

# Práctica 3: Diseño modular de programas C++ con enteros

## 3.1. Objetivos de la práctica

Los objetivos de la práctica son los siguientes:

- Programar algunas funciones que resuelven problemas de tratamiento de números enteros y de gestión de fechas.
- Desarrollar algunos módulos de biblioteca en los que se definen funciones que trabajan con números enteros y con fechas.
- Aprender a poner a punto programas modulares en C++, desarrollando módulos de biblioteca, por un lado; y programas C++ que hacen uso de ellos, por otro.
- Introducir el concepto de pruebas de unidad, ejecutando código diseñado específicamente para detectar errores en módulos de biblioteca.

Al igual que en la práctica anterior, **lo deseable es que acudas a la sesión asociada a la práctica con el trabajo ya comenzado y avanzado**, con el objetivo de aprovechar la sesión de prácticas para resolver dudas, completar aquellas tareas en las que hayan surgido dificultades no superadas y para que el profesor pueda supervisar tu trabajo realizado y pueda hacer sugerencias para su mejora.

## 3.2. Trabajo a desarrollar en esta práctica

Los proyectos de programación a desarrollar en esta práctica se localizarán en un nuevo directorio denominado «practica3» y ubicado en la carpeta «programacion1».

En el repositorio <https://github.com/prog1-eina/practica3> tienes el código de partida para esta práctica, que contiene los programas con los que se va a trabajar ya configurados para ser compilados, ejecutados y depurados. Puedes descargarte el área de trabajo completa (botón [Code](#) » [Download ZIP](#)), y descomprimirla en tu directorio «programacion1» como «practica3» (borra el sufijo «-master» que añade GitHub al preparar el fichero comprimido).

El repositorio contiene tres directorios: «.vscode», «src» y «test».

En esta práctica, vas a tener que trabajar con los ficheros de la carpeta de código fuente «src». En concreto, en las primeras tareas trabajarás con el código del directorio «src/calculadora». Este directorio contiene el código de un programa denominado «calculadora» («calculadora.exe» en Windows) descompuesto en módulos que ya está creado y configurado. La idea es que observes su estructura




siguiendo las indicaciones de la sección 3.2.1, y que realices pequeñas modificaciones en el código (secciones 3.2.2 y 3.2.3). En la sección 3.2.4 observarás la organización de un programa estructuralmente más complicado, denominado «calculos-test» («calculos-test.exe» en Windows) y cuyo código está repartido entre los directorios «src/calculadora», «test/calculadora» y «test/testing-prog1». Este programa hace pruebas de algunas de las funciones del programa «calculadora».

Las tareas de las secciones 3.2.6 y 3.2.7 solicitan que, a partir del material descargado del repositorio de GitHub y disponible en el directorio «src/calendario», completes el código de un par de programas, en los que reside la mayor carga de programación de esta práctica.

Antes de comenzar a trabajar en la práctica, se recomienda la lectura completa (aunque sea somera) del enunciado de esta práctica, con objeto de obtener una idea del conjunto de las tareas que la componen y de la carga de trabajo que puede suponer. Después, se puede avanzar secuencialmente desde el principio de la misma, leyendo **atenta y detenidamente** las explicaciones que se dan.

### 3.2.1. Estructura modular de un programa

Todo programa C++ consta de, al menos, un módulo principal que incluye una función `main` (esta es la primera función a la que se invoca cuando comienza la ejecución de un programa C++). Un programa C++ puede presentar una estructura modular si, además del módulo principal, consta de uno o más módulos de biblioteca adicionales.

Se va a tomar como punto de partida el programa interactivo y dirigido por menú de estructura modular presentado en el tema 7 de la asignatura, cuyo código se encuentra en el directorio «src/calculadora» del repositorio que has descargado. Para verlo, abre en Visual Studio Code el directorio «practica3» (con la opción de menú  Open Folder...) y, después, despliega las carpetas «src» y «calculadora».

Este proyecto está ya configurado para poder trabajar con código distribuido en más de un fichero. Vamos a observar su estructura:

- El código del programa se encuentra ubicado en el directorio «calculadora» de la carpeta «src» y consta de un módulo principal y de un módulo de biblioteca (el módulo «calculos»), organizados en un total de tres ficheros de código fuente, que son:
  - Fichero «calculadora-main.cpp», con el código del módulo principal del programa.
  - Fichero «calculos.hpp» de interfaz del módulo «calculos».
  - Fichero «calculos.cpp» de implementación del módulo «calculos».
- El fichero «Makefile», que contiene las reglas necesarias para que la herramienta «make» invoque al compilador «g++» de forma que se genere el programa «calculadora» o «calculadora.exe», según sea tu sistema operativo.

Aunque con eso es suficiente para este programa, se han configurado tareas de Visual Studio Code para facilitar las tareas de compilación, ejecución y depuración del programa. Esta configuración está almacenada en un par de ficheros del directorio «.vscode», cuyo contenido se describe más adelante en esta sección.

El propósito del directorio «test» se explica en más detalle en la sección 3.2.4.

## Uso de la directiva `#include`

Los recursos definidos en la interfaz del módulo de biblioteca `calculos` han de ser visibles desde el módulo principal. Para lograrlo, en el código del módulo principal, localizado en el fichero «`calculadora-main.cpp`», se ha escrito una directiva del precompilador `#include`, que se encarga de insertar en el código a compilar el código del fichero de interfaz especificado (en este caso, el módulo «`calculos`»).

```
#include "calculos.hpp"
```

Compruébalo en el fichero «`calculadora-main.cpp`».

Esta directiva hace que el preprocesador de C++ trate el contenido del fichero de interfaz especificado («`calculos.hpp`») como si estuviera escrito en el propio fichero «`calculadora-main.cpp`» (concretamente, en el punto donde se ha situado la directiva). Con ello se logra que los elementos declarados en el fichero de interfaz del módulo de biblioteca «`calculos`» **estén en ámbito a partir de ese punto del módulo principal del programa** y que, por lo tanto, puedan ser utilizados en las funciones definidas en este.

Conviene insistir que el fichero que hay que especificar en la directiva `#include` es el fichero de interfaz del módulo (el que tiene extensión «`.hpp`») y **no** el fichero de implementación (fichero con extensión «`.cpp`»).

## Compilación y ejecución del proyecto «Calculadora»

Vamos a compilar el proyecto «Calculadora» desde una terminal. Puedes utilizar la terminal integrada de Visual Studio Code (puedes abrir una nueva con la orden `Terminal` > `New Terminal`) o abrir una terminal externa en el directorio «`practica3`», tal y como se explicó en la tarea 1b de la sección 1.3 de la práctica 1.

Compilaremos inicialmente el programa ejecutando cada una de las tres invocaciones necesarias al compilador «`g++`»:

```
> cd src
> cd calculadora
> g++ -c calculos.cpp -o calculos.o
> g++ -c calculadora-main.cpp -o calculadora-main.o
> g++ calculadora-main.o calculos.o -o calculadora
```

Los dos primeros comandos (`cd src` y `cd calculadora`) han servido para que la terminal se ubique en el directorio en el que se encuentra el código fuente.

Los dos comandos siguientes han hecho que se compilen los ficheros de código fuente «`calculos.cpp`» y «`calculadora-main.cpp`», generándose los ficheros «`calculos.o`» y «`calculadora-main.o`». Habrás visto que estos ficheros aparecían uno a uno, conforme compilabas, en el directorio «`src/calculadora`». Estos ficheros «`calculos.o`» y «`calculadora-main.o`» contienen el código máquina binario equivalente al código C++ de los ficheros «`calculos.cpp`» y «`calculadora-main.cpp`», pero ninguno de los dos es todavía ejecutable: al fichero «`calculos.o`» le falta un punto de entrada (porque en «`calculos.cpp`» no hay definida ninguna función `main`) y al fichero «`calculadora-main.o`» le falta el código de las funciones definidas en «`calculos.cpp`» (además de otras particularidades para ser realmente ficheros ejecutables). A estos ficheros se les denomina *ficheros objeto* o *resultados intermedios de la compilación*.

Es el comando `g++ calculadora-main.o calculos.o -o calculadora` el que ha enlazado el código binario de «`calculos.o`» y «`calculadora-main.o`» para generar el programa ejecutable («`calculadora`» o «`calculadora.exe`», dependiendo del sistema operativo en el que estés trabajando).

Ahora puedes ejecutar el programa «`calculadora`» escribiendo lo siguiente en el terminal:

```
> ./calculadora
```

o, en caso de estar en una terminal de Windows 8 o anterior:

```
> calculadora.exe
```

Compilar cada fichero de un programa individualmente sería propenso a errores y, además, no es cómodo. El gestor de compilación «`make`» permite especificar las instrucciones necesarias para compilar un programa a través de reglas escritas en ficheros denominados «`Makefile`» o ficheros que terminan con la extensión «`.mk`».

En el caso del programa «`calculadora`», os hemos proporcionado las reglas para su compilación en el fichero «`Makefile`» del directorio «`practica3`». Vamos a ejecutarlas a través del comando `make`:

```
> cd ..
> cd ..
> make
mkdir build
g++ -g -Wall -Wextra -Isrc/calculadora -Itest/testing-prog1 -c src/calculadora/calculos.cpp
                                                    -o build/calculos.o
g++ -g -Wall -Wextra -Isrc/calculadora -Itest/testing-prog1 -c src/calculadora/calculadora-main.cpp
                                                    -o build/calculadora-main.o
mkdir bin
g++ -g build/calculos.o build/calculadora-main.o -o bin/calculadora
```

Los dos primeros comandos (`cd ..`) han servido para que la terminal se vuelva a ubicar en el directorio correspondiente a la práctica («`practica3`»), en el que se encuentra el fichero «`Makefile`».

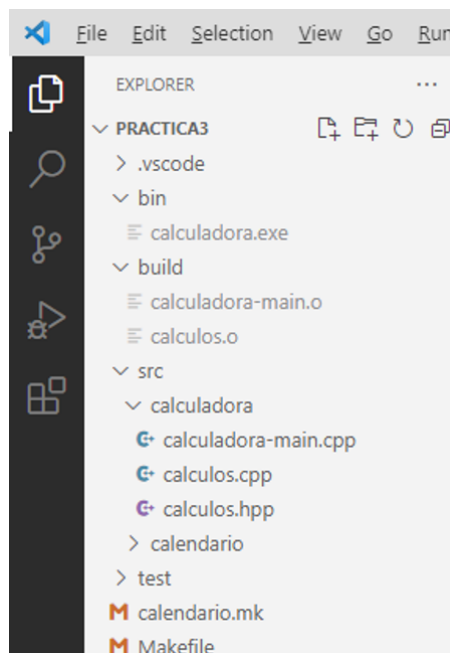
El tercer comando ha invocado al gestor de compilación «`make`», que, por defecto, ejecuta las reglas contenidas en el fichero denominado «`Makefile`», escribiendo en la pantalla los comandos que ejecuta y creando los ficheros objeto y el ejecutable.

Estos comandos son algo distintos a los que hemos ejecutado antes:

- El código fuente no está en el mismo directorio que el fichero «`Makefile`». Por eso, los nombres de los ficheros a compilar son «`src/calculadora/calculadora-main.cpp`» y «`src/calculadora/calculos.cpp`» en lugar de simplemente «`calculadora-main.cpp`» y «`calculos.cpp`» (se antepone a los nombres de los ficheros las rutas relativas de los directorios que los contienen).
- En el fichero «`Makefile`» se indica que los ficheros intermedios y el fichero ejecutable se creen en directorios específicos («`build`» y «`bin`», respectivamente). Esto es una buena práctica habitual en programación, que evita que ficheros intermedios y ejecutables se mezclen en el mismo directorio que el código fuente, tal y como nos ha ocurrido antes cuando hemos compila-

lado nosotros fichero a fichero<sup>1</sup>. Por ello, los ficheros intermedios son «build/calculos.o», «build/calculadora-main.o» y el fichero ejecutable es «bin/calculadora».

El fichero «Makefile» se ha configurado también para que, de ser necesario, se creen los directorios «build» y «bin» a través de los comandos `mkdir build` y `mkdir bin`. Estos dos comandos, heredados de Unix, permiten crear dos directorios con nombre «build» y «bin», respectivamente. Observa en el panel del explorador como, efectivamente, han aparecido estos directorios y que contienen los ficheros «calculos.o», «calculadora-main.o» y «calculadora» (o «calculadora.exe», en Windows):



- Se ha invocado al compilador `g++` con argumentos extra que modifican ligeramente su comportamiento:
  - g** indica al compilador que incluya información de depuración al generar los ejecutables o los ficheros intermedios de compilación. Esta información es necesaria y útil si luego se quiere utilizar el depurador.
  - Wall -Wextra** indica al compilador que muestre información sobre usos del lenguaje C++ en el código que se compila que, pese a ser sintácticamente correctos, plantean problemas que muy posiblemente resulten en posteriores errores de ejecución. Esta información es conocida como *advertencias* (*warnings*, en inglés).
  - Isrc/calculadora** indica al compilador que, cuando encuentre una cláusula de inclusión de un fichero (`#include`), si este no se encuentra en el mismo directorio que el fichero que se está compilando, busque dicho fichero también en el directorio «src/calculadora». La finalidad del argumento `-Itest/testing-prog1` es análoga.

Por lo demás, el comportamiento es el mismo que el visto en clase o en la primera compilación que hemos hecho:

- Los primeros comandos corresponden con las compilaciones individuales de cada uno de los ficheros de extensión «.cpp» del proyecto. En nuestro caso, estos comandos son:

<sup>1</sup>Cuando compruebes que el programa `make` crea los ficheros objeto y ejecutables en los directorios «build» y «bin», puedes borrar los ficheros objeto y ejecutable que has generado antes al compilar manualmente en el directorio «src/calculadora»: «calculos.o», «calculadora-main.o» y «calculadora.exe» o «calculadora», dependiendo de tu sistema operativo

- Una primera invocación para compilar el fichero «calculos.cpp»:  

```
g++ -g -Wall -Wextra -Isrc/calculadora -Itest/testing-prog1  
-c src/calculadora/calculos.cpp -o build/calculos.o
```
- Una segunda invocación para compilar el fichero «calculadora-main.cpp»:  

```
g++ -g -Wall -Wextra -Isrc/calculadora -Itest/testing-prog1  
-c src/calculadora/calculadora-main.cpp -o build/calculadora-main.o
```
- La última compilación corresponde a la creación del programa ejecutable a partir de los resultados intermedios de las compilaciones anteriores, así como la inclusión del código objeto de las bibliotecas estándares utilizadas. En nuestro caso, corresponde con el comando:  

```
g++ -g build/calculos.o build/calculadora-main.o -o bin/calculadora
```

Esta última invocación al compilador solo se realiza cuando las compilaciones correspondientes a los ficheros de código fuente (con la extensión «.cpp») no han detectado errores.
- Entre esas invocaciones al compilador, se han intercalado los comandos `mkdir build` y `mkdir bin` que han creado los directorios «build» y «bin», respectivamente.

Si tienes curiosidad, puedes consultar el contenido del fichero «Makefile» de este proyecto. Es más complejo que el fichero «Makefile» correspondiente al tema 7<sup>2</sup>, ya que hace uso de conceptos como variables y reglas implícitas que no se han visto en clase, pero que facilitan la escritura del mismo y su mantenimiento. En la página <https://www.gnu.org/software/make/manual/make.html> tienes a tu disposición el manual completo de la herramienta `make`, aunque en clase solo se han presentado los conceptos de las secciones 2.1 a 2.3 de dicho manual. Aprenderás más sobre esta herramienta en la asignatura de *Programación II* (y necesitarás conocerla en otras asignaturas de la carrera).

En esta práctica y en las siguientes, te proporcionaremos los ficheros «Makefile» ya preparados para su uso.

## Ejecución del programa

Una vez compilado el programa, puedes ejecutarlo desde la misma terminal, escribiendo la orden `bin/calculadora` (o si estás trabajando en un sistema operativo Windows 8 o anterior, `bin\calculadora`):

```
> bin/calculadora
MENÚ DE OPERACIONES
=====
0 - Finalizar
1 - Calcular el número de cifras de un entero
2 - Sumar las cifras de un entero
3 - Extraer una cifra de un entero
4 - Calcular la imagen especular de un entero
5 - Comprobar si un entero es primo

Seleccione una operación [0-5]:
...
```

<sup>2</sup><https://github.com/prog1-eina/tema-07-desarrollo-modular/blob/master/Makefile>

## Configuración de Visual Studio Code para compilar, ejecutar y depurar

En Visual Studio Code pueden definirse *tareas* que faciliten las operaciones de compilar, ejecutar y depurar los programas con los que estemos trabajando. La configuración de estas tareas se almacena en ficheros denominados «tasks.json» y «launch.json» que están ubicados en el directorio «.vscode».

Por ejemplo, en la práctica 2, para poder depurar un programa, Visual Studio Code necesitaba tener configuradas en el directorio «.vscode» dos tareas para compilar el código fuente del programa que está activo en el editor y para depurar ese mismo programa, que ya os facilitábamos creadas.

Para esta parte de la práctica, en el directorio «.vscode» ya se han definido un total de doce de estas tareas. En concreto, tres de ellas son para trabajar con el proyecto «calculadora» en la primera parte de esta práctica:

1. Compilar proyecto «calculadora».
2. Ejecutar proyecto «calculadora».
3. Depurar proyecto «calculadora».

Las dos primeras están definidas en el fichero «tasks.json», mientras que la última está definida en el fichero «launch.json». Puedes abrirlos en Visual Studio Code y ver su contenido, que pasamos a describir a continuación:

```
{
  "label": "Compilar proyecto «calculadora»",
  "type": "shell",
  "command": "make",
  "problemMatcher": ["$gcc"],
  "group": {
    "kind": "build",
    "isDefault": true
  },
},
{
  "label": "Ejecutar proyecto «calculadora»",
  "type": "shell",
  "command": "bin/calculadora",
  "windows": {
    "command": "chcp 65001 ; bin\\calculadora.exe",
  },
  "dependsOn": "Compilar proyecto «calculadora»",
  "problemMatcher": ["$gcc"],
},
```

**label** La propiedad label define el nombre de la tarea, tal y como se mostrará luego en la interfaz de usuario de Visual Studio Code. Para distinguir fácilmente las tareas predefinidas de Visual Studio Code de las tareas que definamos nosotros, hemos dado nombre a estas últimas en español.

**type** El tipo de la tarea. Las tareas que definamos nosotros van a ser siempre de tipo "shell", es decir, van a ser comandos que se van a ejecutar en una terminal.

**command** El comando que se ejecutará en la terminal cuando se ejecute a través de Visual Studio Code. En el caso de la tarea “Compilar proyecto «calculadora»”, el comando es una invocación a la herramienta make de GNU. En el caso de la tarea “Ejecutar proyecto «calculadora»”, el






comando es una invocación al programa ejecutable (bin/calculadora) que se ha creado a partir de nuestro código fuente tras invocar a make.

**windows** Define propiedades específicas a utilizar cuando la tarea se ejecuta en Windows. En el caso de la tarea “Ejecutar proyecto «calculadora»”, se está especificando que el comando a ejecutar sea en realidad la composición de dos:

1. `chcp 65001`: Este comando cambia la codificación de caracteres utilizada por el terminal, para que utilice la misma codificación que el editor del código fuente (UTF-8). De esta forma, cuando ejecutemos el programa, los caracteres acentuados y las ñes aparecerán correctamente en la terminal. En el tema 8 de la asignatura se darán más detalles sobre las distintas codificaciones de caracteres.
2. `bin\calculadora.exe`: La invocación al programa compilado, utilizando la sintaxis de Windows al hacerlo.

**Nota:** Si trabajas en una versión anterior a Windows 10, es muy probable que la sintaxis para la composición secuencial de los dos comandos no sea la adecuada. Además, en versiones anteriores a Windows 10, el soporte a la codificación UTF-8 del comando `chcp` es defectuoso. Si este es tu caso, modifica el valor de esta propiedad para que quede así:

```
"windows": {  
  "command": "bin\\calculadora.exe",  
},
```

**group** Define el grupo al que pertenece la tarea. En el caso de la tarea “Compilar proyecto «calculadora»”, se ha especificado que pertenece al grupo “build” y que es la tarea predeterminada. Esto hace que podamos ejecutarla en Visual Studio Code directamente a través del menú   o a través de la combinación de teclas  +  + .

**problemMatcher** Especifica cómo debe Visual Studio Code tratar los mensajes de error que produzca el comando que se ejecuta en la tarea. En nuestro caso, el valor será siempre `["$gcc"]`.

**dependsOn** Especifica dependencias entre las tareas. En nuestro caso, la tarea de ejecución del programa compilado (“Ejecutar proyecto «calculadora»”), requiere que antes se ejecute, sin errores, la tarea de compilación (“Compilar proyecto «calculadora»”).

La tarea “Depurar proyecto «calculadora»” se define en el fichero «launch.json» y tiene una estructura similar a las definidas en el fichero «tasks.json», aunque con más propiedades definidas, puesto que la tarea de depuración se integra más íntimamente con Visual Studio Code que las tareas de compilación y ejecución.

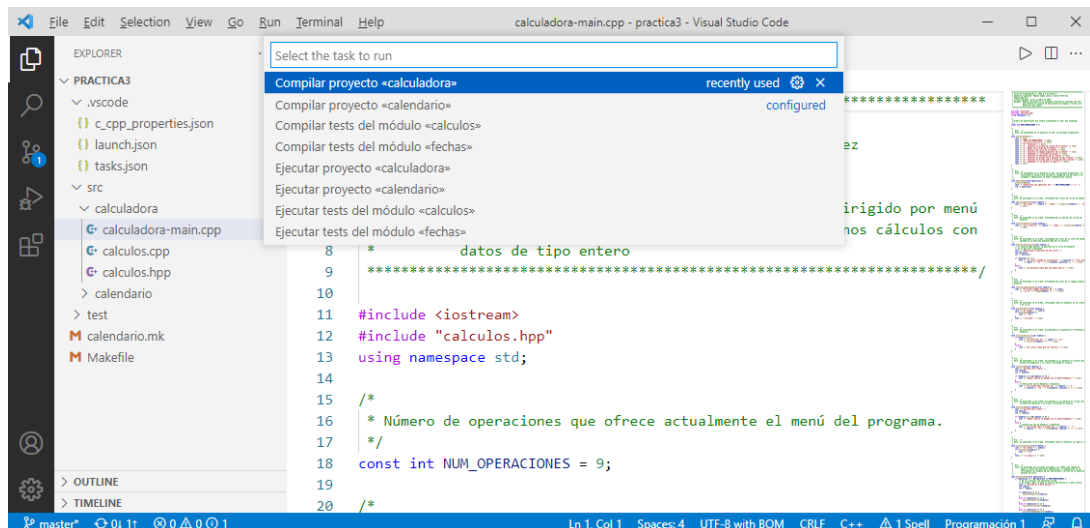
En esta y las siguientes prácticas, te daremos este tipo de tareas de Visual Studio Code ya configuradas y no os pediremos su entrega en Moodle. Puedes, por lo tanto, modificarlas a tu gusto para facilitarte el trabajo de compilar, ejecutar y depurar. Por ejemplo, si en lugar de la tarea de compilación, quieres convertir la tarea de ejecución del programa «calculadora» en la tarea predeterminada, puedes mover el bloque

```
"group": {  
  "kind": "build",  
  "isDefault": true  
},
```

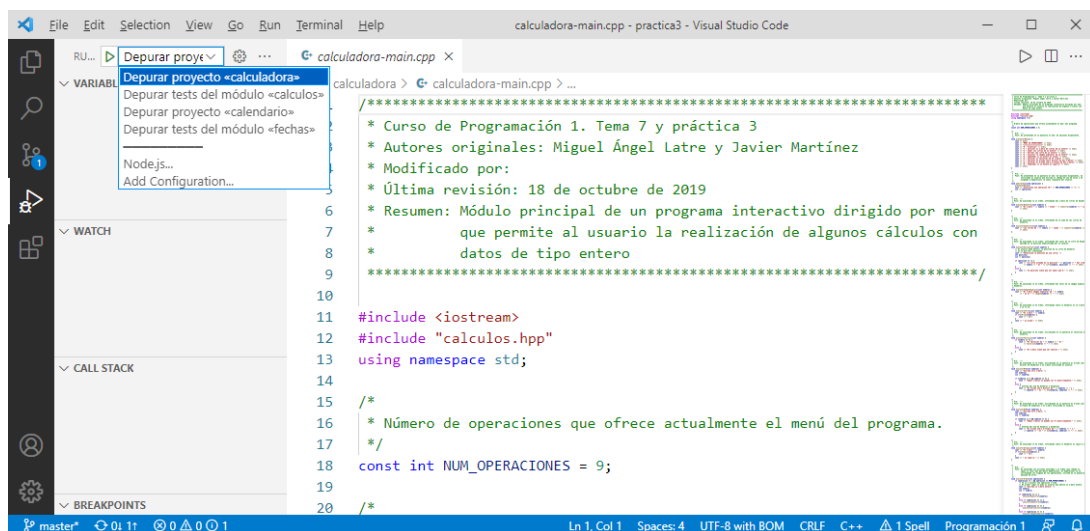


de la tarea “Compilar proyecto «calculadora»” a la tarea “Ejecutar proyecto «calculadora»”.

Se pueden ejecutar las tareas que se hayan configurado en el fichero «tasks.json» a través de la orden **Terminal** » **Run Task...**. Una vez seleccionada la opción de menú, aparecerá una lista con los nombres de las tareas configuradas en la parte superior de la ventana. Bastará entonces con seleccionar la tarea que queramos ejecutar:



Con respecto a las tareas de depuración, si hay más de una tarea definida, se puede seleccionar la tarea que se ejecuta en el depurador a través del desplegable de la parte superior del panel de depuración:



Si deseas saber más sobre la configuración de tareas en Visual Studio Code, te recomendamos que visites la documentación sobre tareas de Visual Studio Code que está disponible en <https://code.visualstudio.com/docs/editor/tasks>.

### 3.2.2. Primera modificación del módulo principal del proyecto «calculadora»

El módulo «calculos» declara y define siete funciones, pero solo cinco de ellas se utilizan en el programa principal. En concreto, las funciones factorial y mcd no se utilizan en el módulo principal.

Usar parcialmente un módulo no supone ningún problema para nuestros programas (por ejemplo, cuando utilizamos la biblioteca estándar «iostream» para poder leer y mostrar información del/al usuario, utilizamos solo una mínima parte de lo que hay definido en ella).

No obstante, modifica el programa para que presente nuevas opciones de menú que permitan calcular el factorial de un número positivo y el máximo común divisor de dos enteros no simultáneamente nulos, siguiendo el siguiente esquema de interacción con el usuario:

```
...
MENÚ DE OPERACIONES
=====
0 - Finalizar
1 - Calcular el número de cifras de un entero
2 - Sumar las cifras de un entero
3 - Extraer una cifra de un entero
4 - Calcular la imagen especular de un entero
5 - Comprobar si un entero es primo
6 - Calcular el factorial de un número
7 - Calcular el máximo común divisor de dos números

Seleccione una operación [0-7]: 6
Escriba un número entero: 7
El factorial de 7 es 5040.

MENÚ DE OPERACIONES
=====
...

Seleccione una operación [0-7]: 6
Escriba un número entero: -8
El número tiene que ser natural.

MENÚ DE OPERACIONES
=====
...

Seleccione una operación [0-7]: 7
Escriba un número entero: 48
Escriba otro número: 60
El máximo común divisor de 48 y 60 es 12.

MENÚ DE OPERACIONES
=====
...

Seleccione una operación [0-7]: 7
Escriba un número entero: 0
Escriba otro número: 0
Ambos números no pueden ser 0 simultáneamente.

MENÚ DE OPERACIONES
=====
...

Seleccione una operación [0-7]:
...
```

En la tarea correspondiente a esta sección, solo va a ser necesario que modifiques el fichero «calculadora-main.cpp». Evidentemente, no debes ponerte a escribir directamente código para calcular el máximo común divisor o el factorial, sino hacer uso de las funciones `mcd` y `factorial` del módulo «calculos», invocándolas desde el módulo principal.

### 3.2.3. Modificación del módulo «calculos» del proyecto «calculadora»

En la tarea correspondiente a esta sección, **vas a añadir dos funciones nuevas al módulo «calculos» para, posteriormente, añadir otras dos opciones al menú del programa principal.**

#### Adición de funciones al módulo «calculos»

Añade las siguientes funciones:

- Una función denominada `mcm` que, dados dos enteros no simultáneamente nulos, devuelva el entero positivo correspondiente al mínimo común múltiplo de ambos.
- Una función denominada `esCapicua` que, dado un número entero, devuelva el valor booleano `true` si y solo si el número es capicúa. Recuerda que un número se dice que es *capicúa* si se lee igual de izquierda a derecha que de derecha a izquierda (por ejemplo, los números 6996, 123321, etc.).

Se presentan a continuación las cabeceras de las funciones que se deben añadir:

```
/*
 * Pre:  a != 0 o b != 0
 * Post: Ha devuelto el mínimo común múltiplo de «a» y «b».
 */
unsigned mcm(int a, int b);

/*
 * Pre:  ---
 * Post: Ha devuelto true si y solo si el número «n» es capicúa cuando se escribe en base 10.
 */
bool esCapicua(int n);
```

Para añadir estas funciones, en el fichero «calculos.hpp» tienes que añadir las declaraciones de ambas funciones (cabeceras completas con tipo devuelto, nombre, lista de parámetros y especificación con precondition y poscondition).

Posteriormente, en el fichero «calculos.cpp» tienes que añadir las definiciones de dichas funciones (cabeceras y cuerpos). A la hora de escribir los cuerpos de estas funciones, puedes hacer uso de otras funciones del módulo. Eligiéndolas adecuadamente, el cuerpo de cada una de las funciones solicitadas puede constar de una única instrucción en la que invoques a otra función ya existente en el módulo «calculos». Por ejemplo, existe una propiedad matemática que relaciona el máximo común divisor y el mínimo común múltiplo de dos números  $a$  y  $b$ , que puede ser de mucha utilidad para diseñar de forma muy simple la función `mcm`:

$$a \cdot b = \text{mcm}(a, b) \cdot \text{mcd}(a, b)$$

#### Segunda modificación del módulo principal del proyecto «calculadora»

Modifica de nuevo el fichero «calculadora-main.cpp» para que presente otras dos opciones más en el menú que permitan calcular el mínimo común múltiplo de dos enteros no simultáneamente nulos y determinar si un número es capicúa o no, siguiendo el siguiente esquema de interacción con el usuario:

```
...
MENÚ DE OPERACIONES
=====
0 - Finalizar
1 - Calcular el número de cifras de un entero
2 - Sumar las cifras de un entero
3 - Extraer una cifra de un entero
4 - Calcular la imagen especular de un entero
5 - Comprobar si un entero es primo
6 - Calcular el factorial de un número
7 - Calcular el máximo común divisor de dos números
8 - Calcular el mínimo común múltiplo de dos números
9 - Comprobar si un entero es capicúa
```

```
Seleccione una operación [0-9]: 8
Escriba un número entero: 48
Escriba otro número: 18
El mínimo común múltiplo de 48 y 18 es 144.
```

```
MENÚ DE OPERACIONES
```

```
=====
```

```
...
```

```
Seleccione una operación [0-9]: 8
Escriba un número entero: 0
Escriba otro número: 0
Ambos números no pueden ser 0 simultáneamente.
```

```
MENÚ DE OPERACIONES
```

```
=====
```

```
...
```

```
Seleccione una operación [0-9]: 9
Escriba un número entero: 5246425
El número 5246425 es capicúa.
```

```
MENÚ DE OPERACIONES
```

```
=====
```

```
...
```

```
Seleccione una operación [0-9]: 9
Escriba un número entero: 12322
El número 12322 no es capicúa.
```

```
MENÚ DE OPERACIONES
```

```
=====
```

```
...
```

```
Seleccione una operación [0-9]:
```

```
...
```

### 3.2.4. Pruebas de las funciones `esCapicua` y `mcm` a través del proyecto «calculadora-test»

El trabajo de desarrollo de un programa no concluye hasta que se han realizado las pruebas necesarias para validar su buen comportamiento.

Las funciones del módulo «calculos» pueden probarse indirectamente a través de programas como, por ejemplo, el desarrollado en el proyecto «calculadora». Así, al ejecutar el programa «calculadora», se ejecutan las funciones `numCifras`, `sumaCifras`, `cifra`, `imagen` y `esPrimo` del módulo «calculos», y puede comprobarse el buen (o mal) funcionamiento de las mismas. Sin embargo, también es posible hacer pruebas directamente sobre las propias funciones a través de un programa o programas específicos que realicen invocaciones a las funciones y comprueben los resultados obtenidos, comparándolos con los resultados esperados.

Este es el caso del programa denominado «calculadora-test» cuyo código está en el directorio «test/calculadora». Este programa consta de:




- un módulo principal («calculos-test-main.cpp»);
- tres módulos de biblioteca:
  - el módulo «calculos-test» (que está compuesto por los ficheros «calculos-test.hpp» y «calculos-test.cpp»);
  - el módulo «calculos», con el que habéis trabajado en las secciones anteriores y que sigue compuesto por los ficheros «calculos.hpp» y «calculos.cpp» y cuyo código sigue ubicado en el directorio «src/calculadora»;
  - y el módulo «testing-prog1», que está compuesto por los ficheros «testing-prog1.hpp» y «testing-prog1.cpp» que están ubicados en el directorio «test/testing-prog1».

La composición del proyecto por estos cuatro módulos está definida en el fichero «Makefile», donde se establece que `calculos-test` depende de los ficheros intermedios «calculos.o», «calculos-test-main.o», «calculos-test.o» y «testing-prog1.o» resultantes de la compilación de los módulos mencionados anteriormente. El fichero «Makefile» ha sido configurado para que sea capaz de encontrar los ficheros de código fuente correspondientes a los módulos que hay que compilar en los directorios «src/calculadora», «test/calculadora» y «test/testing-prog1».

Abre el fichero «calculos-test.cpp» del directorio «test/calculadora». En él se hace uso de las funciones `esCapicua` y `mcm` y, por tanto, hay una directiva de inclusión al fichero de cabecera «calculos.hpp».

Como se ha indicado antes en la sección 3.2.1, se han definido tareas de Visual Studio Code para facilitar las tareas de compilación, ejecución y depuración de este programa de pruebas. Estas tareas son, en concreto:

1. Compilar tests del módulo «calculos».
2. Ejecutar tests del módulo «calculos».
3. Depurar tests del módulo «calculos».

Ejecuta la acción de ejecución de los test del módulo «calculos» ( ) y elige la opción , que provocará también la compilación del programa de pruebas. Este programa realizará varias invocaciones a las funciones `esCapicua` y `mcm` e informará de si el resultado de cada invocación es el correcto o no. Si al ejecutar el programa se detectara algún error en la ejecución de alguna de las funciones, te informará de ello a través de un mensaje como el siguiente:

Prueba `mcm(-10, 15)` incorrecta:

debería haber obtenido **30** pero ha calculado **31**.

Si este es el caso, corrige las funciones de que se trate hasta que el programa no detecte ningún error.

**Nota:** Si la terminal que utiliza Visual Studio Code en tu equipo no reconoce las secuencias de escape, verás el texto del mensaje de error rodeado de caracteres extraños. Si este fuera tu caso y estás en Windows 10 o Windows 11, prueba a ejecutar el fichero de modificación del registro «habilitar-secuencias-escape-windows.reg» disponible en Moodle<sup>3</sup>.

Si no estás en Windows o sigue sin funcionar, localiza el fichero «test/testing-prog1/testing-prog1.cpp» y modifica las líneas 27 a 29 para que queden como sigue:

```
const string ESCAPE_ROJO = " ";  
const string ESCAPE_ROJO_NEGRITA = " ";  
const string ESCAPE_NORMAL = " ";
```

El programa «calculos-test» permite validar parcialmente el trabajo que hayas desarrollado en el módulo «calculos» con respecto a las funciones `esCapicua` y `mcm` y, en su caso, detectar y corregir errores. Puedes echar un vistazo al código de las funciones que hacen las pruebas («calculos-test.cpp») y a los casos de prueba que se utilizan («calculos-test-main.cpp»).

En todo caso, que el programa de pruebas no detecte errores no es garantía de que no los haya.

### 3.2.5. Estructura del proyecto «calendario»

Entre el contenido del directorio «practica3» que te has descargado de GitHub verás que hay también un fichero denominado «calendario.mk» y dos directorios denominados «calendario» dentro de los directorios «src» y «test», on los que vamos a trabajar en esta segunda parte de la práctica.

La estructura es similar a la del proyecto «calculadora»:

- El código del programa se encuentra ubicado en el directorio «calendario» de la carpeta «src» y consta de un módulo principal y de un módulo de biblioteca (el módulo «fechas»), organizados en un total de tres ficheros de código fuente:
  - Fichero «calendario-main.cpp», con el código del módulo principal del programa.
  - Fichero «fechas.hpp» de interfaz del módulo «calendario».
  - Fichero «fechas.cpp» de implementación del módulo «calendario».
- El código de un programa de pruebas del módulo «fechas» está ubicado en el directorio «test/calendario» y consta de:
  - El módulo principal del programa de pruebas (fichero «fechas-test-main.cpp»).
  - Un módulo denominado «fechas-test», que realiza pruebas de las funciones declaradas en la biblioteca «fechas». Este módulo está formado por el fichero de interfaz «fechas-test.hpp» y su correspondiente fichero de implementación «fechas-test.cpp».
  - El módulo denominado «fechas», ubicado en el directorio «src/calendario».
  - El módulo «testing-prog1», compuesto por los ficheros «testing-prog1.hpp» y «testing-prog1.cpp» que están ubicados en el directorio «test/testing-prog1». Se trata del mismo módulo que se utilizó en el programa de pruebas «calculos-test».

- El fichero «calendario.mk», que contiene las reglas necesarias para que la herramienta «make» invoque al compilador «g++» de forma que se puedan generar los programas «calendario» y «fechas-test».
- En el directorio «.vscode», los ficheros «tasks.json» y «launch.json» ya tenían configuradas seis tareas para compilar, ejecutar y depurar los dos programas mencionados.

### 3.2.6. Módulo de biblioteca «fechas» del proyecto «calendario»

Utiliza la tarea “Compilar tests del módulo «fechas»”. El proyecto «fechas-test» debería compilar sin errores, pero con advertencias y, al ser ejecutado, informará de resultados incorrectos en la práctica totalidad de las invocaciones a las funciones del módulo «fechas». Esto último es lo esperable, puesto que el fichero «fechas.cpp» se ha suministrado sin completar su código.

Implementa las funciones que faltan en «fechas.cpp». Hazlo incrementalmente: comienza por la primera función y, antes de pasar a la siguiente, ejecuta de nuevo el proyecto «fechas-test» y asegúrate de que el programa obtiene los resultados esperados en todas las invocaciones a dicha función implementada.

La especificación de las funciones del módulo de biblioteca «fechas» se presenta a continuación, para que puedan servirte de referencia.

```
/*
 * Año a partir del cual la función «diaDeLaSemana» va a poder ser invocada, aprovechando el
 * dato de que el 1 de enero de 1900 fue lunes.
 */
const unsigned AGNO_INICIAL = 1900;

/*
 * Pre: La terna de parámetros «dia», «mes» y «agno» definen una fecha válida del calendario
 * gregoriano, la fecha «dia/mes/agno».
 * Post: El valor del parámetro «f», al ser escrito en base 10, tiene un formato de ocho
 * dígitos «aaaammdd» que representa la fecha «dia/mes/agno» donde los dígitos «aaaa»
 * representan el año de la fecha, los dígitos «mm», el mes y los dígitos «dd», el día.
 */
void componer(unsigned dia, unsigned mes, unsigned agno, unsigned& f);

/*
 * Pre: El valor de «f» escrito en base 10 tiene la forma «aaaammdd» donde los dígitos «aaaa»
 * representan el año de una fecha válida del calendario gregoriano, los dígitos «mm», el
 * mes y los dígitos «dd», el día.
 * Post: Los valores de los parámetros «dia», «mes» y «agno» son iguales, respectivamente, al
 * día, al mes y al año de la fecha «f».
 */
void descomponer(unsigned f, unsigned& dia, unsigned& mes, unsigned& agno);

/*
 * Pre: Los valores de los parámetros «f1» y «f2» escritos en base 10 tienen la forma
 * «aaaammdd», donde los dígitos «aaaa» representan el año, los dígitos «mm», el mes y
 * los dígitos «dd» el día de sendas fechas del calendario gregoriano.
 * Post: Ha devuelto true si y solo si la fecha representada por el valor del parámetro «f1» es
 * anterior a la representada por «f2».
 */
bool esAnterior(unsigned f1, unsigned f2);
```

```
/*
 * Pre: 1 <= dia <= 31, 1 <= mes <= 12, agno > 1582 y la fecha formada por «dia», «mes» y
 *      «agno» representan una fecha válida del calendario gregoriano.
 * Post: Tras la ejecución de la función, los parámetros «fecha», «dia», «mes» y «agno»
 *      representan la fecha correspondiente al día siguiente al que representaban al
 *      iniciarse la ejecución de la función.
 *
 *      Por ejemplo, si d, m y a son variables de tipo entero y d = 17, m = 10 y a = 2019,
 *      tras la invocación diaSiguiente(d, m, a) los valores de las variables serían
 *      d = 18, m = 10 y a = 2019.
 *      Si los valores fueran d = 29, m = 2 y a = 2020, tras la invocación
 *      diaSiguiente(d, m, a) los valores serían d = 1, m = 3 y a = 2020.
 *      Si los valores fueran d = 31, m = 12 y a = 2022, tras la invocación
 *      diaSiguiente(d, m, a) los valores serían d = 1, m = 1 y a = 2023.
 */
void diaSiguiente(unsigned& dia, unsigned& mes, unsigned& agno);

/*
 * Pre: agno > 1582.
 * Post: Ha devuelto true si y solo si el año «agno» es bisiesto de acuerdo con las reglas del
 *      calendario gregoriano.
 */
bool esBisiesto(unsigned agno);

/*
 * Pre: 1 <= mes <= 12 y agno > 1582.
 * Post: Ha devuelto el número de días del mes correspondiente al parámetro «mes» del año
 *      correspondiente al parámetro «agno».
 *      Por ejemplo: diasDelMes(10, 2018) devuelve 31,
 *                  diasDelMes(2, 2018) devuelve 28 y
 *                  diasDelMes(2, 2020) devuelve 29.
 */
unsigned diasDelMes(unsigned mes, unsigned agno);

/*
 * Pre: agno > 1582.
 * Post: Ha devuelto el número de días que tiene el año «agno».
 *      Por ejemplo: diasDelAgno(2018) devuelve 365 y
 *                  diasDelAgno(2020) devuelve 366.
 */
unsigned diasDelAgno(unsigned agno);

/*
 * Pre: 1 <= dia <= 31, 1 <= mes <= 12 y agno > 1582 y la fecha formada por «dia/mes/agno»
 *      es una fecha válida del calendario gregoriano.
 * Post: Ha devuelto el número de día del año de la fecha formada por «dia/mes/agno».
 *      Por ejemplo: diaEnElAgno(1, 1, 2019) devuelve 1;
 *                  diaEnElAgno(31, 12, 2018) devuelve 365;
 *                  diaEnElAgno(1, 2, 2019) devuelve 32 y
 *                  diaEnElAgno(31, 12, 2020) devuelve 366.
 */
unsigned diaEnElAgno(unsigned dia, unsigned mes, unsigned agno);
```



```

/*
 * Pre: Los valores de los parámetros «dia», «mes» y «agno» representan conjuntamente una
 * fecha válida del calendario gregoriano igual o posterior al 1 de enero de 1900.
 * Post: Ha devuelto un entero que codifica el día de la semana de la fecha representada por
 * los valores de los parámetros «dia», «mes» y «agno», de acuerdo con la siguiente
 * codificación: el 0 codifica el lunes, el 1 codifica martes y así sucesivamente hasta
 * el 6, que codifica el domingo.
 */
unsigned diaDeLaSemana(unsigned dia, unsigned mes, unsigned agno);

```

### 3.2.7. Escritura del calendario de un mes

Una vez que el módulo «fechas» ha sido desarrollado y probado, vamos a centrarnos en el proyecto «calendario» para desarrollar en él el programa planteado en la clase de problemas correspondiente a desarrollo modular<sup>4</sup>.

Se desea disponer de un programa que escriba en la pantalla el calendario de un determinado mes de un año que se solicita al usuario. El programa debe asegurarse de que el mes introducido está comprendido entre 1 y 12 y de que el año es igual o posterior a 1900, volviendo a solicitar los datos al usuario el número de veces que sea necesario hasta que cumplan las restricciones establecidas.

Se facilita el dato que el 1 de enero de 1900 fue lunes.

A continuación, se muestra un ejemplo de una posible ejecución del mismo, donde puede observarse tanto la petición y lectura de datos como la escritura del calendario elegido.

```

Introduzca el mes [1-12]: 13
El mes debe estar comprendido entre 1 y 12: -8
El mes debe estar comprendido entre 1 y 12: 0
El mes debe estar comprendido entre 1 y 12: 11
Introduzca un año igual o posterior a 1900: 1492
El año debe ser igual o posterior a 1900: 1899
El año debe ser igual o posterior a 1900: 2019

```

```

                NOVIEMBRE 2019
    L   M   X   J   V   S   D
-----
                1   2   3
    4   5   6   7   8   9  10
  11  12  13  14  15  16  17
  18  19  20  21  22  23  24
  25  26  27  28  29  30

```

Escribe el código de la función main correspondiente a este programa en el fichero «calendario-main.cpp». Aprovecha la descomposición modular que se hizo en la clase de problemas y apóyate en funciones que vayan resolviendo problemas cada vez más simples. Haz uso libre de las funciones del módulo «fechas», pero **no modifiques el fichero de interfaz «fechas.hpp»**<sup>5</sup>. Si tu

<sup>4</sup>Disponible en <https://miguel-latre.github.io/transparencias/pbs-tema-07-diseno-descendente.pdf>

<sup>5</sup>Lee la nota adicional a este respecto en la sección 3.3.

función `main` necesita de funciones auxiliares que no están en el módulo «`fechas`», añádelas al fichero «`calendario-main.cpp`».

### 3.3. Entrega de la práctica

Deberán subirse a Moodle los ficheros «`calculadora-main.cpp`», «`calculos.cpp`» y «`calculos.hpp`» del proyecto «`calculadora`» y los ficheros «`calendario-main.cpp`» y «`fechas.cpp`» del proyecto «`calendario`».

**El fichero «`fechas.hpp`» no hay que subirlo a Moodle, puesto que se ha indicado que no debe ser modificado** (ni añadiendo funciones en el mismo, ni modificando las cabeceras de las ya existentes). Si, pese a la indicación realizada, lo modificas, el resultado de la corrección que realicemos los profesores probablemente terminará en error de compilación en los proyectos «`calendario`» o «`fechas-test`», lo que acarreará una calificación de 0 en esa parte.

La entrega deberá realizarse antes del **sábado 28 de octubre a las 18:00**.