diseñe un programa para calcular la suma de los 100 primeros términos de la sucesión

$$a_i = \frac{(-1)^i(i^2-1)}{2i} \tag{1}$$

donde debe calcularse de forma explícita, en la obtención de cada término, el valor $(-1)^i$ (mediante un bucle for). La implementación ofrecida presenta un error lógico. Por ejemplo, para $i = 5$ el sumatorio debe dar $-1{,}10833333$ y no se obtiene este valor. Al final debéis reparar este error y además modularizar el cálculo del término $(-1)^i$ mediante una función.

El código a usar aparece a continuación (página siguiente) y está disponible en decsai.

```
#include <iostream>

using namespace std;

int main()
{
  double terminol, terminoCuadrado, denominador, acumulador=0;

  // Bucle de generacion de los terminos
  for(int i=1; i <= 100; i++){
    // Bucle para calcular la potencia de i
    for(int j=1; j <= i; j++){
      terminol=terminol*(-1);
    }

    // Se calcula el termino al cuadrado
    terminoCuadrado=i*i-1;

    // Se calcula el denominador
    denominador=2*i;

    // Se calcula el termino y se acumula
    acumulador+=terminol*terminoCuadrado/denominador;
  }

  cout << "Suma de terminos: " << acumulador << endl;
}
```