

Prácticas de Programación 1

Grado en Ingeniería Informática



Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

**Miguel Ángel Latre, Ricardo J. Rodríguez, Rafael Tolosana y
Javier Martínez**

Área de Lenguajes y Sistemas Informáticos
Departamento de Informática e Ingeniería de Sistemas



Universidad
Zaragoza

Curso 2021-22



Distribuido bajo licencia CC BY-NC-SA. ©Profesorado del DIIS
<https://creativecommons.org/licenses/by-nc-sa/4.0/deed.es/>

Práctica 2: Programas elementales escritos en C++

2.1. Objetivos de la práctica

Esta práctica tiene dos objetivos:

- En la primera parte de la sesión de laboratorio correspondiente a esta práctica, cada estudiante va a aprender a **utilizar la herramienta de depuración GDB (o GNU Debugger), integrada en el entorno Visual Studio Code**. Un depurador es una herramienta de gran utilidad, ya que ayuda al programador a identificar las causas de los errores de ejecución de sus programas para intentar solucionarlos.
- En la segunda parte de la sesión, cada estudiante debe diseñar e implementar **seis programas C++ que resuelven sencillos programas que, en esencia, se limitan a transformar la información suministrada por el usuario modificando su presentación**.

Cada estudiante debe haber iniciado el desarrollo de estos programas antes de acudir a la sesión de prácticas. Como se ha comentado en la primera práctica, es importante que se venga a las prácticas con el trabajo ya iniciado con objeto de aprovechar el tiempo de prácticas en resolver posibles dudas o cuestiones que pudierais tener, relativas a los programas a desarrollar.

En esta segunda parte de la sesión cada estudiante podrá consultar al profesor las dudas que le hayan surgido y el profesor podrá supervisar y hacer un seguimiento del trabajo realizado por cada estudiante.

Los seis programas planteados en esta práctica deberían estar desarrollados antes de finalizar la semana en la que cada estudiante tiene su sesión de prácticas. No obstante, en la Sección 2.5.2 se especifica la fecha límite en la que entregar la práctica, así como el contenido exacto que se solicita entregar.

En la Sección 2.4 se enuncian tres problemas adicionales, de **complejidad bastante más alta** que los seis anteriores, y que no forman parte de la entrega obligatoria asociada a esta práctica. No obstante, se recomienda su resolución y entregarlos si, una vez programadas sus soluciones, funcionan correctamente.

Por el contrario, si los problemas planteados en la primera parte de la práctica te parecen demasiado complicados, puedes invertir algo de tiempo en tratar de resolver los ejercicios planteados en la sección «Ejercicios básicos» de Moodle. Se trata de ejercicios muy simples y algo repetitivos que pueden ayudarte a familiarizarte con los conceptos que se introducen en las clases de teoría, el lenguaje C++ y con el entorno de programación utilizado en la asignatura.

2.2. Tecnología y herramientas

Los programas con los que se va a trabajar en esta práctica se asociarán a un nuevo directorio de trabajo denominado «practica2», que se ubicará dentro del directorio «programacion1» creado en la práctica 1 para albergar el código desarrollado en la asignatura.


En el repositorio <https://github.com/prog1-eina/practica2> tienes disponible el código fuente del fichero «tablas.cpp», con el que vamos a trabajar en la primera tarea de esta práctica. Para aprender a utilizar el depurador (explicado en la Sección 2.2.1), **en lugar de descargarte todo el repositorio, descarga solo el fichero con el código fuente** (en el enlace <https://raw.githubusercontent.com/prog1-eina/practica2/master/tablas.cpp>) y cópialo a tu directorio «practica2».

El programa correspondiente al fichero «tablas.cpp» es el de la escritura de varias tablas de multiplicar en la pantalla con el que se trabajará más a fondo en el tema 5 del curso. No obstante, vamos a poder ver cómo funciona y cómo se ejecuta paso a paso con la ayuda del depurador en la siguiente sección. Compila y ejecuta el programa para comprobar que la configuración de Visual Studio Code es correcta.



2.2.1. El depurador *GNU Debugger*

Un **depurador** (*debugger* en inglés) es una herramienta que permite detener la ejecución de un programa y ejecutarlo paso a paso para comprobar qué instrucciones del código fuente se están ejecutando y cuáles son los valores de las variables una vez detenido. Habitualmente, esta herramienta se utiliza para localizar los defectos en el código que puedan estar causando errores en la ejecución del mismo.

Vamos a utilizar el depurador con el programa «tablas.cpp». Lanza el Visual Studio Code y abre el directorio «practica2» en el que has descargado el código fuente. Aunque en este caso no hay errores en dicho programa, vamos a usar el depurador GDB para observar paso a paso la ejecución y el comportamiento del programa.

1. En primer lugar, hay que configurar el directorio para poder depurar el programa correspondiente al fichero que se está editando. Para ello, selecciona la opción de menú  **Start Debugging**. Aparecerá un panel emergente en el que seleccionar el entorno de depuración. Selecciona el primero de ellos (debería de ser **C++ (GDB/LLDB)**).

A continuación, hay que elegir la configuración. Selecciona la primera opción **g++.exe - Build and debug active file**. Si aparece repetido dos veces, selecciona la primera.

2. La depuración habrá comenzado inmediatamente. Antes de poder depurar adecuadamente nuestros programas, vamos a tener que modificar la configuración por defecto que proporciona Visual Studio Code.
 - Detén la depuración pulsando en el botón de parada que habrá aparecido en un panel flotante en la parte superior () o seleccionando la orden  **Stop Debugging**.
 - Se habrá abierto automáticamente en el editor un fichero denominado «launch.json» que contiene la configuración de depuración que se utilizará, almacenada en un formato que determinado por los desarrolladores de Visual Studio Code. Se abre automáticamente por si fuera preciso hacer alguna modificación, como es nuestro caso. Este fichero lo crea Visual Studio Code en un directorio denominado «.vscode» donde este entorno guarda toda la configuración relativa al directorio de trabajo que tengamos abierto. En el panel de exploración de ficheros de la parte izquierda de Visual Studio Code (o en el explorador de archivos), puedes observar que aparece este nuevo directorio y, dentro de él, el fichero «launch.json».

- En el fichero «launch.json», localiza una línea que contiene el texto `"externalConsole"`. El valor booleano que aparece a continuación determina si la entrada y salida del programa a depurar se va a realizar en un terminal externo o en el terminal de depuración embebido en Visual Studio Code. Como nuestros programas van a leer datos del teclado, necesitamos realizar la depuración con un terminal externo. Para ello, modifica el valor de la propiedad `"externalConsole"` a `true` para indicarle a Visual Studio Code que se utilice un terminal externo. Tiene que quedarte la línea del fichero «launch.json» algo como esto:

```
...
"externalConsole": true,
...
```

- Guarda los cambios hechos en el fichero de configuración (menú `File >> Save` o combinación de teclas `Ctrl + S`).

Si tu sistema operativo es Windows o GNU/Linux, puedes saltar al siguiente punto.

Si tu sistema operativo es macOS, hay que realizar algunos pasos adicionales debido a temas de permisos de seguridad. Dentro del fichero «tasks.json», que estará localizado también en el directorio «.vscode», tienes que añadir lo siguiente:

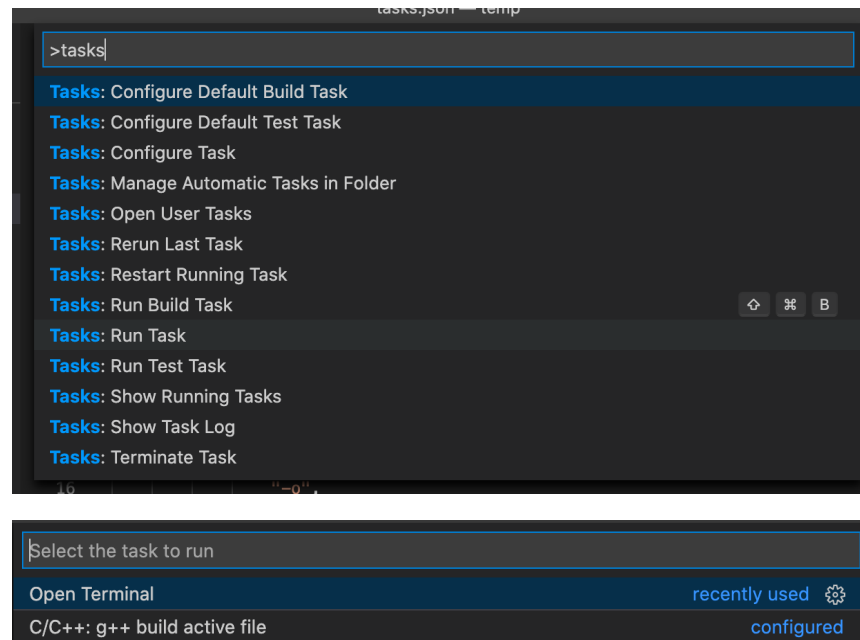
```
{
  "label": "Open Terminal",
  "type": "shell",
  "command": "osascript -e 'tell application \"Terminal\"\\ndo script \"echo hello\"\\nend tell'",
  "problemMatcher": []
}
```

El fichero «tasks.json» debería de quedarte algo similar a esto (observa que tras el texto añadido aparece una coma, dado que hay otra tarea configurada ya):

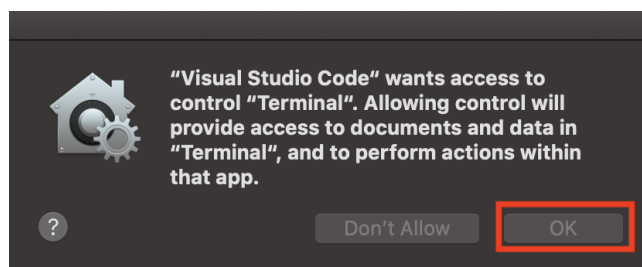
```
.vscode > {} tasks.json > [ ] tasks
1  {
2    "tasks": [
3      {
4        "label": "Open Terminal",
5        "type": "shell",
6        "command": "osascript -e 'tell application \"Terminal\"\\ndo script \"echo hello\"\\nend tell'",
7        "problemMatcher": []
8      },
9      {
10       "type": "shell",
11       "label": "C/C++: g++ build active file",
12       "command": "/usr/bin/g++",
13       "args": [
14         "-g",
15         "${file}",
16         "-o",
17         "${fileDirname}/${fileBasenameNoExtension}"
18       ],
19       "options": {
20         "cwd": "${workspaceFolder}"
21       },
22       "problemMatcher": [
23         "$gcc"
24       ],
25       "group": {
26         "kind": "build",
27         "isDefault": true
28       }
29     ]
30   },
31   "version": "2.0.0"
32 }
```

Una vez salvado el fichero, ejecuta `⌘ + ⌥ + P` para que salga el menú de ejecución. Allí, introduce `tasks` y selecciona `Tasks: Run Task`. En las opciones que aparecerán, seleccio-

na `Open Terminal`, que corresponde a la tarea que has añadido anteriormente en el fichero «tasks.json»:



Aparecerá una ventana donde te pregunta si deseas permitir al Visual Studio Code acceder al programa de terminal. Dale permisos seleccionando el botón `OK`.



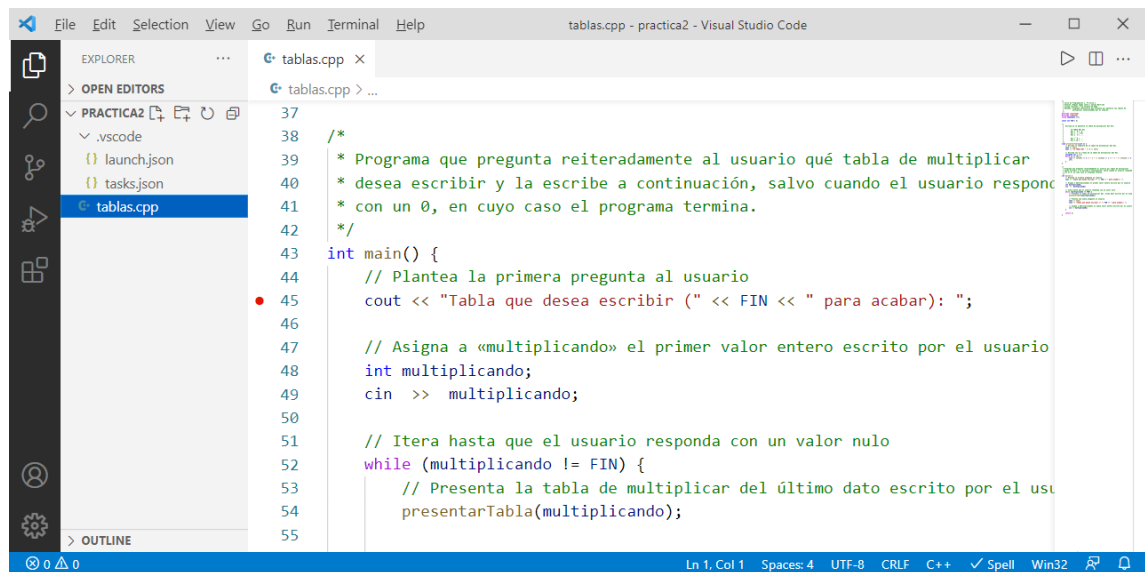
Tras esto, Visual Studio Code ya tendrá permisos para interactuar con la terminal externa cada vez que depures.

Una vez configurada la tarea de depuración en Visual Studio Code (solo habrá que hacerlo una vez por cada directorio en el que trabajes), podemos comenzar a utilizar el depurador.

Si hubieras tenido problemas siguiendo los pasos anteriores, puedes descargar el repositorio completo desde GitHub (<https://github.com/prog1-eina/practica2>), que ya está configurado para poder utilizar el depurador en esta práctica (únicamente para sistemas Windows). Desde GitHub, selecciona `Clone or download` > `Download ZIP` para descargar el contenido. En tu equipo, descomprime el fichero ZIP y copia el directorio «practica2» a «programacion1» (borra el sufijo «-master» que añade GitHub al preparar el fichero comprimido).

3. Para depurar un programa, hay que definir en él un *punto de interrupción* (en inglés, *breakpoint*) en una instrucción de su código fuente. Un punto de interrupción hace referencia a una instrucción del programa en la que se detendrá la ejecución del mismo cuando estemos depurando.

Pon un primer punto de interrupción en la línea 45 del código. Para ello, selecciona con el ratón la línea donde vas a definir el punto de interrupción y haz clic en el margen de la misma, justo a la izquierda del número de línea. El punto de interrupción queda identificado a través de un punto rojo, como se aprecia en la siguiente figura:



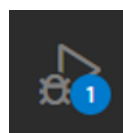
En un programa podemos definir cuantos puntos de interrupción nos interese. Para eliminar un punto de interrupción, basta con volver a hacer clic sobre él.

4. Inicia la depuración de nuevo a través de la orden **Run** > **Start Debugging** (o presionando la tecla **F5**). Si tu sistema operativo es Windows y te da algún error en este paso de iniciar la depuración, puede deberse a dos cosas:
 - a) Te falta de instalar el depurador. Para solucionar esto, busca el **MinGW Install Manager** en el menú **Inicio** de Windows e instala mingw32-gdb-bin, de manera similar a como se hacía en el manual de instalación de MinGW.
 - b) Si el depurador lo tienes instalado y te sigue dando error, verifica que tu directorio de trabajo (aquel donde tienes las práctica de la asignatura y donde estás trabajando con Visual Studio Code) no contiene ningún carácter especial (espacios en blanco, eñes o caracteres con tilde).

La ejecución del programa se inicia, deteniéndose al llegar al punto de interrupción, sobre el que aparece una flecha amarilla que va a indicarnos a partir de ahora cuál es la siguiente instrucción a ejecutar. También ha aparecido de nuevo la barra de herramientas de depuración en la parte superior de la ventana:



En la barra de herramientas de la izquierda de la ventana de Visual Studio Code ha aparecido un círculo azul con un «1» dentro en el icono que da acceso al panel de depuración:

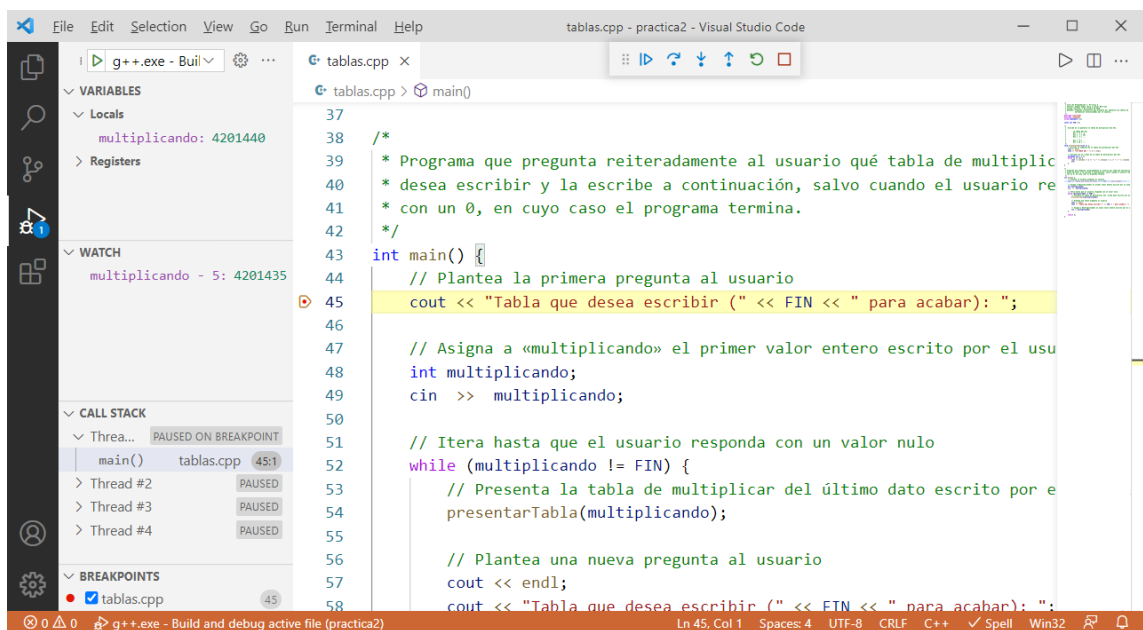


El «1» del círculo azul indica que tenemos una sesión de depuración activa. Haz clic en el icono del panel de depuración para que Visual Studio Code muestre el panel de depuración.

En la parte superior de este panel de depuración, denominada **VARIABLES**, podrán observarse los valores de las variables que se encuentren en ámbito en el punto en el que se halle detenido el programa. En este momento, podemos ver que hay ya una variable denominada `multiplicando`,

junto con su valor actual (que posiblemente sea un valor extraño, porque la variable todavía no ha sido inicializada. Este valor podría variar entre una ejecución y otra).

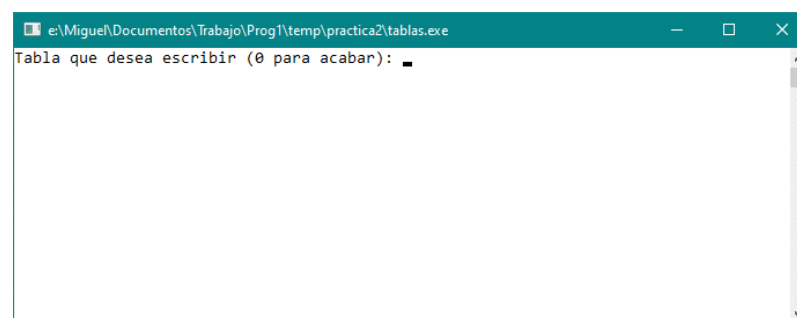
En las partes inferiores del panel tenemos una zona **WATCH** en la que podremos añadir expresiones para que sean evaluadas conforme avance la ejecución del programa que se está depurando (por ejemplo, podríamos añadir una expresión multiplicando - 5 para que nos mostrara todo el rato el valor de la variable multiplicando con 5 unidades menos), una zona **CALL STACK** en la que se observa la pila de llamadas a funciones (conforme se avance en la depuración de un programa, irá apareciendo aquí contenido) y, por último, una lista con los puntos de interrupción presentes en el programa (**BREAKPOINTS**). En esta última zona, podemos ver que se encuentra listado el punto de interrupción de la línea 48 de nuestro fichero «tablas.cpp».



También se ha abierto un terminal en una ventana aparte, a través del cual el programa que estamos depurando interactuará con el usuario.


5. Vamos a ejecutar el programa instrucción a instrucción a través de la orden **Run >> Step Over**. También es posible hacerlo presionando la tecla **F10** o el icono con un punto y una flecha que parece saltar sobre el punto de la barra de herramientas de depuración ().

Observa que, tras ejecutar la instrucción de la línea 45, ha aparecido el texto «¿Qué tabla desea escribir (0 para acabar)?» en el terminal externo y la flecha amarilla se ha movido a la siguiente instrucción ejecutable del código (la instrucción de lectura del teclado de la línea 49).



6. Si ahora vuelves a solicitar la ejecución de la siguiente instrucción, verás como la flecha amarilla desaparece. El programa que estamos depurando está bloqueado hasta que escribas un dato entero en su terminal y completes la línea presionando la tecla **↵**.

Cuando vuelvas a Visual Studio Code, comprueba que la flecha amarilla ha vuelto a aparecer (apuntando a la cabecera del bucle `while`) y que el valor de la variable `multiplicando` ha cambiado y además se ha resaltado con un fondo azul.




7. Ejecuta paso a paso (continúa con la orden `Run >> Step Over`) una iteración completa del bucle de la función `main()`. Comprueba que cuando se ejecuta la instrucción `presentarTabla(multiplicando)`, la tabla de multiplicar del número que hayas elegido aparece escrita de golpe. Cuando el programa solicite un nuevo número, escribe otro número distinto de cero.
8. Cuando ejecutes la segunda iteración del bucle, detente justo antes de la ejecución de la instrucción `presentarTabla(multiplicando)`. Vamos ahora a ejecutar paso a paso las instrucciones del procedimiento `presentarTabla`. Para ello, elige la orden `Run >> Step Into`. También es posible hacerlo presionando la tecla `F11` o el icono con una flecha que apunta hacia abajo acabando en un punto ().

Observa que la siguiente instrucción a ejecutar es la primera del procedimiento `presentarTabla`. A diferencia de la orden `Step Over`, la orden `Step Into` se *mete dentro* de un procedimiento o una función. Recuerda que antes al ejecutar `Step Over` sobre `presentarTabla(multiplicando)` se ejecutaba todo el procedimiento de manera transparente y se detenía en la instrucción siguiente y ahora, sin embargo, el programa está detenido *dentro* del procedimiento.

Observa, además, que ahora tenemos en ámbito el parámetro `n` (que tiene como valor el de la variable `multiplicando`, usada como argumento en la invocación de este procedimiento, y que es igual al segundo dato que has introducido por el teclado) y la variable `i` (que en realidad va a ser declarada un par de instrucciones más adelante).

Comprueba que la variable `multiplicando` de la función `main` no está en ámbito. Aunque la variable siga viva en la ejecución, las reglas de ámbito de C++ impiden hacer uso de ella en el cuerpo del procedimiento `presentarTabla`.

En el tema 5 veremos con más detalle el proceso que se sigue al invocar a funciones y procedimientos y trabajaremos el concepto de *ámbito* de un elemento de un programa.

9. Ejecuta paso a paso el cuerpo del procedimiento `presentarTabla`. Observa que, conforme se ejecuta el bucle, va variando el valor de la variable `i` y se va escribiendo poco a poco en la pantalla la tabla del número `n`.
10. Cuando vuelvas a la función `main` y se te vuelva a pedir un número, introduce un último número distinto de cero y vuelve a llevar el punto de ejecución al cuerpo del procedimiento `presentarTabla`. Puedes hacerlo combinando órdenes `Run >> Step Over` y `Run >> Step Into` o estableciendo un nuevo punto de interrupción en el cuerpo del procedimiento `presentarTabla` y avanzando hasta él con la orden `Run >> Continue` (también `F5` o icono del triángulo azul de la barra de herramientas de depuración, ).
11. Ejecuta un par de iteraciones del bucle `while` del procedimiento `presentarTabla` y utiliza una nueva orden para abandonar el cuerpo de este procedimiento sin terminar de ejecutarla paso a paso: `Run >> Step Out` (también  + `F11` o el icono con una flecha que apunta hacia arriba en dirección contraria a un punto azul: ).
12. Comprueba que el procedimiento `presentarTabla` ha terminado de ejecutarse y que has vuelto a la función `main` (en la terminal aparece toda la tabla de multiplicación). Como ves, la opción `Step Out` termina de ejecutar por completo la función o procedimiento que se esté depurando y devuelve el control después de su ejecución, mientras que la opción `Step Into` permite ir ejecutando una a una las instrucciones de la función o del procedimiento (recuerda lo visto en los pasos anteriores).

13. Termina la ejecución de la depuración, bien terminando el programa paso a paso o bien con la orden   (también  +  o ).

2.2.2. Guía de estilo para programar en C++

En esta práctica vas a comenzar a escribir código propio que puede ser revisado posteriormente por los profesores. Por ello, es conveniente fijar un conjunto de reglas de estilo de codificación de programas en C++ que conviene tener en cuenta al escribir el código. El objetivo de las reglas de estilo es favorecer la calidad del código diseñado y su legibilidad.

La que vamos a utilizar en este curso en Moodle la tienes disponible en https://moodle.unizar.es/add/pluginfile.php/5019702/mod_resource/content/5/guia_estilo.pdf. No todas sus secciones son aplicables en este momento, pero donde se indiquen reglas que puedan seguirse al resolver esta práctica y las siguientes, aplícalas.

2.3. Trabajo a desarrollar en esta práctica

Los problemas que aquí se proponen requieren transformar la información proporcionada por el usuario del programa con un determinado formato en una información equivalente con diferente formato. La información tratada en cada problema es de diferente naturaleza: una fecha del calendario, una cantidad de dinero expresada en una determinada moneda (euro, peseta, etc.), la medida de un ángulo, la duración de un periodo de tiempo, etc.

Cada problema exige la puesta a punto de un programa completo en el cual se leen los datos que proporciona el usuario y se calculan y presentan por pantalla los resultados que correspondan.

Los programas que resuelven estos seis problemas se desarrollarán en el directorio «practica2». En ella se crearán seis ficheros, uno por cada uno de los programas a desarrollar: «fecha.cpp», «cambio-moneda.cpp», «cajero.cpp», «tiempo.cpp», «romano.cpp» y «trigonometria.cpp». La descripción del comportamiento de los programas a desarrollar se detallan en las secciones 2.3.1 a 2.3.6.

Este trabajo de desarrollo de programas debe ir acompañado por la realización de un conjunto de pruebas de búsqueda de posibles errores. Esta estrategia se aplicará a todos los trabajos de programación que se lleven a cabo en esta asignatura, ya que **el desarrollo de un programa no concluye hasta que se han realizado todas las pruebas necesarias para validar su buen comportamiento**. El objeto de estas pruebas es localizar errores y fallos de funcionamiento para corregir, en su caso, sus causas en el código fuente y lograr el programa se comporte respetando sus especificaciones para cualquier juego de datos dado.

En las descripciones de los programas que han de desarrollarse, se dan algunas pautas para la realización de pruebas sobre cada uno de ellos. En todos los casos, se propone reflexionar sobre los datos con los que probarlos y sobre los resultados que debería ofrecer el programa antes incluso de haberlo programado, ya que dicha reflexión previa puede ayudar a terminar de comprender el problema. Es por ello aconsejable determinar *a mano*, antes de comenzar el desarrollo los programas, los resultados esperados de los casos de prueba que se establezcan. Una vez desarrollado el programa, será necesario comparar los resultados ofrecidos por el programa con los resultados esperados al diseñar el caso de prueba, y corregir los errores que pueda haber (si existe alguno).

En una primera etapa, desarrolla los seis programas propuestos suponiendo que, cuando al usuario se le solicitan datos numéricos que cumplen con una serie de características (por ejemplo, el número tiene que ser positivo), el usuario los va a proporcionar cumpliendo dichas características. Los casos de prueba, por lo tanto, se centrarán en verificar que el programa funciona correctamente con entradas de

usuario válidas.

En una segunda etapa (detallada en la Sección 2.3.7), se te pedirá modificar ligeramente los programas realizados para que sean algo más robustos ante entradas del usuario que no cumplan con todas las restricciones impuestas.

2.3.1. Programa «fecha». Cambio de formato de una fecha

Debe desarrollarse un programa C++ que interaccione con el usuario mostrando el siguiente comportamiento (recuerda que este ejemplo de interacción muestra la interacción deseada del algoritmo: por un lado, las cadenas que escribe el programa por pantalla; y por otro, subrayado y en negrita, los valores introducidos por el usuario):

```
Escriba una fecha con formato de 8 cifras [aaaammdd]: 14921012  
La fecha escrita es 12/10/1492.
```

La fecha escrita por el usuario con formato *aaaammdd*, que se ha subrayado al transcribir el ejemplo de ejecución, es reescrita por el programa con formato *dd/mm/aaaa*, donde *dd* es el día de la fecha, *mm* es el mes y *aaaa* es el año. Por el momento, vamos a suponer que **el usuario siempre va a escribir números positivos que representan fechas válidas con formato *aaaammdd***, con el mes comprendido entre 1 y 12 y el día entre 1 y el último día del mes correspondiente.

Un segundo ejemplo se presenta a continuación para ilustrar el comportamiento deseado del programa.

```
Escriba una fecha con formato de 8 cifras [aaaammdd]: 20140706  
La fecha escrita es 06/07/2014.
```

Fíjate que en este programa «fecha.cpp» se han proporcionado dos ejemplos distintos de ejecución que cubren la posibilidad de que tanto el día como el mes tengan una o dos cifras. Observa que, en los casos en los que el día o el mes tengan solo una cifra, la escritura debe hacerse utilizando siempre dos dígitos, anteponiendo un 0. Para obtener este comportamiento, quizás conviene que recuerdes los manipuladores para dar formato a los datos de salida explicados en la Práctica 1 (en su sección 1.2.5).

2.3.2. Programa «cambio-moneda». Conversión de euros a pesetas

El programa interactivo que debe ser desarrollado pide al usuario que escriba una cantidad de dinero **no negativa** expresada en euros y procede a informarle de su desglose en euros y céntimos y también de su equivalente en pesetas. Recuerda que 1 euro son 166,386 pesetas.

```
Escriba una cantidad no negativa de dinero en euros: 43.653  
Son 43 euros y 65 céntimos que equivalen a 7263 pesetas.
```

Observa que el programa escribe la cantidad entera de euros por un lado y la de céntimos por otro. Además, redondea la cantidad de céntimos al céntimo más próximo y la cantidad de pesetas equivalentes es también redondeada al número entero de pesetas más próximo al resultado exacto.

Un segundo ejemplo se presenta a continuación para ilustrar el comportamiento deseado del programa.

Escriba una cantidad no negativa de dinero en euros: **43.6583**
 Son 43 euros y 66 céntimos que equivalen a 7264 pesetas.

Este programa vamos a probarlo exhaustivamente. Además de con los ejemplos del enunciado, vamos a probarlo con los valores límite mínimo y máximo que provocan que cada uno de los tres datos fundamentales de salida (euros enteros, céntimos y pesetas) pasen de 0 a 1. En la tabla que aparece a continuación, cada fila representa una prueba específica del programa a partir de una entrada de usuario concreta. Para cada caso de prueba, además de la entrada del usuario, se indica la razón por la que se incluye la prueba y el resultado esperado de la misma.

Entrada usuario	Razón del caso de prueba	Resultado esperado
43.653	Primer ejemplo del enunciado	Son 43 euros y 65 céntimos que equivalen a 7263 pesetas.
43.6583	Segundo ejemplo del enunciado	Son 43 euros y 66 céntimos que equivalen a 7264 pesetas.
0	Mínimo valor posible de la entrada del usuario	Son 0 euros y 0 céntimos que equivalen a 0 pesetas.
0.001	Mínimo valor posible no nulo de la entrada del usuario (considerando solo 3 decimales)	Son 0 euros y 0 céntimos que equivalen a 0 pesetas.
0.003	Máximo valor cuya conversión es 0 pesetas	Son 0 euros y 0 céntimos que equivalen a 0 pesetas.
0.004	Mínimo valor cuya conversión es 1 peseta y máximo que corresponde con 0 céntimos	Son 0 euros y 0 céntimos que equivalen a 1 pesetas.
0.005	Mínimo valor cuyo redondeo corresponde con 1 céntimo	Son 0 euros y 1 céntimos que equivalen a 1 pesetas.
0.994	Máximo valor que corresponde con 0 euros enteros	Son 0 euros y 99 céntimos que equivalen a 165 pesetas.
0.995	Mínimo valor que corresponde con 1 euro entero	Son 1 euros y 0 céntimos que equivalen a 166 pesetas.
20	Un valor entero (sin decimales) distinto de 0	Son 20 euros y 0 céntimos que equivalen a 3328 pesetas.

2.3.3. Programa «cajero». Funcionamiento de un cajero automático

El programa interactivo a desarrollar presenta el siguiente comportamiento:

Cantidad a retirar en euros [positiva y múltiplo de 10]: **280**

Billetes	Euros
=====	=====
1	10
1	20
5	50

El programa informa al usuario del número de billetes que le devolverá un cajero al retirar la cantidad de dinero especificada, que será **positiva y múltiplo de 10**. Conviene advertir que el cajero dispone únicamente de billetes de diez, de veinte y de cincuenta euros y que siempre minimizará el número de billetes a entregar.

Cantidad a retirar en euros [positiva y múltiplo de 10]: 590	
Billetes	Euros
=====	=====
0	10
2	20
11	50

Además de los ejemplos anteriores, haz pruebas con los valores mínimo y máximo que se pueden sacar de un cajero automático: 10 y 600 euros, respectivamente. Haz pruebas también con las tres cantidades que obligan al cajero a entregar exactamente un billete, ya sea de 10, 20 o 50 euros. Haz también una prueba adicional con la cantidad de dinero que obliga al cajero a entregar un total de tres billetes, siendo exactamente uno de cada tipo.

Puedes organizar el diseño de estas pruebas en una tabla similar a la del programa anterior.

2.3.4. Programa «tiempo». Medida del tiempo

El programa interactivo a desarrollar pide al usuario que exprese el tiempo de duración de un determinado evento como una cantidad **natural** expresada en segundos, para informar a continuación por la pantalla de la equivalencia del tiempo introducido en días, horas, minutos y segundos.

Duración en segundos: 615243
Este tiempo equivale a 7 días 2 horas 54 minutos y 3 segundos

Un segundo ejemplo que muestra los resultados que presenta el programa:

Duración en segundos: 11412
Este tiempo equivale a 0 días 3 horas 10 minutos y 12 segundos

Haz pruebas, además de con estos dos ejemplos de ejecución, con el mínimo valor posible de la duración (0 segundos), con las tres duraciones máximas que dan equivalencias estrictamente inferiores a un minuto, a una hora y a un día, así como con las cuatro duraciones mínimas que equivalen exactamente a un segundo, un minuto, una hora y un día. Como en los casos anteriores, organiza los casos de prueba en una tabla.

2.3.5. Programa «romano». Escritura de números romanos

El programa interactivo a desarrollar presenta el siguiente comportamiento:

Escriba un entero entre 1 y 10: 8
8 = VIII

El programa pregunta al usuario por un número entero en el intervalo [1, 10] y le informa por pantalla de su equivalencia como número romano. Un segundo ejemplo que muestra los resultados que presenta el programa:

```
Escriba un entero entre 1 y 10: 5  
5 = V
```

Un último ejemplo del diálogo entre programa y usuario correspondiente a una nueva ejecución del programa:

```
Escriba un entero entre 1 y 10: 4  
4 = IV
```

El programa «romano.cpp» debes probarlo con todos los valores comprendidos entre 1 y 10, ambos inclusive.

2.3.6. Programa «trigonometria». Ángulos y funciones trigonométricas

El programa interactivo a desarrollar pide al usuario que defina un ángulo en grados, minutos y segundos sexagesimales y le informa por pantalla, a continuación, del valor equivalente de ese ángulo en radianes, así como de los valores de su seno, coseno y tangente. El programa desarrollado debe presentar los resultados **respetando al máximo el formato mostrado en los siguientes ejemplos**: los ángulos en radianes se expresan con tres cifras decimales y los valores de las funciones trigonométricas seno, coseno y tangente, con cuatro decimales.

```
Escriba el valor de un ángulo (grados, minutos y segundos): 60 0 0  
Valor del ángulo en radianes: 1.047 radianes  
sen 1.047 = 0.8660  
cos 1.047 = 0.5000  
tg 1.047 = 1.7321
```

Un segundo ejemplo que muestra los resultados que presenta el programa:

```
Escriba el valor de un ángulo (grados, minutos y segundos): 112 9 45  
Valor del ángulo en radianes: 1.958 radianes  
sen 1.958 = 0.9261  
cos 1.958 = -0.3772  
tg 1.958 = -2.4550
```

Este programa, además de ser probado con los ejemplos del enunciado, conviene ser probado con ejemplos en los que tanto los grados, como los minutos y los segundos tomen los valores que razonablemente se considerarían como límites mínimo y máximo razonables (0 en cuanto al mínimo de los tres, y 359 o 59 en lo que respecta a los máximo para grados por un lado y minutos y segundos, por otro). También se debería probar con la menor entrada posible no nula ($0^{\circ} 0' 0''$) y con una entrada que sea mayor o igual que 360 grados. También nos podemos plantear qué debería ocurrir en el caso de entradas más extrañas, como por ejemplo que en algún caso los minutos o los segundos fueran mayores o iguales que 60 (por ejemplo, $30^{\circ} 90' 0''$ no es la forma más elegante de escribir el ángulo $31^{\circ} 30' 0''$, pero no es ningún error).

Completa para ello los datos que faltan en la siguiente tabla:

Entrada usuario			Razón del caso de prueba	Resultado esperado			
°	'	''		rad	sen	cos	tg
60	0	0	Primer ejemplo del enunciado.	1,047	0,8660	0,5000	1,7321
112	9	45	Segundo ejemplo del enunciado.	1,958	0,9261	-0,3772	-2,4550
0	0	0	Mínimo valor posible de la entrada del usuario.
0	0	1	Mínimo valor no nulo.
359	59	59	Máximos habituales para grados, minutos y segundos.
...	Más de 360°.
...	Más de 60 minutos.
...	Más de 60 segundos.

2.3.7. Robustez de los programas desarrollados

Al desarrollar y probar los programas anteriores, se ha trabajado bajo la hipótesis de que el usuario va a proporcionar entradas válidas cuando le son solicitadas (por ejemplo, si se le pide un entero positivo múltiplo de diez, el usuario va a escribir un dato que cumpla con dichas características). En esta sección, se va a aumentar la robustez de los programas desarrollados desconfiando (solo ligeramente) de las entradas proporcionadas por el usuario y comprobando que las restricciones impuestas en cada problema realmente se cumplen.

Lamentablemente, va a ser posible hacerlo solo de forma muy limitada, ya que en todos y cada uno de los problemas se está solicitando al usuario datos de tipo numérico leídos del teclado directamente a variables de tipo entero o real a través del operador de extracción (>>). Las técnicas para detectar y resolver el problema de que el usuario introduzca información no numérica requerirían cambios importantes en la forma en la que leemos los datos del teclado y no forman parte del objetivo de esta práctica. En cambio, sí que va a ser posible comprobar que los datos numéricos facilitados cumplan con las restricciones impuestas en cada problema:

Programa «fecha». Pese a que el enunciado indica que el usuario va a introducir siempre datos que representan fechas correctas, piensa (antes de ejecutarlo) en cómo debería comportarse el programa si el usuario introduce un número negativo, como -20181012.

En este caso, convendría que el programa escribiera un mensaje de error en la pantalla en lugar de intentar transformar la fecha. En este caso, además, la función `main` debería devolver un entero positivo en lugar de 0, para indicar que la ejecución no ha sido correcta. El entero que devuelve la función `main` se considera el *código de salida* del programa. En la mayoría de los sistemas operativos, que un programa acabe con un código de salida 0 indica que el programa terminó de forma correcta, mientras que un código mayor que 0 significa que el programa terminó erróneamente. El valor del código de salida suele representar en este caso el código de error del programa.

No te preocupes de otro tipo de entradas erróneas por parte del usuario en este caso, como que este introduzca un entero que no represente una fecha correcta del calendario (como, por ejemplo, 20181033, 20191000, 20210015 o 20221315). Parte de la práctica práctica 3 está centrada a trabajar con fechas del calendario y no merece la pena preocuparse por esto ahora.

Programa «cambio-moneda». Piensa primero en cómo se va a comportar tu programa si el usuario introduce un valor negativo de euros y ejecútalo para comprobarlo. Prueba con los valores opuestos a la tabla de casos de prueba de este programa y observa si los resultados son también los opuestos a los indicados como esperados en la tabla.

Analiza cuál es la causa por la que los resultados del programa no son correctos.

Como antes, se puede aumentar la robustez de este programa haciendo que escriba un mensaje de error en la pantalla cuando la entrada del usuario sea negativa, y haciendo que la función `main` devuelva un entero positivo en lugar de 0 para indicar que la ejecución no ha sido correcta.

Programa «cajero». Los datos de tipo entero que como entrada de usuario no cumplirían con la restricción de que fuese un entero positivo y múltiplo de 10 son:

- a) enteros negativos; y
- b) enteros positivos que no sean múltiplos de 10.

En el primer caso, el cajero no puede proporcionar billetes y en el segundo, no podría proporcionar toda la cantidad de dinero solicitada. Por tanto, sería adecuado considerar estos casos como situaciones de error y limitarse a escribir un mensaje de error en la pantalla en lugar de escribir el número de billetes de cada tipo en la pantalla.

Programa «tiempo». Piensa primero en cuál debería ser la salida del programa si el usuario escribiera -615243 como entrada y ejecútalo para comprobarlo. Si el resultado del programa difiere del que consideras que tendría que tener en el caso de enteros negativos, corrígelo. En caso contrario, déjalo como está.

Programa «romano». ¿Qué datos enteros no serían entradas válidas? Modifica el programa para que en esos casos, muestre un mensaje de error y la función `main` devuelva un valor positivo para indicar una terminación con error.

Programa «trigonometria». En el caso de este programa, cabe plantearse que debería ocurrir en el caso de entradas más problemáticas: *¿qué debería ocurrir si se expresa el ángulo con una cantidad negativa de grados? ¿Y si solo los minutos o los segundos son negativos? ¿Y si la entrada es de exactamente 90 o 270 grados?*

Conviene decidir antes de comprobar cómo se comporta el programa si esas situaciones descritas son o no son de error (no necesariamente lo son) y, en el caso de no serlo, cuál debe ser el comportamiento del programa.

Entrada usuario			Razón del caso de prueba	Resultado esperado			
°	'	"		rad	sen	cos	tg
...	Grados negativos (minutos y segundos nulos o positivos).	
...	Minutos negativos (grados y segundos nulos o positivos).	
...	Segundos negativos (grados y minutos nulos o positivos).	
90	0	0	Tangente no definida.	1,571	1,0000	0,0000	$\pm\infty?$
270	0	0	Tangente no definida.	1,571	1,0000	0,0000	$\pm-\infty?$

2.4. Problemas adicionales

2.4.1. Calculadora de expresiones enteras simples

Diseña un programa que, de forma reiterada, solicite al usuario la introducción de una expresión entre dos datos enteros y un solo operador. El primer entero tiene que ser distinto de cero y las operaciones que admite el programa y los símbolos con los que se van a representar son los siguientes:

Operación	Símbolo
Suma	+
Resta	—
Multiplicación	*
Cociente de la división entera	/
Resto de la división entera	%
Potenciación	^

Los símbolos escogidos son los mismos que se utilizan en C++, excepto en el caso de la potenciación. El programa termina cuando el usuario introduce 0 como primer número de la expresión.

Se muestra a continuación un ejemplo de ejecución del programa:

```

Escriba una operación entre enteros: 6 + 8
6 + 8 = 14

Escriba una operación entre enteros: 125 - 28
125 - 28 = 97

Escriba una operación entre enteros: -7 * 6
-7 * 6 = -42

Escriba una operación entre enteros: 99 / 4
99 / 4 = 24

Escriba una operación entre enteros: 99 % 4
99 % 4 = 3

Escriba una operación entre enteros: 2 ^ 8
2 ^ 8 = 256

Escriba una operación entre enteros: 0

```

Nota para la resolución: El operador de extracción (>>) permite leer del teclado y almacenar en una variable cualquier tipo de dato primitivo, **saltándose cualquier espacio en blanco, salto de línea o tabulador** sin necesidad de programar nada especial.

Escribe el programa que resuelve este problema en un fichero denominado «calculadora-basica.cpp» en el directorio «practica2».

2.4.2. Distancia euclídea entre dos puntos

Escribe un programa que, dadas las coordenadas de dos puntos del plano por el usuario, indique la distancia euclídea que existe entre ellos:

```

Introduzca las coordenadas de un punto: 0.0 0.0
Introduzca las coordenadas de otro punto: 1.0 3.0
La distancia entre los puntos es de 3.1623 unidades

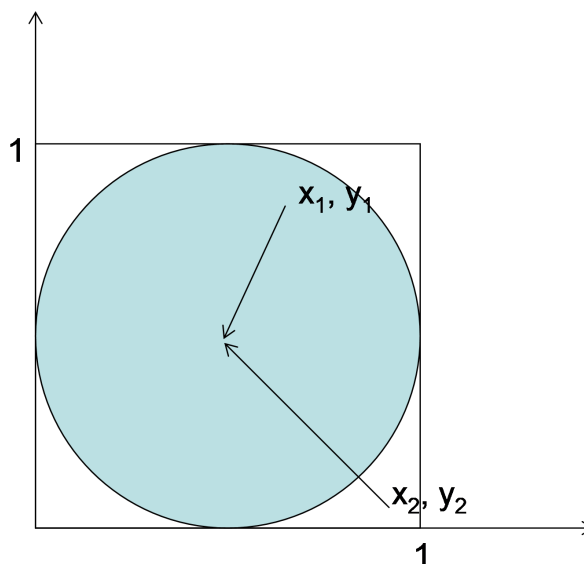
```

Escribe el programa que resuelve este problema en un fichero denominado «distancia.cpp» en el directorio «practica2».

2.4.3. Aproximación de π

El método de Montecarlo sirve para calcular aproximaciones de áreas encerradas bajo la curva de una función. Se basa en generar un número n de puntos aleatorios y calcular cuántos se encuentran dentro de la curva.

Escribe un programa en C++ que utilice el método de Montecarlo para hallar una aproximación de π , utilizando una curva cuya área se conoce. Para ello, puede tomarse una circunferencia de radio $\frac{1}{2}$ y tangente a ambos ejes de coordenadas ubicada en el primer cuadrante y generar puntos aleatorios ($0 \leq x \leq 1; 0 \leq y \leq 1$) en el cuadrado que circunscribe dicha circunferencia. La aproximación de π puede obtenerse analizando la relación existente entre el número n de puntos aleatorios generados, el número de puntos que efectivamente han «caído» en el círculo y la probabilidad teórica de que un punto generado aleatoriamente se encuentre dentro del círculo.



Para generar números pseudoaleatorios en C++ puede utilizarse la función `rand()`¹ de la biblioteca `<cstdlib>`, que genera números pseudoaleatorios entre 0 y `RAND_MAX`. Por ejemplo, el siguiente fragmento de código escribiría en pantalla 10 números pseudoaleatorios:

```
for (int i = 0; i < 10; i++) {  
    cout << rand() << endl;  
}
```

Si ejecutas el código anterior varias veces, comprobarás que la secuencia de números generada es siempre la misma. En el programa que se pide desarrollar, esto no es un problema. Sin embargo, si se desea que la secuencia de números pseudoaleatorios sea distinta cada vez, es preciso invocar al procedimiento `srand()`² previamente a la invocación a `rand()`. Normalmente, el procedimiento `srand()` se ha de invocar **una única vez** al comienzo del programa.

Por último, conviene saber que la implementación de la función `rand()` es dependiente de la implementación de cada compilador y la calidad de la secuencia de números pseudoaleatorios que produce,

¹Descripción de la función en <http://www.cplusplus.com/reference/cstdlib/rand/>

²Descripción del procedimiento en <http://www.cplusplus.com/reference/cstdlib/srand/>

así como su distribución y periodo, no está garantizada. C++ dispone de una biblioteca predefinida denominada `<random>`, que es la que debería utilizarse en programas en los que la calidad de la pseudoaleatoriedad sea importante.

Escribe el programa que resuelve este problema en un fichero denominado «aproximacion-pi.cpp» en el directorio «practica2».

2.5. Resultados y entrega de la práctica

2.5.1. Resultados

Como resultado de esta segunda práctica, cada estudiante dispondrá en su equipo de una carpeta denominada «programacion1» dentro de la cual se localizarán los siguientes ficheros organizados en el directorio «practica2»:

- Fichero C++ «fecha.cpp».
- Fichero C++ «cambio-moneda.cpp».
- Fichero C++ «cajero.cpp».
- Fichero C++ «tiempo.cpp».
- Fichero C++ «romano.cpp».
- Fichero C++ «trigonometria.cpp».

Si has realizado los problemas adicionales, sus soluciones deberían estar también en el directorio «practica2», contenidas en los siguientes ficheros:

- Fichero C++ «calculadora-simple.cpp».
- Fichero C++ «distancia.cpp».
- Fichero C++ «aproximacion-pi.cpp».

2.5.2. Entrega de la práctica

Antes del **sábado 23 de octubre a las 18:00** deberán haberse subido a Moodle los ficheros «fecha.cpp», «cambio-monedas.cpp», «cajero.cpp», «tiempo.cpp», «romano.cpp» y «trigonometria.cpp». Puede tratarse tanto de la versión «básica» solicitada en las secciones 2.3.1 a 2.3.6, como de la «robusta» solicitada en la sección 2.3.7.

Si se han realizado uno o más de los problemas adicionales, se pueden subir también a Moodle los ficheros correspondientes («calculadora-simple.cpp», «distancia.cpp» y «aproximacion-pi.cpp»), antes del mismo día a la misma hora.